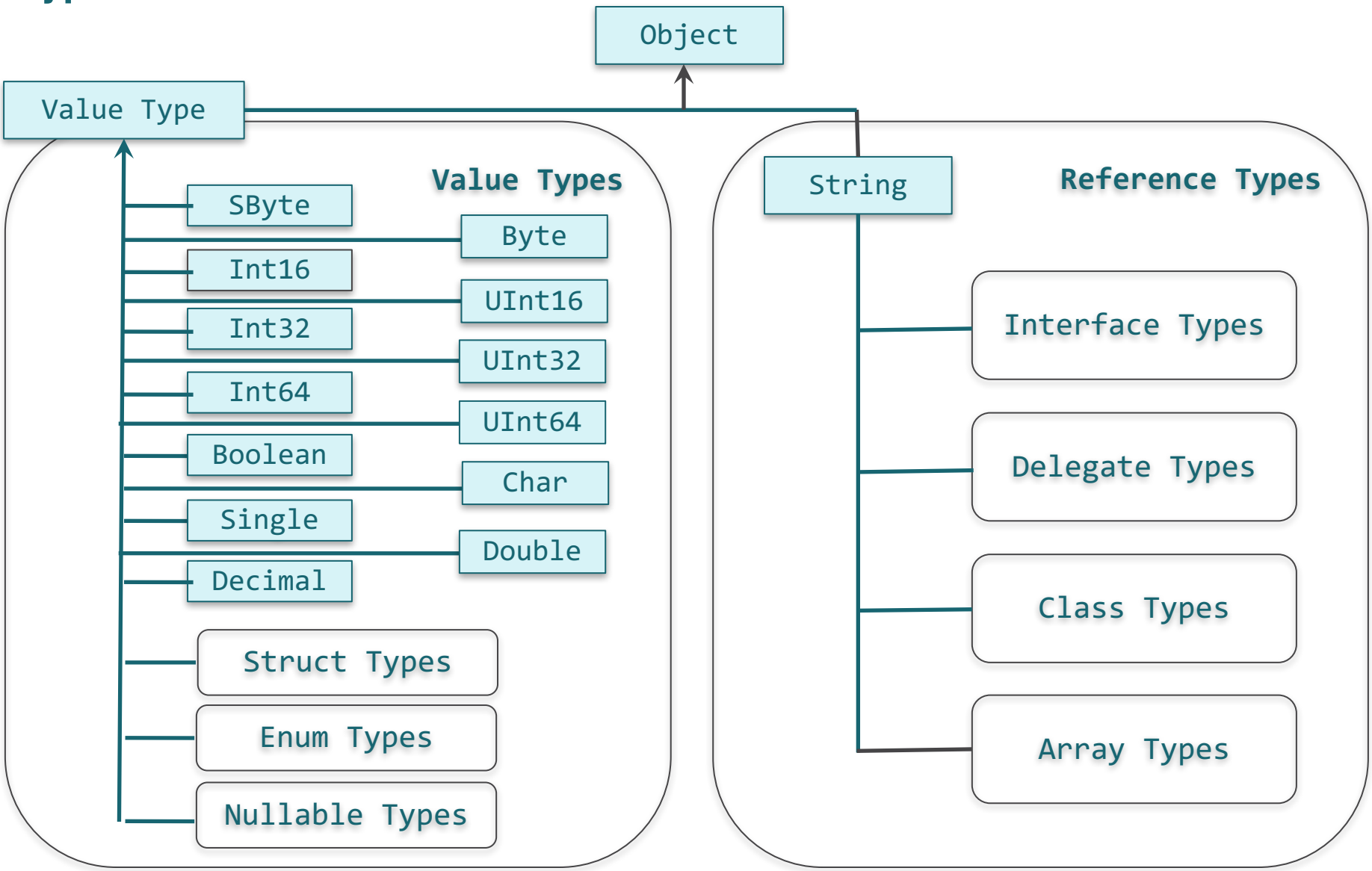

epam
CREATING TYPES IN C#
.NET & JS LAB MINSK
Anzhelika KRAVCHUK

Types


Object
Value Type
Value Types
SByte
Byte
Int16
UInt16
Int32
UInt32
Int64
UInt64
Boolean
Char
Single
Double
Decimal
Struct Types
Enum Types
Nullable Types
String
Reference Types
Interface Types
Delegate Types
Class Types
Array Types

Class

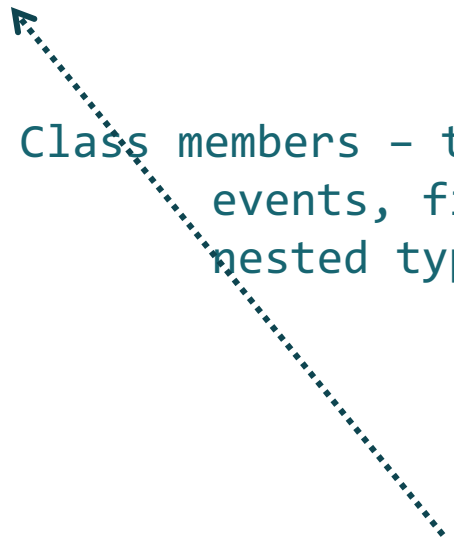
A class can contain both code and data, and it can choose to make some of its features publicly available, while keeping other features accessible only to code within the class. So classes offer a mechanism for encapsulation—they can define a clear public programming interface for other people to use, while keeping in ternal implementation details inaccessible.

```
class House
{
    ...
}
```

Class definitions always contain the class keyword followed by the name of the class

Class

```
[Attributes]
[Class modifiers] class ClassName [Generic type parameters, a base
                                   class, and interfaces]
{
        Class members – these are methods, properties, indexers,
                events, fields, constructors, overloaded operators,
                nested types, and a finalizer

}
```



public, internal, abstract, sealed, static, unsafe, partial

Class members. Fields

A field is a variable that is a member of a class or struct

Static modifier	static
Access modifier	public internal private protected
Inheritance modifier	new
Unsafe code modifier	unsafe
Read-only access modifier	readonly
The modifier of multithreading	volatile

Class members. Methods

A method is a procedure or function inside a class

Static modifier	static
Access modifier	public internal private protected
Inheritance modifier	new virtual abstract override sealed
Unmanaged code modifier	unsafe extern
Partial method modifier	partial
Asynchronous code modifier	async

Class members. Constructors

Static modifier	static
Access modifier	public internal private protected
Unmanaged code modifier	unsafe extern

Class members. Instance Constructors

Constructors run initialization code on a class or struct. A constructor is defined like a method, except that the method name and return type are reduced to the name of the enclosing type

```
public class Residence
{
    public Residence(ResidenceType type, int numberOfBedrooms)
    {
    }
    public Residence(ResidenceType type, int numberOfBedrooms,
        bool hasGarage)
    {
    }
    public Residence(ResidenceType type, int numberOfBedrooms,
        bool hasGarage, bool hasGarden)
    {
    }
}
```

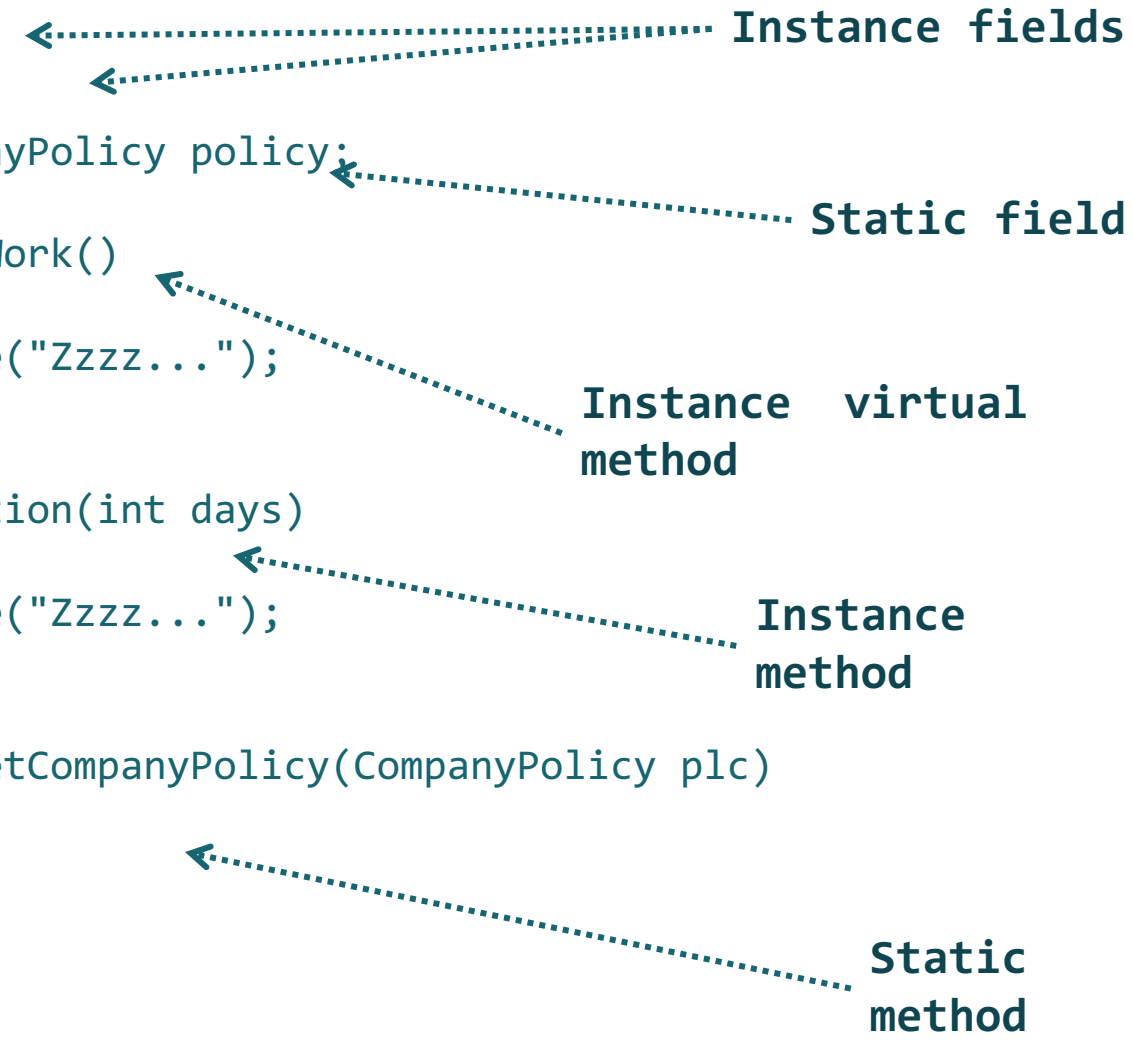
CLR calls the constructors automatically 

Class members. Instance Constructors

```
public class Residence
{
    private ResidenceType type;
    private int numberOfBedrooms;
    private bool hasGarage;
    private bool hasGarden;
    private Residence(ResidenceType type, int numberOfBedrooms, bool
hasGarage, bool hasGarden)
    {
        this.type = type;
        this.numberOfBedrooms = numberOfBedrooms;
        this.hasGarage = hasGarage;
        this.hasGarden = hasGarden;
    }
    public Residence() : this(ResidenceType.House, 3, true, true{ }
    ...
}
```

Objects Creating

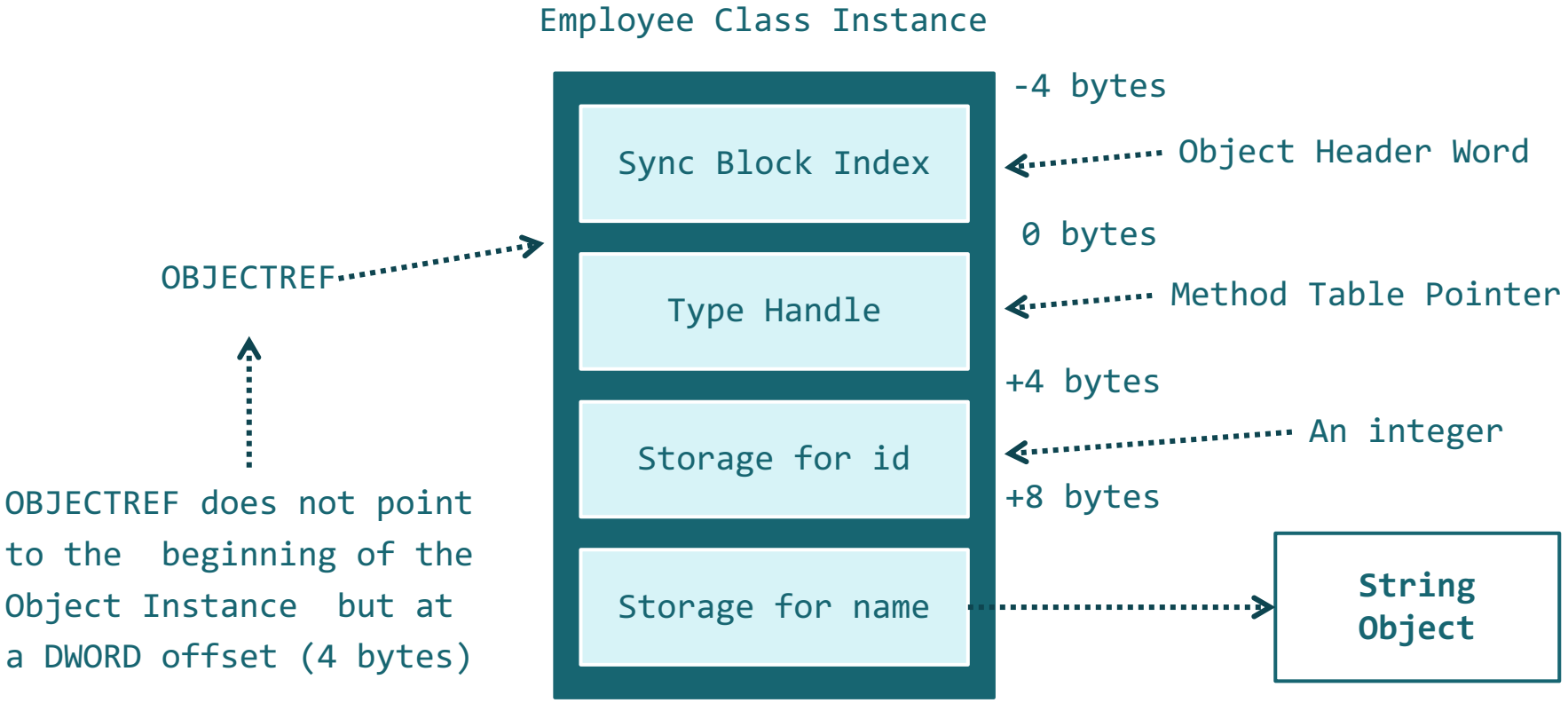
```
public class Employee
{
    private int id;
    private string name;
    private static CompanyPolicy policy;

    public virtual void Work()
    {
        Console.WriteLine("Zzzz...");
    }

    public void TakeVacation(int days)
    {
        Console.WriteLine("Zzzz...");
    }

    public static void SetCompanyPolicy(CompanyPolicy plc)
    {
        policy = plc;
    }
}
```



Objects Creating

- EE allocates memory for the object
- The EE initializes the pointer to the method table
- EE lays the pointer to the object in the ecx register and passes control to the constructor specified in the newobj instruction that generated the object creation code
- If no unhandled exceptions occurred during the constructor's operation, the reference to the object is placed in one or another scope variable, from which the object creation code was called

Objects Creating



Partial Types and Methods

Partial types allow a type definition to be split—typically across multiple files. A common scenario is for a partial class to be auto-generated from some other source (such as a Visual Studio template or designer) and for that class to be augmented with additional hand-authored methods

```
// PaymentFormGen.cs - auto-generated
partial class PaymentForm { ... }

 // PaymentForm.cs - hand-authored
partial class PaymentForm { ... }
```

```
// PaymentFormGen.cs - auto-generated
partial class PaymentForm { ... }

 // PaymentForm.cs - hand-authored
class PaymentForm { ... }
```

Partial types are resolved entirely by the compiler, which means that each participant must be available at compile time and must reside in the same assembly.

Partial Types and Methods

A partial type may contain partial methods. These let an auto-generated partial type provide customizable hooks for manual authoring

```
partial class PaymentForm // In auto-generated file
{ ...
        partial void ValidatePayment (ref decimal amount);
}
```

definition

```
partial class PaymentForm // In hand-authored file
{ ...
        partial void ValidatePayment (ref decimal amount)
        {
                if (amount > 100) ...
        }
}
```

implementation

Implicitly private

Struct

A struct is similar to a class, with the following key differences:

- A struct is a value type, whereas a class is a reference type.
- A struct does not support inheritance (other than implicitly deriving from object, or more precisely, System.ValueType).

A struct can have all the members a class can, except the following:

- A parameterless constructor
- Field initializers
- A finalizer
- Virtual or protected members

Because a struct is a value type, each instance does not require instantiation of an object on the heap; this incurs a useful savings when creating many instances of a type. For instance, creating an array of value type requires only a single heap allocation.

Definition and use of structure

```
struct Currency
{
    public string currencyCode;   // The ISO 4217 currency code
    public string currencySymbol; // The currency symbol ($,£,...)
    public int fractionDigits;    // The number of decimal places
}
```

```
Currency unitedStatesCurrency;
unitedStatesCurrency.currencyCode = "USD";
unitedStatesCurrency.currencySymbol = "$";
unitedStatesCurrency.fractionDigits = 2;
```

To create an instance of a struct, it is not necessary to use the operator new, but the sruct in this case is considered to be uninitialized

Enum

```
d = 5;
```

```
d = DayOfWeek.Friday;
```

Day

- Monday
- Tuesday
- Wednesday
- Thursday
- Friday
- Saturday
- Sunday

Code is easier to maintain, because only expected values of variables are determined

Code is easier to read, because easily identifiable names are assigned

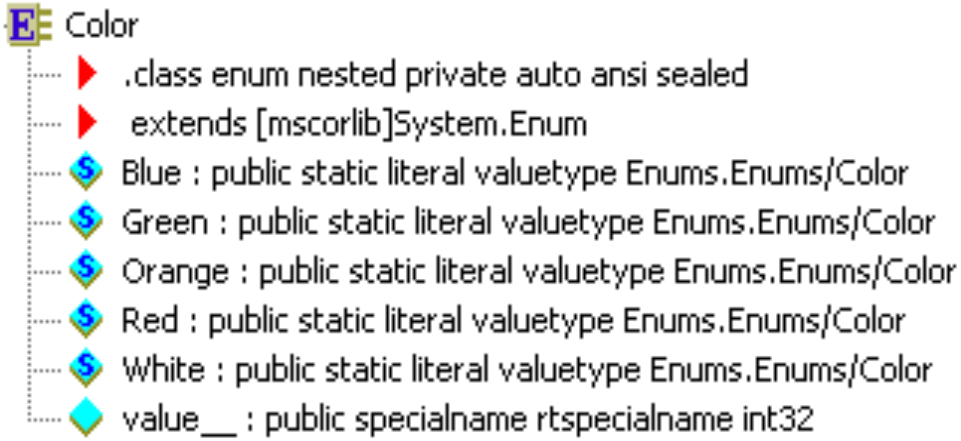
Code is easier to type, because IntelliSense displays a list of possible values that you can use

Enumerated types undergo strict type checking

Enum

```
public enum Color
{
    White,
    Red,
    Green,
    Blue,
    Orange
}
```

```
//psevdocode
public struct Color : System.Enum
{
    public const Color White = (Color) 0;
    public const Color Red = (Color) 1;
    public const Color Green = (Color) 2;
    public const Color Blue= (Color) 3;
    public const Color Orange = (Color) 4;

    public Int32 value__;
}
```



Enum

```
enum Season
{
    Spring,
    Summer,
    Autumn,
    Winter
};
```

```
enum Season
{
    Spring = 1,
    Summer,
    Autumn,
    Winter
};
```

```
enum Season
{
    Spring = 1,
    Summer,
    Autumn = 3,
    Fall = 3,
    Winter
};
```

```
enum Season : short
{
    Spring,
    Summer,
    Autumn,
    Winter
};
```

byte sbyte short ushort int uint long ulong

Base class FCL (Int32)

Enum

```
enum Day
{
    Monday = 1,
    Tuesday = 2,
    Wednesday = 3,
    Thursday = 4,
    Friday = 5,
    Saturday = 6,
    Sunday = 7
};

static void Main(string[] args)
{
    Day dayOff = Day.Sunday;
}
```

```
[EnumType] variableName = [EnumValue]
```

Enum

```
for(Day dayOfWeek = Day.Monday; dayOfWeek <= Day.Sunday; dayOfWeek++)
{
    Console.WriteLine(dayOfWeek);
}
```

```
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday
```

```
= == != < > <= >= + - ^ & | ~ += -= ++ -- sizeof
```

```
 Day.Monday + Day.Wednesday
```

```
[Flags]
public enum BorderSides { None=0, Left=1, Right=2, Top=4, Bottom=8}
```

Boxing and unboxing

```
Residence house = new Residence();
object obj = house;
```
class

```
Currency currency = new Currency();
object o = currency;
```
struct
boxing

```
Currency anotherCurrency = (Currency)o;
```
unboxing

Internally:

- Memory is allocated from the managed heap. The amount of memory allocated is the size required by the value type's fields plus the two additional overhead members (the type object pointer and the sync block index) required by all objects on the managed heap.
- The value type's fields are copied to the newly allocated heap memory.
- The address of the object is returned. This address is now a reference to an object; the value type is now a reference type.

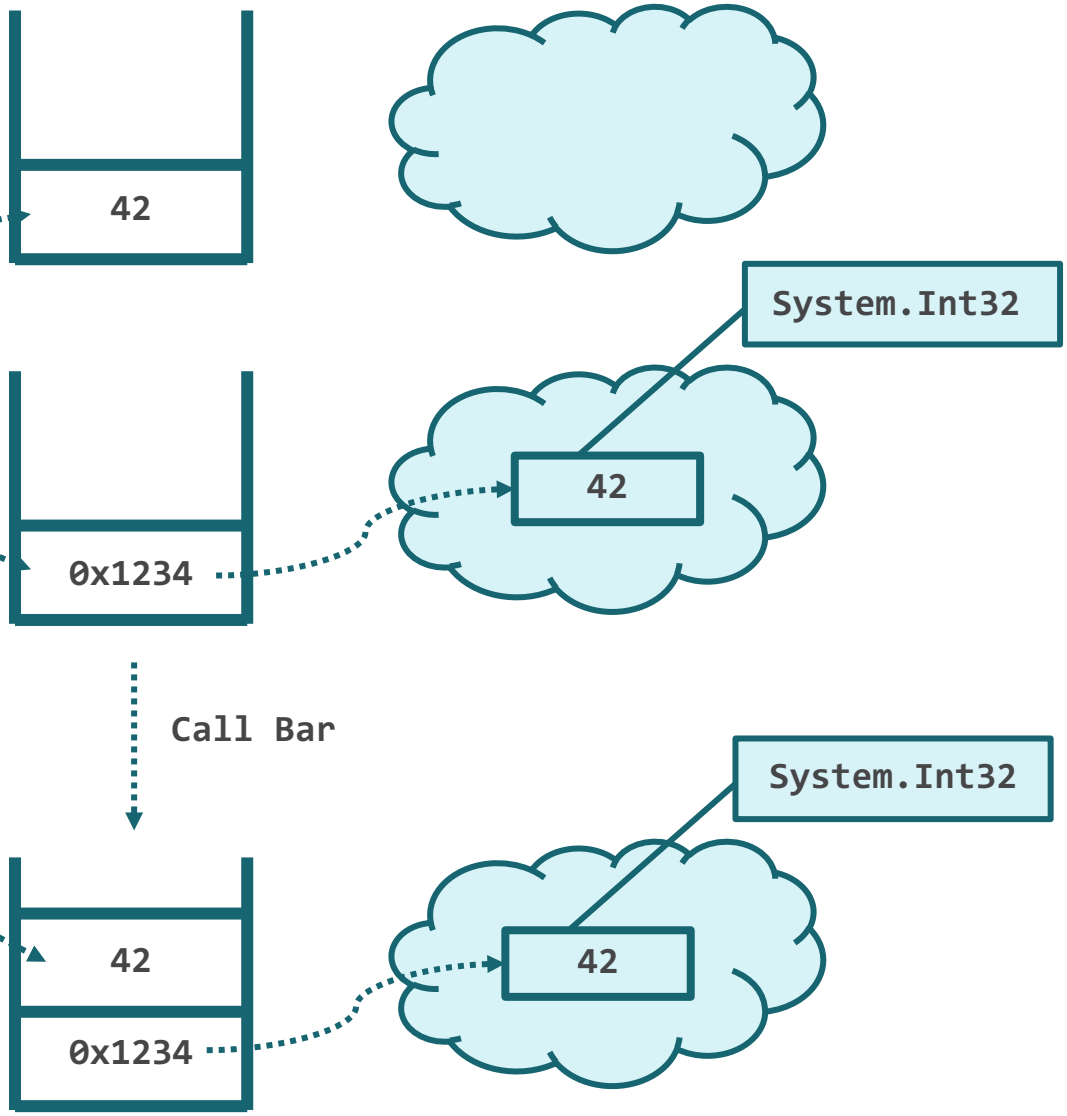
Boxing and unboxing

```
static void Main()
{
        Bar(42);
}
static void Bar(object value)
{
        int a = (int)value;
}
```

```
IL_0000:  nop
IL_0001:  ldc.i4.s    2A
IL_0003:  box         System.Int32
IL_0008:  call        UserQuery.Bar
IL_000D:  nop
IL_000E:  ret

Bar:
IL_0000:  nop
IL_0001:  ldarg.0
IL_0002:  unbox.any   System.Int32
IL_0007:  stloc.0     // a
IL_0008:  ret
```

Boxing and unboxing

```
IL_0000:  nop
IL_0001:  ldc.i4.s   2A
IL_0003:  box        System.Int32
IL_0008:  call       UserQuery.Bar
IL_000D:  nop
IL_000E:  ret

Bar:
IL_0000:  nop
IL_0001:  ldarg.0
IL_0002:  unbox.any  System.Int32
IL_0007:  stloc.0    // a
IL_0008:  ret
```



Nullable Types

Reference types can represent a nonexistent value with a null reference. Value types, however, cannot ordinarily represent null values.

```
Residence house = null;
...
if (house == null)
{
    house = new Residence(...);
}
```

```
Currency currency = null;
```


```
Currency? currency = null;
...
if (currency == null)
{
     currency = new Currency(...);
}
```

Nullable Types

To represent null in a value type, you must use a special construct called a nullable type. A nullable type is denoted with a value type followed by the ? symbol

T? translates into System.Nullable<T>, which is a lightweight immutable structure, having only two fields, to represent Value and HasValue

Nullable<Int32>	-2 147 483 648 … 2 147 483 647 or null
Nullable<bool>	true, false or null

```
Currency? currency = null;
...
if (currency.HasValue)
{
    Console.WriteLine(currency.Value);
}
```

Nullable Types

```
int? i = null;

int j = 99;

i = 100
```

```
i = j;
```

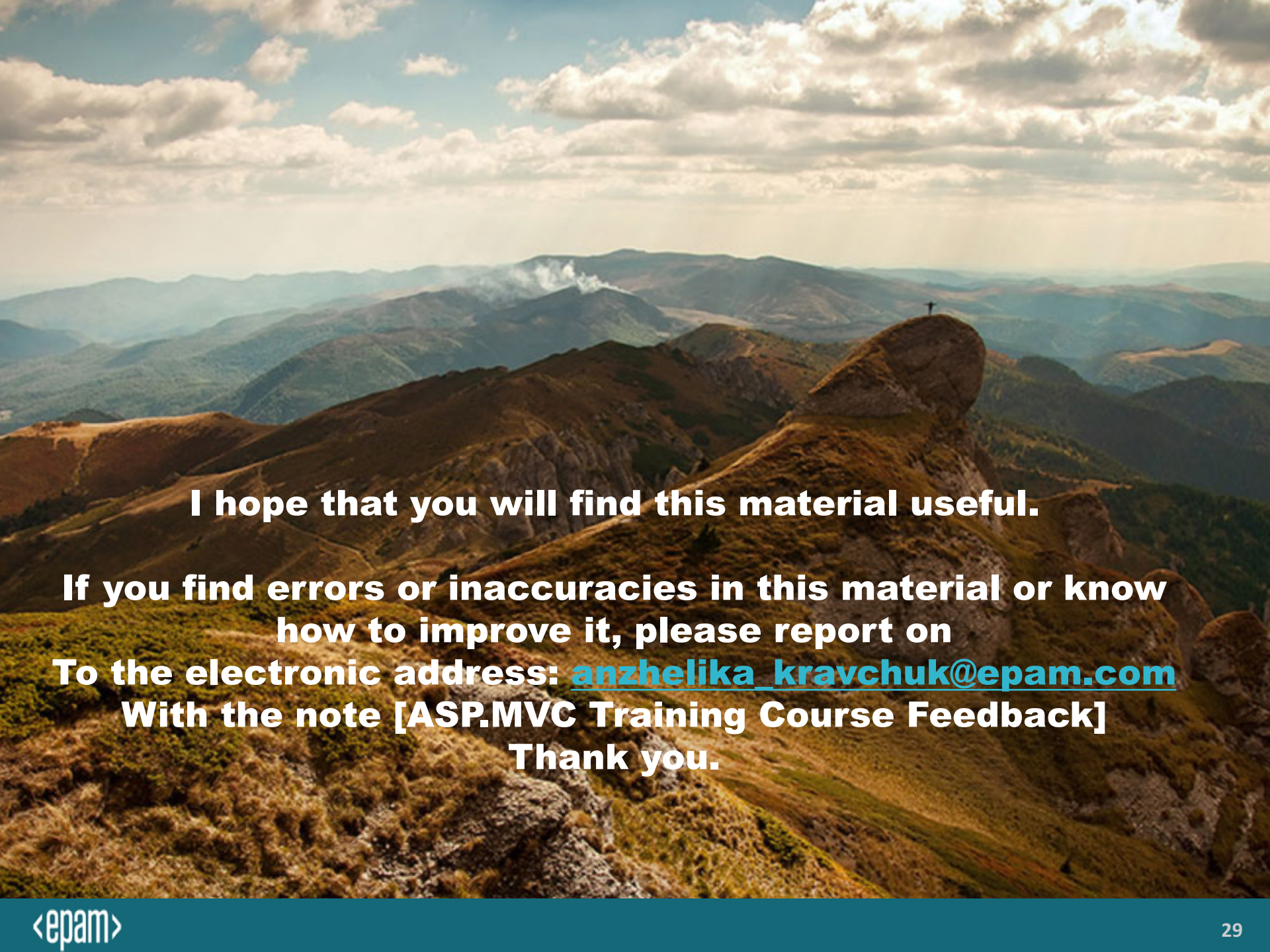
```
j = i;
```

```
int x = (b.HasValue) ? b.Value : 123;
int x = b ?? 123;
```

```
string temp = GetFilename();
filename = (temp != null) ? temp : "Untitled";
string filename = GetFilename() ?? "Untitled”;
```

```
System.Text.StringBuilder sb = null;
string s = sb?.ToString();
// No error; s instead evaluates to null
```


Thank you for attention!


I hope that you will find this material useful.
If you find errors or inaccuracies in this material or know how to improve it, please report on
To the electronic address: anzhelika_kravchuk@epam.com
With the note [ASP.MVC Training Course Feedback]
Thank you.