

C# Language & .NET Platform

10th Lecture:

C# Type System

<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.msdnaa.net/curriculum/license_curriculum.aspx)

Problems Without Generic Types

Assume we need a class that can work with arbitrary objects

```
class Buffer {  
    private object[] data;  
    public void Put(object x) {...}  
    public object Get() {...}  
}
```

Problems

- Type casts needed

```
buffer.Put(3); // boxing imposes run-time costs  
int x = (int) buffer.Get(); // type cast (unboxing) imposes run-time costs
```

- One cannot statically enforce homogeneous data structures

```
buffer.Put(3); buffer.Put(new Rectangle());  
Rectangle r = (Rectangle) buffer.Get(); // can result in a run-time error!
```

- Special types IntBuffer, RectangleBuffer, ... introduce redundancy

Generic Class Buffer

generic type

placeholder type

```
class Buffer<Element> {  
    private Element[] data;  
    public Buffer(int size) {...}  
    public void Put(Element x) {...}  
    public Element Get() {...}  
}
```

- works also for structs and interfaces
- placeholder type *Element* can be used like a normal type

Usage

```
Buffer<int> a = new Buffer<int>(100);  
a.Put(3); // accepts only int parameters; no boxing  
int i = a.Get(); // no type cast needed!
```

```
Buffer<Rectangle> b = new Buffer<Rectangle>(100);  
b.Put(new Rectangle()); // accepts only Rectangle parameters  
Rectangle r = b.Get(); // no typ cast needed!
```

Benefits

- homogeneous data structure with compile-time type checking
- efficient (no boxing, no type casts)

Generics in Java
Templates in C++

What Happens Behind the Scene (in C#/.NET)?

```
class Buffer<Element> {...}
```

Compiler generates CIL code for class Buffer with a placeholder for Element.

Instantiation with value types

```
Buffer<int> a = new Buffer<int>();
```

At run time the CLR generates a new class Buffer<int> in which Element is replaced with int.

```
Buffer<int> b = new Buffer<int>();
```

Uses existing Buffer<int>.

```
Buffer<float> c = new Buffer<float>();
```

At run time the CLR generates a new class Buffer<float> in which Element is replaced with float.

Instantiation with reference types

```
Buffer<string> a = new Buffer<string>();
```

At run time the CLR generates a new “class” impl. Buffer<object> which can work with all reference types.

```
Buffer<string> b = new Buffer<string>();
```

Uses existing Buffer<object>.

```
Buffer<Node> b = new Buffer<Node>();
```

Uses existing Buffer<object> code, but CLR generates new “description”, i.e. Method Table, for Buffer<Node> class.

Generic Buffer in ILDASM

TestOfGenerics.exe - IL DASM

File View Help

- TestOfGenerics.exe
 - MANIFEST
 - TestOfGenerics
 - TestOfGenerics.Buffer`1 <Element>
 - .class private auto ansi beforefieldinit
 - data : private !Element []
 - .ctor : void(int32)
 - Get : !Element()
 - Put : void(!Element)
 - TestOfGenerics.Program
 - .class private auto ansi beforefieldinit
 - .ctor : void()
 - Main : void(string[])

.assembly TestOfGenerics
{

Multiple Placeholder Types



Buffer with priorities

```
class Buffer <Element, Priority> {  
    private Element[] data;  
    private Priority[] prio;  
    public void Put(Element x, Priority prio) {...}  
    public void Get(out Element x, out Priority prio) {...}  
}
```

Usage

```
Buffer<int, int> a = new Buffer<int, int>();  
a.Put(100, 0);  
int elem, prio;  
a.Get(out elem, out prio);
```

```
Buffer<Rectangle, double> b = new Buffer<Rectangle, double>();  
b.Put(new Rectangle(), 0.5);  
Rectangle r; double prio;  
b.Get(out r, out prio);
```

C++ allows also placeholders for constants, C# does not

Genericity and Inheritance

```
class Buffer <Element>: List<Element> {  
    ...  
    public void Put(Element x) {  
        this.Add(x); // Add is inherited from List  
    }  
}
```

can also implement generic interfaces

From which classes may a generic class be derived?

- from a non-generic class

```
class T<X>: B {...}
```

- from an instantiated generic class

```
class T<X>: B<int> {...}
```

- from a generic class
with the same placeholder

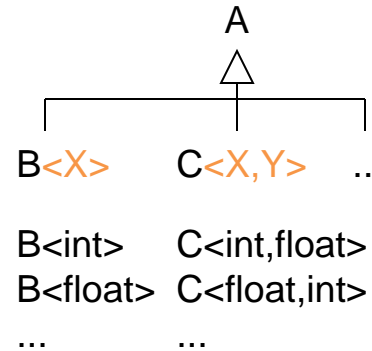
```
class T<X>: B<X> {...}
```

Assignment Compatibility

Assigning T<x> to a non-generic base class

```
class A {...}
class B<X>: A {...}
class C<X,Y>: A {...}
```

```
A a1 = new B<int>();
A a2 = new C<int, float>();
```



Assigning T<x> to a generic base class

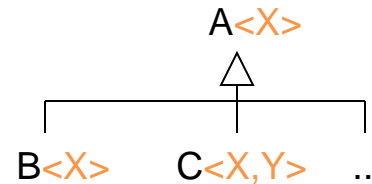
```
class A<X> {...}
class B<X>: A<X> {...}
class C<X,Y>: A<X> {...}
```

```
A<int> a1 = new B<int>();
A<int> a2 = new C<int, float>();
```

```
A<int> a3 = new B<short>();
```

ok, if corresponding placeholders are replaced by the same type

illegal



Overriding Methods

```
class Buffer<Element> {  
    ...  
    public virtual void Put(Element x) {...}  
}
```

Methods inherited from an instantiated generic class

```
class MyBuffer: Buffer<int> {  
    ...  
    public override void Put(int x) {...}  
}
```

Element is replaced by the concrete type int

Methods inherited from a generic class

```
class MyBuffer<Element>: Buffer<Element> {  
    ...  
    public override void Put(Element x) {...}  
}
```

Element remains to be a placeholder

The following is **illegal** (it is not allowed to inherit a placeholder)

```
class MyBuffer: Buffer<Element> {  
    ...  
    public override void Put(Element x) {...}  
}
```

Run-time Type Checks

Instantiated generic types can be used like non-generic types

```
Buffer<int> buf = new Buffer<int>(20);  
object obj = buf;
```

```
if (obj is Buffer<int>)  
    buf = (Buffer<int>) obj;
```

```
Type t = typeof(Buffer<int>);  
Console.WriteLine(t.Name); // => Buffer'1[System.Int32]
```

Reflection yields also the concrete types substituted for the placeholders!

Constraints

Constraints about placeholder types are specified as base types

interface or base class

```
class OrderedBuffer <Element, Priority> where Priority: IComparable {  
    Element[] data;  
    Priority[] prio;  
    int lastElem;  
    ...  
    public void Put(Element x, Priority p) {  
        int i = lastElem;  
        while (i >= 0 && p.CompareTo(prio[i]) > 0) {data[i+1] = data[i]; prio[i+1] = prio[i]; i--;}  
        data[i+1] = x; prio[i+1] = p;  
    }  
}
```

Allows operations on instances of placeholder types

Usage

```
OrderedBuffer<int, int> a = new OrderedBuffer<int, int>();  
a.Put(100, 3);
```

parameter must implement IComparable

Multiple Constraints

```
class OrderedBuffer <Element, Priority>  
  where Element: MyClass  
  where Priority: IComparable, ISerializable {  
  ...  
  public void Put(Element x, Priority p) {...}  
  public void Get(out Element x, out Priority p) {...}  
}
```

Usage

must be a subclass of MyClass

must implement IComparable and ISerializable

```
OrderedBuffer<MySubclass, MyPrio> a = new OrderedBuffer<MySubclass, MyPrio>();  
...  
a.Put(new MySubclass(), new MyPrio(100));
```

Constructor Constraint

For creating objects of a generic types

```
class Stack<T, E> where E: Exception, new() {  
    T[] data = ...;  
    int top = -1;  
  
    public void Push(T x) {  
        if (top >= data.Length)  
            throw new E();  
        else  
            data[++top] = x;  
    }  
}
```

specifies that the placeholder *E* must have a parameterless constructor.

Usage

```
class MyException: Exception {  
    public MyException(): base("stack overflow or underflow") {}  
}
```

```
Stack<int, MyException> stack = new Stack<int, MyException>();  
...  
stack.Push(3);
```

Overview of Constraints



Constraint

where T : struct

Description

The type argument must be a **value type**. Any value type except *Nullable* can be specified.

where T : class

The type argument must be a **reference type**, including any class, interface, delegate, or array type.

where T : new()

The type argument **must have a public parameterless constructor**. When used in conjunction with other constraints, the `new()` must be specified last.

where T : BaseClassName

The type argument **must be or derive from** the specified **base class**.

where T : InterfaceName

The type argument **must be or implement** the specified **interface**. Multiple interface constraints can be specified. The constraining interface can also be generic.

where T : U

The **type argument** supplied for **T** **must be or derive from** the **type argument** supplied for **U**. This is called a naked type constraint.

Generic Methods

Methods that can work with arbitrary data types

```
static void Sort<T> (T[] a) where T: IComparable {  
    for (int i = 0; i < a.Length-1; i++) {  
        for (int j = i+1; j < a.Length; j++) {  
            if (a[j].CompareTo(a[i]) < 0) {  
                T x = a[i]; a[i] = a[j]; a[j] = x;  
            }  
        }  
    }  
}
```

can sort any array as long as the array elements implement *IComparable*

Usage

```
int[] a = {3, 7, 2, 5, 3};  
...  
Sort<int>(a); // a == {2, 3, 3, 5, 7}
```

```
string[] s = {"one", "two", "three"};  
...  
Sort<string>(s); // s == {"one", "three", "two"}
```

From the method parameters the compiler can usually infer the concrete type that is to be substituted for the placeholder type; so one can simply write:

```
Sort(a); // a == {2, 3, 3, 5, 7}
```

```
Sort(s); // s == {"one", "three", "two"}
```

Null Values in Generic Types/Methods

Setting a value to null

```
void Foo<T>() {  
    T x = null;           // error  
    T y = 0;             // error  
    T z = default(T);    // ok! 0, '\0', false, null  
}
```

Comparing a value against null

```
void Foo<T>(T x) {  
    if (x == null) {  
        Console.WriteLine(true);  
    } else {  
        Console.WriteLine(false);  
    }  
}
```

for reference types $x == null$ does a comparison
for nullable types $x == null$ does a comparison
for value types $x == null$ returns *false*

```
Foo(3);           // false  
Foo(0);           // false  
Foo("Hello");    // false  
Foo<string>(null); // true
```

Trick: How to declare a tree of chars using Dictionary<K, V>?



Trick: How to declare a “recursive” type?

- If something like:

```
Dictionary<char, Dictionary<char, Dictionary<char, ... >>> treeRoot;
```

- is needed ...

Trick: How to declare a “recursive” type?

- If something like:

```
Dictionary<char, Dictionary<char, Dictionary<char, ... >>> treeRoot;
```

- is needed, it can be declared as:

```
class Node : Dictionary<char, Node> {  
  
    // Disadvantage:  
    // All Dictionary<K, K> constructors other than Dictionary<char, Node>()  
    // need to be explicitly recreated in Node class (with at least with an empty body):  
  
    public Node(xxx) : base(xxx) {}  
    public Node(yyy) : base(yyy) {}  
    ...  
}
```

Indexers (parametric properties)

Programmable operator for indexing a collection

```
class File {
    FileStream s;

    public int this [int index] {
        get { s.Seek(index, SeekOrigin.Begin);
              return s.ReadByte();
            }
        set { s.Seek(index, SeekOrigin.Begin);
              s.WriteByte((byte) value);
            }
    }
}
```

Usage

```
File f = ...;
int x = f[10]; // calls f.get(10)
f[10] = 'A'; // calls f.set(10, 'A')
```

- get or set method can be omitted (write-only / read-only)
- indexers can be overloaded with different index type
- .NET library has indexers for *string* (s[i]), *List* (a[i]), etc.

Indexers (other example 1)

```
class MonthlySales {
    int[] apples = new int[12];
    int[] bananas = new int[12];
    ...
    public int this[int month] {           // set method omitted => read-only
        get { return apples[month-1] + bananas[month-1]; }
    }

    public int this[string month] {       // overloaded read-only indexer
        get {
            switch (month) {
                case "Jan": return apples[0] + bananas[0];
                case "Feb": return apples[1] + bananas[1];
                ...
            }
        }
    }
}
```

```
MonthlySales sales = new MonthlySales();
...
Console.WriteLine(sales[1] + sales["Feb"]);
```

Indexers (other example 2)

```
class MonthlySales {
    int[] apples = new int[12];
    int[] bananas = new int[12];
    ...
    public int this[int month] {           // set method omitted => read-only
        get { return apples[month-1] + bananas[month-1]; }
    }

    public int this[int month, string kind] { // overloaded read-only indexer
        get {
            switch (kind) {
                case "apple": return apples[month-1];
                case "banana": return bananas[month-1];
                ...
            }
        }
    }
}
```

```
MonthlySales sales = new MonthlySales();
```

```
...
Console.WriteLine(sales[1]);
Console.WriteLine(sales[1, "banana"]);
Console.WriteLine(sales[1, "apple"]);
```

What is the output the following program?

```
struct S {
    public int x;
    public int y;
    public override string ToString() {
        return string.Format("x={0},y={1}", x, y);
    }
}

class Program {
    static void Main(string[] args) {
        S[] arr = new S[1];
        arr[0] = new S();
        arr[0].x = 10;
        Console.Write(arr[0]);

        List<S> list = new List<S>();
        list.Add(new S());
        list[0].x = 20;
        Console.Write(list[0]);
    }
}
```

Option	Result
A	(x=10,y=0)(x=20,y=0)
B	(x=10,y=0)(x=0,y=0)
C	<i>It will generate a runtime error (unhandled exception).</i>
D	<i>It will not compile.</i>

What is the output the following program?

```
struct S {
    public int x;
    public int y;
    public override string ToString() {
        return string.Format("x={0},y={1}", x, y);
    }
}

class Program {
    static void Main(string[] args) {
        S[] arr = new S[1];
        arr[0] = new S();
        arr[0].x = 10;
        Console.Write(arr[0]);

        List<S> list = new List<S>();
        list.Add(new S());
        list[0].x = 20;
        Console.Write(list[0]);
    }
}
```

Option	Result
A	(x=10,y=0)(x=20,y=0)
B	(x=10,y=0)(x=0,y=0)
C	<i>It will generate a runtime error (unhandled exception).</i>
D	<i>It will not compile: "Cannot modify the return value of 'System.Collections.Generic.List<S>.this[int]'</i> because it is not a variable"

Implementing Generic Interfaces

```
interface I<T> {  
    void m(T x);  
    T m();  
}  
  
class A : I<int>, I<long> {  
  
    ?  
  
}
```

Implementing Generic Interfaces

```
interface I<T> {
    void m(T x);
    T m();
}

class A : I<int>, I<long> {
    public void m(int x) { Console.WriteLine("int m"); }
    public void m(long x) { Console.WriteLine("long m"); }
    public int m() { return -1; }
    long I<long>.m() { return -2; }
}

class Program {
    static void Main(string[] args) {
        A a = new A();

        a.m(0);
        a.m(0L);    // Same as a.m(0L);
        ((I<int>) a).m(0);
        ((I<long>) a).m(0);

        Console.WriteLine(a.m());
        Console.WriteLine(((I<long>) a).m());
    }
}
```

Implementing Generic Interfaces

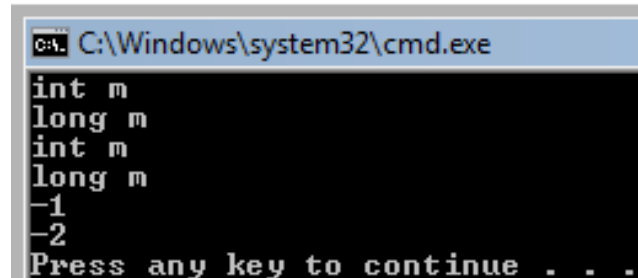
```
interface I<T> {
    void m(T x);
    T m();
}

class A : I<int>, I<long> {
    public void m(int x) { Console.WriteLine("int m"); }
    public void m(long x) { Console.WriteLine("long m"); }
    public int m() { return -1; }
    long I<long>.m() { return -2; }
}

class Program {
    static void Main(string[] args) {
        A a = new A();

        a.m(0);
        a.m(0L);    // Same as a.m(0L);
        ((I<int>) a).m(0);
        ((I<long>) a).m(0);

        Console.WriteLine(a.m());
        Console.WriteLine(((I<long>) a).m());
    }
}
```




```
C:\Windows\system32\cmd.exe
int m
long m
int m
long m
-1
-2
Press any key to continue . . .
```

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>

IEnumerable and IEnumerator (2)

- 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 (3)

- Classes which implement IEnumerable are:
 - Array
 - String
 - and many more.
- For all IEnumerable's foreach-statement can be used

Example:

```
int[] a = {1, 6, 8, 9, 15}; // object of abstract type Array
foreach (int i in a) System.Console.WriteLine(i);
```

Enumerators

foreach loop can be applied to objects of classes which implement IEnumerable

```
class MyClass: IEnumerable<T> {  
    ...  
    public IEnumerator<T> GetEnumerator() {  
        return new MyEnumerator(...);  
    }  
  
    private class MyEnumerator: IEnumerator<string> {  
        public string Current { get {...} }  
        public bool MoveNext() {...}  
        public void Reset() {...}  
    }  
}
```

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

```
MyClass x = new MyClass();  
...  
foreach (string s in x) ...
```

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);  
    ...  
}
```