

C# Language & .NET Platform

14th Lecture

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

Department of
Distributed and
Dependable
Systems



Pavel Ježek

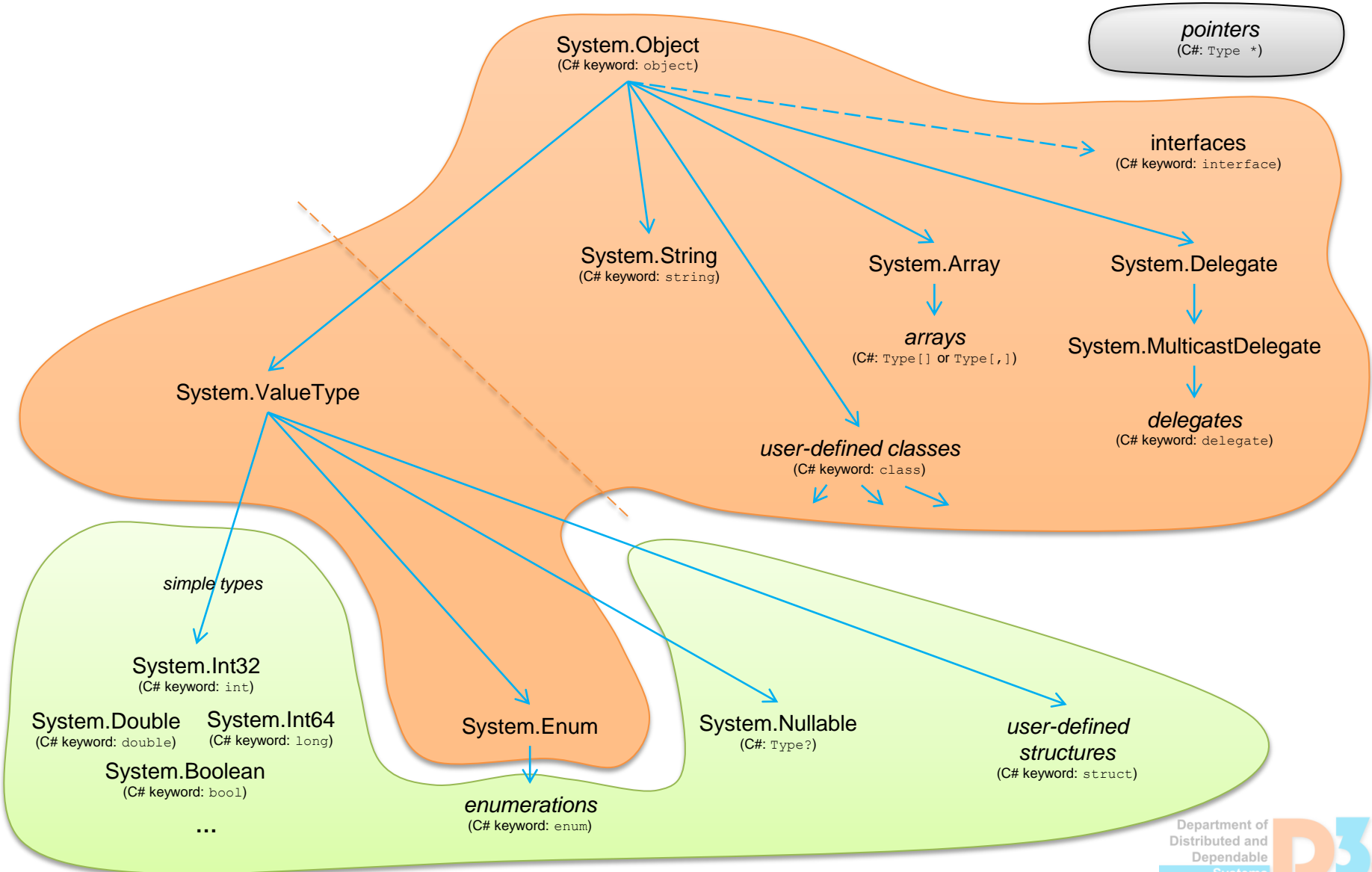
pavel.jezek@d3s.mff.cuni.cz



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)

CLI Type Inheritance

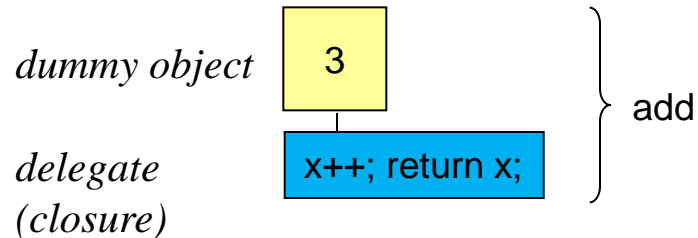


Outer Variables

If anonymous methods access variables of the enclosing method these variables are evacuated into a dummy object (capturing) – all anonymous methods and the enclosing method itself are then using a single “evacuated” variable (see next slide).

```
delegate int Adder();
```

```
class Test {  
    static Adder CreateAdder() {  
        int x = 0;  
        return delegate { x++; return x; };  
    }  
    static void Main() {  
        Adder add = CreateAdder();  
        Console.WriteLine(add());  
        Console.WriteLine(add());  
        Console.WriteLine(add());  
    }  
}
```



The dummy object lives as long as the delegate object

Output:

```
1  
2  
3
```

Anonymous methods in C# always exist only as closures, never as “lambda functions” with free variables (open bindings)!

Lambda Expressions

- Example of C# 2.0 anonymous method:

```
class Program {
    delegate T BinaryOp<T>(T x, T y);

    static void Method<T>(BinaryOp<T> op, T p1, T p2) {
        Console.WriteLine(op(p1, p2));
    }

    static void Main() {
        Method(delegate(int a, int b) { return a + b; }, 1, 2);
    }
}
```

- **C# 3.0** lambda expressions provide further simplification:

```
class Program {
    delegate T BinaryOp<T>(T x, T y);

    static void Method<T>(BinaryOp<T> op, T p1, T p2) {
        Console.WriteLine(op(p1, p2));
    }

    static void Main() {
        Method((a, b) => a + b, 1, 2);
    }
}
```

Lambda Expressions (2)

- Expression or statement body
- Implicitly or explicitly typed parameters
- Examples:

```
x => x + 1 // Implicitly typed, expression body
x => { return x + 1; } // Implicitly typed, statement body
(int x) => x + 1 // Explicitly typed, expression body
(int x) => { return x + 1; } // Explicitly typed, statement body
(x, y) => x * y // Multiple parameters
() => Console.WriteLine() // No parameters
```

- A lambda expression is a value, that does not have a type but can be implicitly converted to a compatible delegate type

```
delegate R Func<A,R>(A arg);
Func<int,int> f1 = x => x + 1; // Ok
Func<int,double> f2 = x => x + 1; // Ok
Func<double,int> f3 = x => x + 1; // Error – double cannot be
// implicitly converted to int
```

Lambda Expressions (2b)

- Expression or statement body
- Implicitly or explicitly typed parameters
- Examples:

```
x => x + 1 // Implicitly typed, expression body
x => { return x + 1; } // Implicitly typed, statement body
(int x) => x + 1 // Explicitly typed, expression body
(int x) =>
(x, y) =>
() => Cons
```

Lambda expressions in C# **always** exist only as **closures**,
and **never** as “**lambda functions**” with free variables
(open bindings)!

- A lambda converted implicitly

```
delegate R Func<A,R>(A arg);
Func<int,int> f1 = x => x + 1; // Ok
Func<int,double> f2 = x => x + 1; // Ok
Func<double,int> f3 = x => x + 1; // Error – double cannot be
// implicitly converted to int
```

Lambda Expressions (3)

- Lambda expressions participate in inference process of type arguments of generic methods
- In initial phase, nothing is inferred from arguments that are lambda expressions
- Following the initial phase, additional inferences are made from lambda expressions using an iterative process

Lambda Expressions (4)

- Generic extension method example:

```
public class List<T> : IEnumerable<T>, ... { ... }
    public static class Sequence {
        public static IEnumerable<S> Select<T,S>(
            this IEnumerable<T> source,
            Func<T, S> selector)
        {
            foreach (T element in source) yield return selector(element);
        }
    }
```

- Calling extension method with lambda expression:

```
List<Customer> customers = GetCustomerList();
IEnumerable<string> names = customers.Select(c => c.Name);
```

- Rewriting extension method call:

```
IEnumerable<string> names = Sequence.Select<T, S>(customers, c => c.Name);
```

- T type argument is inferred to Customer based on source argument type

```
Sequence.Select<Customer, S>(customers, c => c.Name)
```

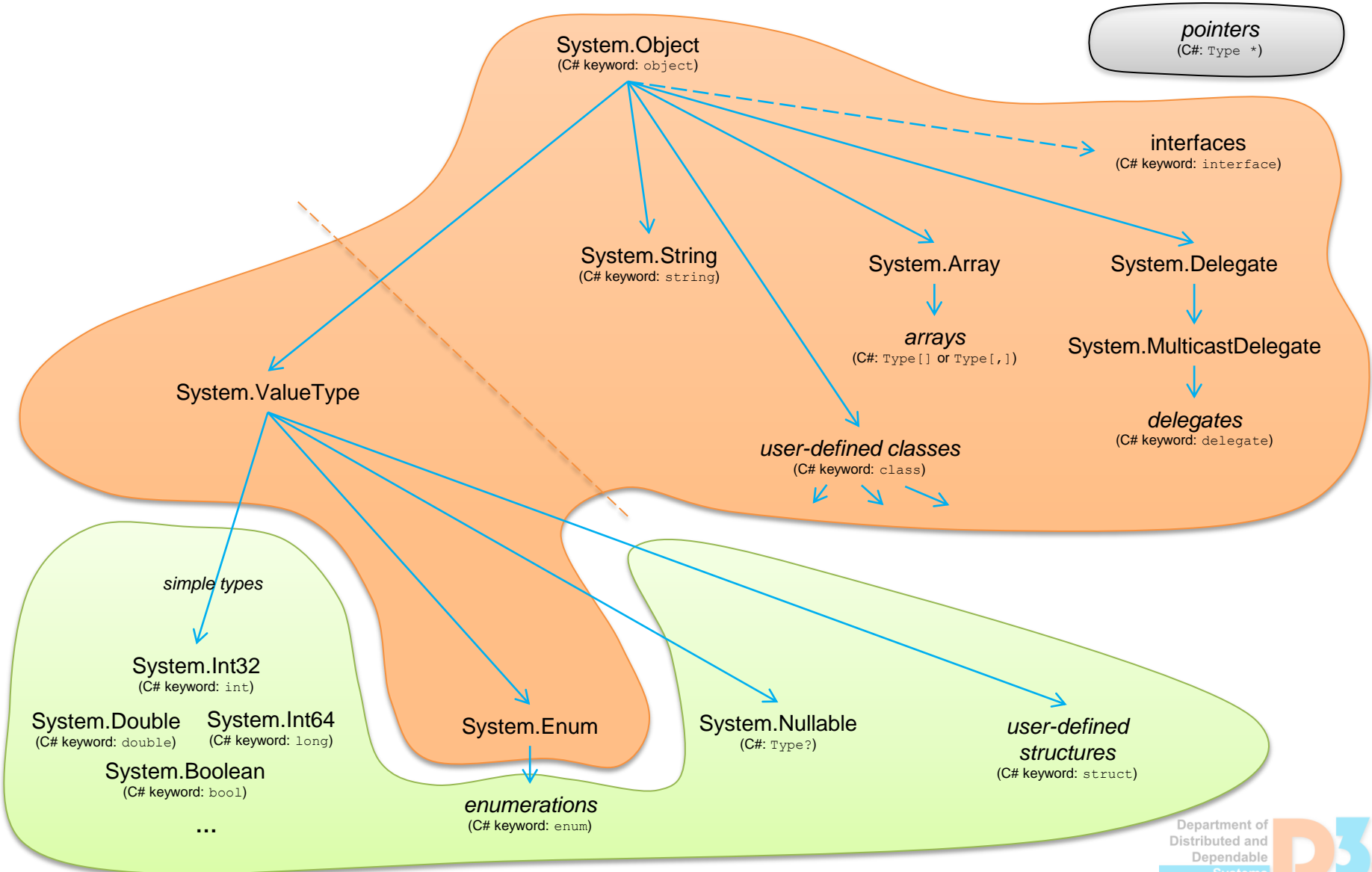
- c lambda expression argument type is inferred to Customer

```
Sequence.Select<Customer, S>(customers, (Customer c) => c.Name)
```

- S type argument is inferred to string based on return value type of the lambda expression

```
Sequence.Select<Customer, string>(customers, (Customer c) => c.Name)
```

CLI Type Inheritance



Nullable Types

```
int i = 123;
```

```
int? x;
```

```
x = i;
```

```
x = 456;
```

```
x = null;
```

```
if (x != null) {
```

```
    int j;
```

```
    j = x;    // ERROR
```

```
    j = (int) x; // OK
```

```
} else {
```

```
    int j = (int) x; // EXCEPTION – InvalidOperationException
```

```
}
```

Nullable Types

```
[Serializable]
public struct Nullable<T> where T : struct {
    public Nullable ( T value )

    public bool HasValue { get; }
    public T Value { get; }
    public T GetValueOrDefault ()
    public T GetValueOrDefault ( T defaultValue )

    ...
}
```

```
int i = 123;
int? x = 456;
int? y = null;
object o1 = i;           // o1 = reference to boxed int 123
object o2 = x;           // o2 = reference to boxed int 456
object o3 = y;           // o3 = null
int i1 = (int)o1;        // i1 = 123
int i2 = (int)o2;        // i2 = 456
int i3 = (int)o3;        // Error, System.NullReferenceException
int? ni1 = (int?)o1;    // ni1 = 123
int? ni2 = (int?)o2;    // ni2 = 456
int? ni3 = (int?)o3;    // ni3 = null

x == null    null == x    // !x.HasValue
x != null    null != x    // x.HasValue
int? z = x ?? y;        // x.HasValue ? x : y
int? u = x + y;         // (x.HasValue && y.HasValue) ? (x + y) : null
```

Nullable Types, bool?

x	Y	x & y	x y
true	true	true	true
true	false	false	true
true	null	null	true
false	true	false	true
false	false	false	false
false	null	false	null
null	true	null	true
null	false	false	null
null	null	null	null