02
Advanced Aspects of .NET

Giovanni Ciatto
giovanni.ciatto@unibo.it

Dipartimento di Informatica – Scienza e Ingegneria
Alma Mater Studiorum—Università di Bologna, Cesena

a.a. 2022/2023

Outline

Structures as Custom Value Types I

Why structures

- Differently from most platforms, .NET let developers define custom values types
- It does so via the notion of structures
- Built-in value types (e.g. `Int32`, `Boolean`, etc) are defined as structures

Structures as Custom Value Types II

Structures vs. Classes

Think about structures as ordinary classes, except that:

- they do not support inheritance
  - from neither other structures nor classes
- they can just implement interfaces
- their instances are allocated on the stack, by default
- their instances are passed by value through method calls

Purpose of Structures

Defining lightweight types whose (de)allocation is quick

Structures as Custom Value Types III

Syntax of Structures

`struct` ⟨*Name*⟩ `[:` ⟨*Interface Name*⟩`]` `{` ⟨*Members*⟩ `}`

- where ⟨*Members*⟩ is an ordinary list of
  - fields, methods, constructors, or properties...
  - ...either static or not...
  - ...similarly to class definitions

Structures are sub-types of `Object`! (pt. 1)

! keep in mind to override `Equals`, `GetHashCode`, and `ToString`

Structures as Custom Value Types IV

Example of structure for fractions (a.k.a. rational numbers)

```
struct Rational
{
    public Rational(bool sign, uint num, uint den)
    {
        if (den == 0) throw new DivideByZeroException("Denominator cannot be 0");
        Sign = sign; Num = num; Den = den;
    }

    public bool Sign { get; }
    public uint Num { get; }
    public uint Den { get; }

    public bool Equals(Rational other) =>
        Sign == other.Sign && Num == other.Num && Den == other.Den;
    public override bool Equals(object obj) => obj is Rational other && Equals(other);
    public override int GetHashCode() => HashCode.Combine(Sign, Num, Den);

    public override string ToString() => $"{(Sign ? "+" : "-")}{Num}/{Den}";
}
```

```
Rational oneHalf = new Rational(true, 1, 2);
Console.WriteLine(oneHalf); // +1/2
var minusTwoThird = new Rational(false, 2, 3);
Console.WriteLine(minusTwoThird); // -2/3
```

Next In Line...

Reference vs. Value types Allocation I

Take away

- Reference-type (i.e. classes) objects are allocated on the heap
- Value-type (i.e. structures) objects are allocated on the stack
  - this is what makes them quick, provided that the are small in size

! Such difference brings subtle intricacies in the way reference/value are managed

Reference vs. Value types Allocation II

Consider for instance the `IPoint` interface, which can either

- be implemented by the `CPoint` class
- or be implemented by the `SPoint` class

```
interface IPoint { double X { get; set; } double Y { get; set; } }

class CPoint : IPoint
{
    public CPoint(double x, double y) { X = x; Y = y; }
    public double X { get; set; }
    public double Y { get; set; }

    public override string ToString() => $"CPoint(X: {X}, Y: {Y})";
}

struct SPoint : IPoint
{
    public SPoint(double x, double y) { X = x; Y = y; }
    public double X { get; set; }
    public double Y { get; set; }

    public override string ToString() => $"SPoint(X: {X}, Y: {Y})";
}
```

Reference vs. Value types Allocation III

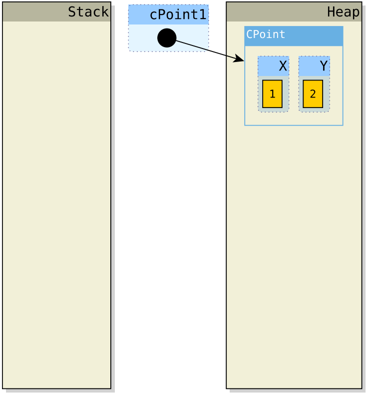
Their instances behave differently, due to the way they are allocated:

```
CPoint cPoint1 = new CPoint(1, 2);
SPoint sPoint1 = new SPoint(3, 4);

CPoint cPoint2 = cPoint1;
SPoint sPoint2 = sPoint1;

cPoint1.X = 5; cPoint1.Y = 6;
sPoint1.X = 7; sPoint1.Y = 8;

Console.WriteLine(cPoint1); // CPoint(X: 5, Y: 6)
Console.WriteLine(sPoint1); // SPoint(X: 7, Y: 8)
Console.WriteLine(cPoint2); // CPoint(X: 5, Y: 6)
Console.WriteLine(sPoint2); // SPoint(X: 3, Y: 4)
```

Reference vs. Value types Allocation IV

Explanation:

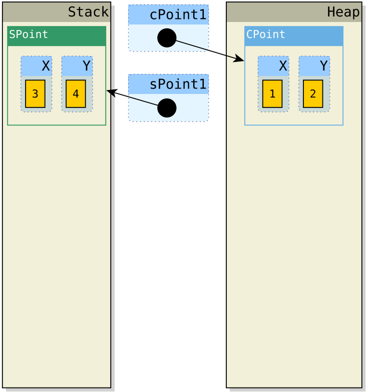
```
CPoint cPoint1 = new CPoint(1, 2);
```



Reference vs. Value types Allocation V

Explanation:

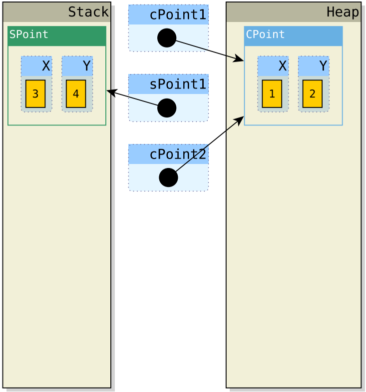
```
SPoint sPoint1 = new SPoint(3, 4);
```



Reference vs. Value types Allocation VI

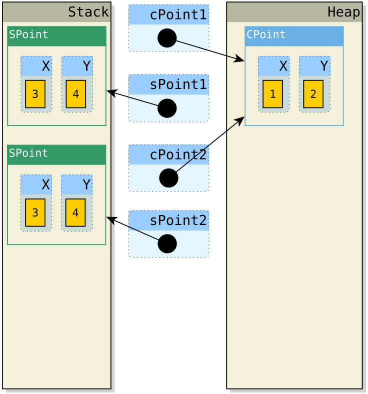
Explanation:

```
CPoint cPoint2 = cPoint1;
```



Explanation:

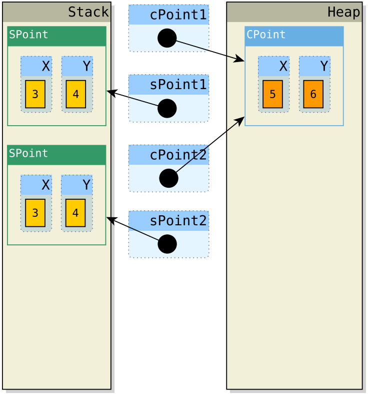
```
SPoint sPoint2 = sPoint1;
```



Reference vs. Value types Allocation VIII

Explanation:

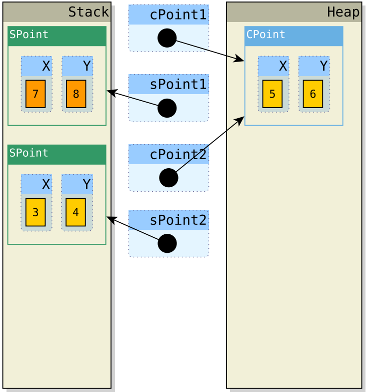
```
cPoint1.X = 5; cPoint1.Y = 6;
```



Reference vs. Value types Allocation IX

Explanation:

```
sPoint1.X = 7; sPoint1.Y = 8;
```



Next In Line...

Value Types – Boxing & Unboxing I

Problem

- All structures are value types
- All structures are syb-types of `Object`, and, possibly, of some interface
- `Object` is a reference type, as well as all interfaces
- How can the Liskov substitution principle hold?

Value Types – Boxing & Unboxing II

Boxing and unboxing

Boxing: value type $\xrightarrow{\textit{assign}}$ reference type

- when a value-type object is assigned to a reference-type variable, the object is copied to the heap

Unboxing: reference type $\xrightarrow{\textit{cast}}$ value type

- when a boxed object is casted back to a value type, the object is copied to the stack

Keep in mind

- Boxing and unboxing are slow and should be minimised
- While unboxing is commonly explicit, boxing is often implicit
  - pay attention to the code you write to avoid unexpected boxing

Value Types – Boxing & Unboxing III

Consider again the CPoint-SPoint example:

```
CPoint cPoint1 = new CPoint(1, 2);
SPoint sPoint1 = new SPoint(3, 4);

CPoint cPoint2 = cPoint1;
SPoint sPoint2 = sPoint1;

cPoint1.X = 5; cPoint1.Y = 6;
sPoint1.X = 7; sPoint1.Y = 8;
```

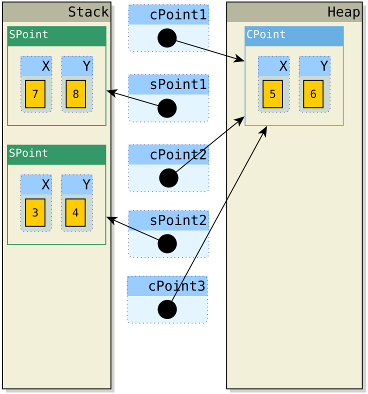
consider now the following additional code:

```
IPoint cPoint3 = cPoint2;
IPoint sPoint3 = sPoint2; // BOXING

cPoint2.X = 9; cPoint2.Y = 0;
sPoint2.X = 1; sPoint2.Y = 2;

Console.WriteLine(cPoint2); // CPoint(X: 9, Y: 0)
Console.WriteLine(sPoint2); // SPoint(X: 1, Y: 2)
Console.WriteLine(cPoint3); // CPoint(X: 9, Y: 0)
Console.WriteLine(sPoint3); // SPoint(X: 3, Y: 4)
```

Value Types – Boxing & Unboxing IV

Explanation:

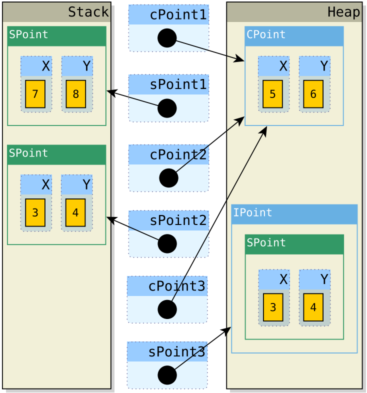
```
IPoint cPoint3 = cPoint2;
```



Value Types – Boxing & Unboxing V

Explanation:

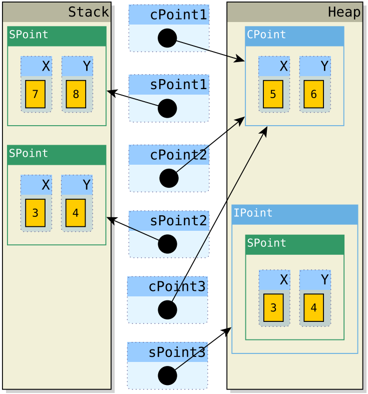
```
IPoint sPoint3 = sPoint2; // BOXING
```



Value Types – Boxing & Unboxing VI

Explanation:

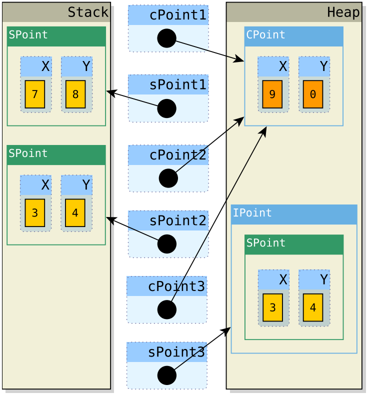
```
IPoint sPoint3 = sPoint2; // BOXING
```



Value Types – Boxing & Unboxing VII

Explanation:

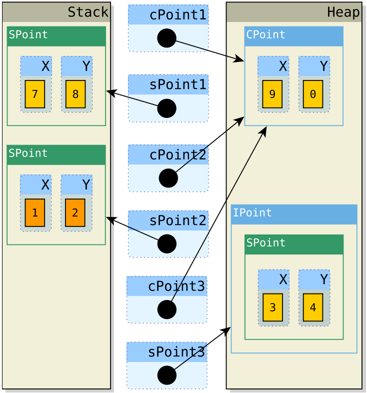
```
cPoint2.X = 9; cPoint2.Y = 0;
```



Value Types – Boxing & Unboxing VIII

Explanation:

```
sPoint2.X = 1; sPoint2.Y = 2;
```



Value Types – Boxing & Unboxing IX

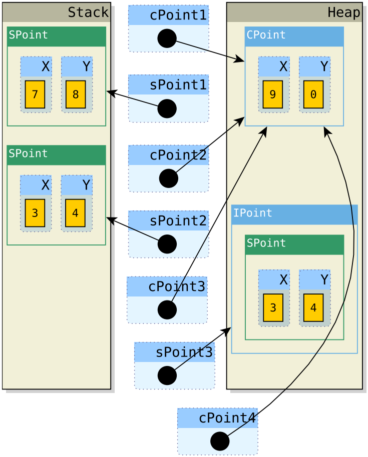
Unboxing is somewhat dual, despite it requires explicit cast:

```
CPoint cPoint4 = (CPoint)cPoint3;
SPoint sPoint4 = (SPoint)sPoint3; // UNBOXING

cPoint3.X = 3; cPoint3.Y = 4;
sPoint3.X = 5; sPoint3.Y = 6;

Console.WriteLine(cPoint3); // CPoint(X: 3, Y: 4)
Console.WriteLine(sPoint3); // SPoint(X: 5, Y: 6)
Console.WriteLine(cPoint4); // CPoint(X: 3, Y: 4)
Console.WriteLine(sPoint4); // SPoint(X: 3, Y: 4)
```

Value Types – Boxing & Unboxing X

Explanation:

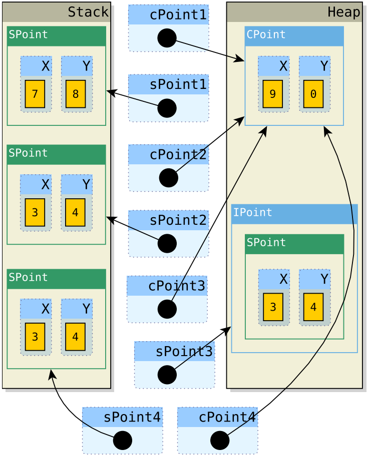
```
CPoint cPoint4 = (CPoint)cPoint3;
```



Value Types – Boxing & Unboxing XI

Explanation:

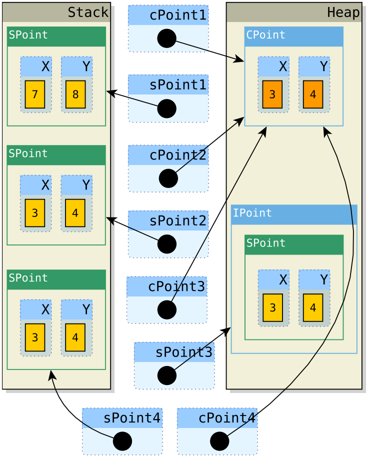
```
SPoint sPoint4 = (SPoint)sPoint3; // UNBOXING
```



Value Types – Boxing & Unboxing XII

Explanation:

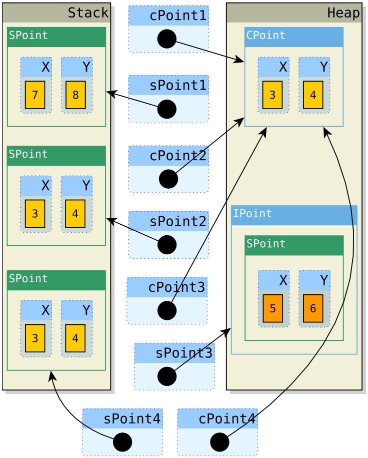
```
cPoint3.X = 3; cPoint3.Y = 4;
```



Value Types – Boxing & Unboxing XIII

Explanation:

```
sPoint3.X = 5; sPoint3.Y = 6;
```



Outline

Overview

- Parameters are passed among method calls in different ways
- Reference (resp. value) types are passed by-reference (resp. by value)
- In general, method calls cannot affect outer variables
  - ! they can alter objects referred by such variables, not their references
    - ie they can only provoke side effects
- Parameters marked as either `ref` or `out` can affect outer scopes
  - ie they can change what outer variables are referencing
- Parameters marked as `params` can occurr 0, 1, or more times
  - these are commonly called "varargs" in other languages
  - → such a mechanism lets a method accept a variable amount of arguments
- Parameters marked by `this` let a static method be called as an instance method

Next In Line...

Method Calls with Value/Reference-Type Parameters I

Ordinary Method Calls – General Rules

- When value-type object are passed, objects are cloned
- When reference-type object are passed, only references are cloned
- Altering parameters in methods bodies does not affect outer variables
- The use case is to support C-like input-output parameters

Method Calls with Value/Reference-Type Parameters II

Consider for instance the following methods:

```csharp
static void Inc(SomeReferenceType arg) =>
    arg.SomeProperty += 1; // side effect on a reference type
static void Inc(SomeValueType arg) =>
    arg.SomeProperty += 1; // USELESS side effect on a value type

static void Replace(SomeReferenceType arg) =>
    arg = new SomeReferenceType(arg.SomeProperty + 1); // re-assigning a local parameter
static void Replace(SomeValueType arg) =>
    arg = new SomeValueType(arg.SomeProperty + 1); // re-assigning a local parameter
```

Method Calls with Value/Reference-Type Parameters III

```
SomeReferenceType value1 = new SomeReferenceType(1);
SomeValueType value2 = new SomeValueType(2);

Console.WriteLine(value1); // SomeReferenceType(1)
Console.WriteLine(value2); // SomeValueType(2)

Inc(value1); // attempts to increase a reference to value1
Inc(value2); // attempts to increase a clone of value1 (leaving value1 unaffected)

Console.WriteLine(value1); // SomeReferenceType(2)
Console.WriteLine(value2); // SomeValueType(2)

// notice that the objects referenced by value1 and value2 are always the same

Replace(value1); // has no effect
Replace(value2); // has no effect

Console.WriteLine(value1); // SomeReferenceType(2)
Console.WriteLine(value2); // SomeValueType(2)

// notice that the objects referenced by value1 and value2 are still the same
```

- method `Inc(SomeValueType)` is useless, since it always alters a local copy of `SomeValueType`
- method `Inc(SomeReferenceType)` works as expected
- both methods `Replace` are meaningless, as the modification they perform on their arguments are not propagated outside

Next In Line...

Method Calls with `ref` Parameters I

`ref` Parameters – General Rules

- Formal `ref` parameters are marked as `ref` in method signatures
- Actual `ref` parameters are marked as `ref` in method calls
- `ref` parameters are always passed by reference, even if `value` types
- Re-assigning a `ref` parameter implies re-assigning some the outer variable which has been passed upon method invocation

Method Calls with `ref` Parameters II

Consider for instance the following methods:

```
static void IncRef(ref SomeReferenceType arg) =>
    arg.SomeProperty += 1; // side effect on a reference type
static void IncRef(ref SomeValueType arg) =>
    arg.SomeProperty += 1; // side effect on a value type (affects outer references too)

static void ReplaceRef(ref SomeReferenceType arg) =>
    arg = new SomeReferenceType(arg.SomeProperty + 1); // re-assigns outer variables too
static void ReplaceRef(ref SomeValueType arg) =>
    arg = new SomeValueType(arg.SomeProperty + 1); // re-assigns outer variables too
```

Method Calls with `ref` Parameters III

```csharp
SomeReferenceType value1 = new SomeReferenceType(1);
SomeValueType value2 = new SomeValueType(2);

Console.WriteLine(value1); // SomeReferenceType(1)
Console.WriteLine(value2); // SomeValueType(2)

IncRef(ref value1); // attempts to increase a reference to value1
IncRef(ref value2); // attempts to increase a reference to value2

Console.WriteLine(value1); // SomeReferenceType(2)
Console.WriteLine(value2); // SomeValueType(3)

// notice that the objects referenced by value1 and value2 are still the same

ReplaceRef(ref value1); // attempts to replace a reference to value1
ReplaceRef(ref value2); // attempts to replace a reference to value2

Console.WriteLine(value1); // SomeReferenceType(3)
Console.WriteLine(value2); // SomeValueType(4)

// notice that the objects referenced by value1 and value2 are DIFFERENT now
```

- outer variables are affected by methods manipulations!

Method Calls with `out` Parameters I

`out` Parameters – General Rules

- `out` parameters are like `ref` parameters...
- ...except that `out` parameters must be assigned before before methods return
- and that `ref` arguments must be initialised before being passed
  - otherwise a compilation error is generated
- The use case is to support C-like output parameters

Method Calls with `out` Parameters II

Consider for instance the following method:

```
static class Utils
{
    public static bool TryFindIndex<T>(IEnumerable<T> list, T item, out uint index)
    {
        index = 0; // remove this line and the method won't compile
        foreach (var x in list)
        {
            if (x.Equals(item)) return true;
            index++;
        }
        return false;
    }
}
```

- it attempts to look for the index of an item in an enumerable
  - returning `true` if the item is found
  - or `false` otherwise
- in case the item is found, its index is stored into the output parameters

Method Calls with `out` Parameters III

Usage example:

```
var list = new List<string>() { "a", "d", "c", "b" };
uint indexOfC;
if (Utils.TryFindIndex(list, "c", out indexOfC))
{
    Console.WriteLine(indexOfC); // 2
}
else
{
    Console.WriteLine("not found"); // NOT PRINTED
}
```

- Takeaway: they serve primarily to provide additional outputs beyond the return value

Next In Line...

Method Calls with Variable Parameters I

`param` Parameters – General Rules

- Parameters marked as `param` in method signatures. . .
- . . . can be provided in arbitrary amounts upong method calls
  eg 0, 1, or more
- They are treated as arrays within methods
- They are treated as ordinary arguments outside methods
- The use case is to support a variable amount of arguments as input

Method Calls with Variable Parameters II

Consider for instance the following method:

```
static class CollectionUtils
{
    public static ISet<T> SetOf<T>(T first, params T[] others)
    {
        var set = new HashSet<T>();
        set.Add(first);
        foreach (var item in others) set.Add(item);
        return set;
    }
}
```

- this is a generic method aimed at creating and filling an `ISet<T>`
- it accepts $1 + N$ parameters of type `T`
    - where N may be 0, 1, or more
- it is handy since `T` is automatically inferred in method calls
    - and must not explicitly provided by developers

Method Calls with Variable Parameters III

Usage example:

```
// 1 + 2 parameters, implicit type
var set1 = CollectionUtils.SetOf("a", "b", "c"); // type of set1 is ISet<string>
foreach (string str in set1) Console.WriteLine(str); // a, b, c

// 1 + 5 parameters, explicit type
var set2 = CollectionUtils.SetOf<int>(1, 2, 2, 3, 4, 4); // type of set2 is ISet<int>
foreach (int num in set2) Console.WriteLine(num); // 1, 2, 3, 4

// 1 + 0 parameters, implicit type
var set3 = CollectionUtils.SetOf(Complex.Zero); // type of set3 is ISet<Complex>
foreach (Complex c in set3) Console.WriteLine(c); // 0
```

Next In Line...

Extension Methods I

Extension Methods – General Rules

- Method defined in non-generic, non-nested **static classes** can be marked as extensions
- The **first argument** of an extension method is marked by `this`
- Extension methods may work as ordinary static methods...
- ...but they can also be called as if they were instance methods
  - ! instance methods of the type of the argument marked by `this`
- Their use case is to add functionalities to a pre-existing type
  - whose definition cannot or should not be extended/altered
    - eg interfaces, sealed classes, structures, enums, etc.
  - without requiring any edit to the type definiton

Extension Methods II

Consider for instance the following method:

```
public static string ToAlternateCase(this string input)
{
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < input.Length; i++)
    {
        var currentChar = "" + input[i];
        sb.Append(i % 2 == 0 ? currentChar.ToUpper() : currentChar.ToLower());
    }
    return sb.ToString();
}
```

- it aims at converting a string into `AlTeRnAtE CaSe`
- it exploits a `StringBuilder`
  - ie an object aimed at creating a string incrementally
- notice the first argument is of type `string` and it is marked by `this`
  - meaning that this is an extension method, extending the `string` type
    - → the method can be invoked on strings as an instance method:

```
Console.WriteLine("Hello World!".ToAlternateCase()); // HeLlO WoRlD!
Console.WriteLine(ToAlternateCase("Hello World!")); // HeLlO WoRlD!
```

Extension Methods III

Generic Extension Method

- Common practice: combining extension and generic methods. . .
- . . . to add functionalities to a wide range of type at once

Extension Methods IV

Consider for instance the following method:

```
public static string ToString<T>(this IEnumerable<T> items, string delimiter,
    string prefix, string suffix)
{
    StringBuilder sb = new StringBuilder(prefix);
    var e = items.GetEnumerator();
    if (e.MoveNext()) sb.Append(e.Current.ToString());
    while (e.MoveNext())
    {
        sb.Append(delimiter);
        sb.Append(e.Current.ToString());
    }
    sb.Append(suffix);
    return sb.ToString();
}
```

- it converts any enumerable of any type `T` into a string
- where the items of the enumerable are representes as strings, separated by `delimiter`
- and the whole string is wrapped between `prefix` and `suffix`

Extension Methods V

Usage Example

```
IEnumerable<string> list = new List<string>() {"a", "b", "c"};
Console.WriteLine(list.ToString(", ", "[", "]")); // [a, b, c]

IEnumerable<int> enumerable = Enumerable.Range(1, 5);
Console.WriteLine(enumerable.ToString("; ", "(", ")")); // (1; 2; 3; 4; 5)
```

Extension Methods VI

Name clashing in Extension Methods

- Of course an extension method may have the same name of some actual instance method of a type

! When this is the case, actual instace methods take priority over extension methods

Extension Methods VII

Consider for instance the following method:

```
public static string ToUpper(this string input) =>
    throw new ArgumentException("Error.");
```

- it is an extension methods for strings
- notice the String class has an instance method named ToUpper

$\rightarrow$ in case of ambiguity, the original method of String is invoked

One can reveal this rule as follows:

```
Console.WriteLine("Hello World!".ToUpper()); // HELLO WORLD!
Console.WriteLine(ToUpper("Hello World!")); // System.ArgumentException: Error.
```

- 1^{st} invocation is ambigous, then the original ToUpper method is called
- 2^{nd} one is not, then the extension method is invoked
    - which provokes an exception!

Outline

Enums I

About enums

- Enums are fixed-size groups of related constants
  - eg days of week, months of the year, seasons, gender
- OOP languages usually represents groups of related constants as types
  - . . . having a fixed amount of instances
- Such types are called enum types, and they have an ad-hoc syntax
- In .NET enums must be sub-types of some built-in integer type
  - ie `(U)Int16/32/64`, or `(S)Byte`
  - $\rightarrow$ .NET enums are **value-types**
  - $\rightarrow$ .NET enums are **integers**

Enums II

Syntax

`enum` ⟨*Name*⟩ [: ⟨*Integer Type*⟩] { ⟨*Constants*⟩ }

- where ⟨*Name*⟩ is the name of the enum type being defined
- and ⟨*Integer Type*⟩ is one of `(U)Int16/32/64`, or `(S)Byte`
  - defaults to `Int32` in case it is missing
- and ⟨*Constants*⟩ is a number of comma-separated symbols in `PascalCase`
  - optionally assigned to their values

Enums III

Example of enum

```
enum SingleDayOfWeek : byte // defaults to int if nothing is specified
{
    Monday,     // defaults to 0
    Tuesday,    // defaults to 1
    Wednesday,  // defaults to 2
    Thursday,   // defaults to 3
    Friday,     // defaults to 4
    Saturday,   // defaults to 5
    Sunday      // defaults to 6
}
```

Usage of enum

```
SingleDayOfWeek first = SingleDayOfWeek.Monday;
Console.WriteLine(first); // Monday
Console.WriteLine((byte)first); // 0
SingleDayOfWeek second = (SingleDayOfWeek)1;
Console.WriteLine(second == SingleDayOfWeek.Tuesday); // true
Console.WriteLine(second > SingleDayOfWeek.Sunday); // false
Console.WriteLine(second + 1); // Wednesday
```

- notice enums are essentially integers

Flag Enums I

Common practice: flag enums

- If an enum values are less than 64. . .
- . . . and you may need to group enum values together
- then you may consider implementing you enum as a flag
- Flag enums are enums whose values correspond to powers of 2
  - bitwise operators can then be exploited to speed-up or simplify some tasks

Flag Enums II

Consider again the `SingleDayOfWeek`:

```
enum SingleDayOfWeek : byte // defaults to int if nothing is specified
{
    Monday,      // defaults to 0
    Tuesday,     // defaults to 1
    Wednesday,   // defaults to 2
    Thursday,    // defaults to 3
    Friday,      // defaults to 4
    Saturday,    // defaults to 5
    Sunday       // defaults to 6
}
```

- this is not a flag enum (values are not powers of 2)

Flag Enums III

Imagine you need a means to determine if a day is:

1. part of the weekend or not
2. even or odd (knowing that Sunday is not considered as even nor as odd)

```
public static bool IsWeekend(SingleDayOfWeek day) =>
    day > SingleDayOfWeek.Friday;

public static bool IsOdd(SingleDayOfWeek day) =>
    (int)day % 2 == 0 && day != SingleDayOfWeek.Sunday;
```

- these methods must take the indexing of values into account

Flag Enums IV

Alternatively, one may model the days of week as a flag enum:

```
[Flags] // notice this attribute!
enum DaysOfWeek : byte
{
    None = 0,
    Monday = 1,
    Tuesday = 2,
    Wednesday = 4,
    Thursday = 16,
    Friday = 32,
    Saturday = 64,
    Sunday = 128,

    WorkingDays = Monday | Thursday | Wednesday | Thursday | Friday,
    Weekend = Saturday | Sunday,

    EvenDays = Tuesday | Thursday | Saturday,
    OddDays = Monday | Wednesday | Friday
}
```

- groups of days can be modelled as well

Flag Enums V

Why do flag enums need power of two?

- Each position in an integer correspond to the presence/lack of a value
- Sets of values can be represented as by integers

$$
\begin{array}{rcl}
\texttt{Monday} = & 1 & = 00000001 \\
\texttt{Wednesday} = & 4 & = 00000100 \\
\texttt{Friday} = & 16 & = 00010000 \\
\hline
\texttt{None} = & 0 & = 00000000 \\
\texttt{OddDays} = & 21 & = 00010101
\end{array}
$$

So, the aforementioned methods can be conveniently implemented:

```
public static bool IsWeekend(DaysOfWeek day) =>
    (day & DaysOfWeek.Weekend) != DaysOfWeek.None;

public static bool IsOdd(DaysOfWeek day) =>
    (day & DaysOfWeek.OddDays) != DaysOfWeek.None;
```

The Enum class I

About the Enum class

- It is the super-type of all enums
- It comes with a number of useful static methods aimed at:
  - Enumerate the values of a given enum
  - Parse the values of a given enum from string
  - Get the names of the values of a given array
- cf. `https://docs.microsoft.com/dotnet/api/system.enum`

The Enum class II

Example 1 – Enumerating values and getting names

```csharp
foreach (SingleDayOfWeek day in Enum.GetValues(typeof(SingleDayOfWeek)))
{
    bool valueIsWeekend = IsWeekend(day);
    string name = Enum.GetName(typeof(SingleDayOfWeek), day);
    Console.WriteLine(name + ", weekend: " + valueIsWeekend);
}
```

```
Monday weekend: False
Tuesday weekend: False
Wednesday weekend: False
Thursday weekend: False
Friday weekend: False
Saturday weekend: True
Sunday weekend: True
```

The Enum class III

Example 2 – Parsing values

```
foreach (var name in new string[] { "monday", "tuesday", "wednesday", "thursday",
    "friday", "saturday", "sunday" })
{
    DaysOfWeek day = Enum.Parse<DaysOfWeek>(name, /* ignore case: */ true);
    bool valueIsWeekend = IsOdd(day);
    Console.WriteLine(day + ", odd: " + valueIsWeekend);
}
```

```
Monday, odd: True
Tuesday, odd: False
Wednesday, odd: True
Thursday, odd: False
Friday, odd: True
Saturday, odd: False
Sunday, odd: False
```

Outline

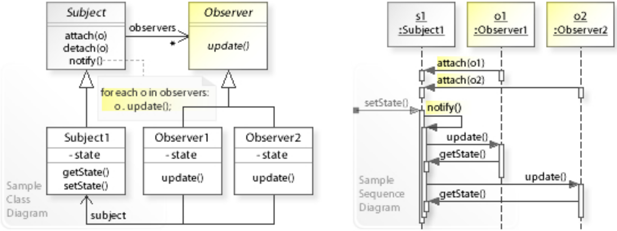
The Observer Pattern I

Observer pattern[1] (a.k.a. event-listener, a.k.a. publish-subscribe):

- lets an entity react to relevant events concerning another entity
- two entities involved:
  - **subject** is the entity whose events are of interest
    - **aka** observable, publisher, or event source
  - **observer** is the entity willing to react to events
    - **aka** listener, or subscriber
- involves two phases:
  1. the observer registers its interest to the subject
  2. the subject notifies all the registered observers whenever an event occurs

The Observer Pattern II

- in OOP, event notification are commonly reified into method calls
- when reified into OOP code, the subject needs 3 methods
    1. a method to let subjects register
    2. a method to let subjects unregister
    3. a (possibly private) method to propagate events to observers
- when reified into OOP code, the observer needs 1 method
    - specifying what to do whenever a new event notifaction is
- in a nutshell:



[1] cf. https://en.wikipedia.org/wiki/Observer_pattern

.NET Events I

About .NET events

- .NET provides built-in support to the observer pattern via events
- Events are yet another sort of **member** in .NET classes/interfaces
  - The **event source/listener** nomenclature is adopted in .NET
  - Classes corresponding to event sources may expose a number of named events
  - Each event defines the type of the listener which may register to it
- Event listeners are instances of some delegate type (i.e. functions)

.NET Events II

Syntax – Definition Side

$$\texttt{event}\ \langle \textit{Delegate Type} \rangle\ \langle \textit{Event Name} \rangle\texttt{;}$$

- where ⟨*Event Name*⟩ is the name of the event, `PascalCase`
- and ⟨*Delegate Type*⟩ is some delegate denoting the possible type of listeners for ⟨*Event Name*⟩
  - most commonly `Action<T>`

.NET Events III

Syntax – Usage Side (Listener Registration)

$$\langle \mathit{Object} \rangle . \langle \mathit{Event\ Name} \rangle \mathrel{+=} \langle \mathit{Event\ Listener} \rangle ;$$

or

$$\langle \mathit{Object} \rangle . \langle \mathit{Event\ Name} \rangle \mathrel{-=} \langle \mathit{Event\ Listener} \rangle ;$$

- to (un)register an $\langle \mathit{Event\ Listener} \rangle$ for $\langle \mathit{Event\ Name} \rangle$
- assuming $\langle \mathit{Object} \rangle$ defines an event named $\langle \mathit{Event\ Listener} \rangle$
- and $\langle \mathit{Event\ Listener} \rangle$ matches the type of $\langle \mathit{Event\ Name} \rangle$

.NET Events IV

Syntax – Usage Side (Event Propagation)

$$\langle \mathit{Object} \rangle\texttt{.}\langle \mathit{Event\ Name} \rangle\texttt{?.Invoke(}\langle \mathit{Args} \rangle\texttt{);}$$

or

$$\texttt{if (}\langle \mathit{Object} \rangle\texttt{.}\langle \mathit{Event\ Name} \rangle\texttt{ != null) \{}$$
$$\langle \mathit{Object} \rangle\texttt{.}\langle \mathit{Event\ Name} \rangle\texttt{(}\langle \mathit{Args} \rangle\texttt{) \}}$$

- where the amounts and types of $\langle \mathit{Args} \rangle$ depend on the type of $\langle \mathit{Event\ Name} \rangle$

.NET Events – Example I

An interface exposing an event

```
interface IButton
{
    string Purpose { get; }

    void Press();

    // event name: OnPressed
    // type of listeners: any method accepting a string and returning void
    event Action<string> OnPressed;
}
```

- instances of `IButton` are buttons having a particular purpose
  - eg the name of the button (Esc, Enter, Tab, etc.)
- buttons can be pressed via the `Press()` method
- whenver the button is pressed, the `OnPressed` event is propagated to listeners
  - and the purpose of the event is passed to each listener

.NET Events – Example II

A class triggering an event

```
class Button : IButton
{
    public Button(string purpose)
    {
        Purpose = purpose;
    }

    public string Purpose { get; }

    public void Press()
    {
        if (OnPressed != null) OnPressed(Purpose); // propagates to ALL listeners
        // or simply: OnPressed?.Invoke(Purpose);
    }

    public event Action<string> OnPressed;
}
```

.NET Events – Example III

Usage of events

```
static void OnButtonPressed(string purpose) =>
    Console.WriteLine($"{purpose} has been pressed, caught by method");

static void Main(string[] args)
{
    Action<string> listener = purpose => {
        Console.WriteLine($"{purpose} has been pressed, caught by lambda");
    };
    IButton esc = new Button("Esc");
    esc.OnPressed += listener; // adding a listener: reference to lambda
    IButton enter = new Button("Enter");
    enter.OnPressed += OnButtonPressed; // adding a listener: reference to method

    esc.Press(); // Esc has been pressed, caught by lambda
    enter.Press(); // Enter has been pressed, caught by method
}
```

Outline

Inconsistencies in C# Operators

Consider the == operator. . .

- It compares refernce types by reference (i.e. checks for identity)
- It compares strings by value
  - ! notice that strings are reference types!
- It compares integers by value

⋮

Consider the += operator. . .

- It compares increases numbers
- It concatenates strings
- It adds listeners to events

⋮

! How are all such inconsistencies possible?

Operators Overloading I

Definition

Operator overloading is a feature letting OOP languages redefine the semantics of some operators on a per-type basis

In .NET

- .NET supports operator overloading on classes, via static methods
  - since version 8, operator overloading is supported for interfaces too
- only a predefined set of operators can be overloadaded
  - eg `+`, `-`, `*`, `/`, `==`, `!=`, `>`, `<`, etc
  - priority and associativity of operators cannot be altered
- classes/interfaces are not constrained to overload all operators
- explicit/implicit casts may be defined as well, via operator overloading
- some built-in classes overload some operators
  - eg `String` overloads at least `+`, `==`, and `!=`

Operators Overloading II

Syntax – Unary Operator

public static $\langle T_2 \rangle$ operator $\langle Symbol \rangle$($\langle T_1 \rangle$ $\langle N_1 \rangle$) { $\langle Code \rangle$ }

- represents a unary operator producing an object of type $\langle T_2 \rangle$
- out an object of type $\langle T_1 \rangle$
- which can be used with prefix syntax, via $\langle Symbol \rangle$

  eg +, -, !, etc.

! commonly, $\langle T_1 \rangle$ and $\langle T_2 \rangle$ are equal to the hosting type

Operators Overloading III

Syntax – Binary Operator

`public static` $\langle T_3 \rangle$ `operator` $\langle \mathit{Symbol} \rangle$`(`$\langle T_1 \rangle$ $\langle N_1 \rangle$`,` $\langle T_2 \rangle$ $\langle N_2 \rangle$`)`
`{` $\langle \mathit{Code} \rangle$ `}`

- represents a binary operator producing an object of type $\langle T_3 \rangle$
- out of two objects of types $\langle T_1 \rangle$ and $\langle T_2 \rangle$
- which can be used with infix syntax, via $\langle \mathit{Symbol} \rangle$
  - eg `+`, `-`, `*`, `/`, `==`, `!=`, `>`, `<`, etc

! commonly, $\langle T_1 \rangle$ and $\langle T_2 \rangle$ are equal to the hosting type

Operators Overloading IV

Syntax – Cast Operator

`public static` ⟨*Usage*⟩ `operator` ⟨T_2⟩ `(`⟨T_1⟩ ⟨N_1⟩`)` `{` ⟨*Code*⟩ `}`

- where ⟨*Usage*⟩ is either `implicit` or `explicit`
- The notation above creates an implicit/explicit casting operator
- converting an object of type ⟨T_1⟩ into an object of type ⟨T_2⟩

! commonly, ⟨T_2⟩ (resp. ⟨T_1⟩) is equal to the hosting type for implicit (resp. explicit) operators
  - usually other types are implictly casted to the hosting type
  - usually the hosting type is excpliticly casted to other type

Operators Overloading – Example I

Complex Numers with Operators

```
public class Complex
{
    public static readonly Complex I = new Complex(0, 1);

    public static Complex Polar(double modulus, double phase) =>
        new Complex(modulus * Math.Cos(phase), modulus * Math.Sin(phase));

    public Complex(double real, double imaginary) { Real = real; Imaginary = imaginary; }


    public double Real { get; }
    public double Imaginary { get; }
    public double Modulus => Math.Sqrt(Real * Real + Imaginary * Imaginary);
    public double Phase => Math.Atan2(Imaginary, Real);


    public override string ToString() => $"{Real} + {Imaginary}*i";

    public override int GetHashCode() => HashCode.Combine(Real, Imaginary);

    public override bool Equals(object obj)
    {
        var other = obj as Complex;
        return !ReferenceEquals(other, null)
               && Real.Equals(other.Real)
               && Imaginary.Equals(other.Imaginary);
    }
```

Operators Overloading – Example II

```csharp

    public static Complex operator -(Complex c) => new Complex(-c.Real, -c.Imaginary);
    public static Complex operator +(Complex c1, Complex c2) =>
        new Complex(c1.Real + c2.Real, c1.Imaginary + c2.Imaginary);
    public static Complex operator -(Complex c1, Complex c2) => c1 + (-c2);
    public static Complex operator *(Complex c1, Complex c2) =>
        Polar(c1.Modulus * c2.Modulus, c1.Phase + c2.Phase);
    public static Complex operator /(Complex c1, Complex c2) =>
        Polar(c1.Modulus / c2.Modulus, c1.Phase - c2.Phase);

    public static bool operator ==(Complex c1, Complex c2) => c1.Equals(c2);
    public static bool operator !=(Complex c1, Complex c2) => !(c1 == c2);

    public static implicit operator Complex(double x) => new Complex(x, 0);
    public static explicit operator double(Complex c) =>
        c.Imaginary == 0.0 ? c.Real : throw new InvalidCastException("Not a real: " + c);
}
```

Operators Overloading – Example III

Notice that:

- 1 unary operator (i.e. `-`), negating both components of a `Complex`
- 4 binary operators (i.e. `+`, `-`, `/`, `*`) are defined to accept and return `Complexes`
  - either working on real/imaginary components or on modulus and phase
  - ! notice that binary minus is defined in terms of other operators
- 2 comparison operators (i.e. `==`, `!=`) are defined in terms of `Complex.Equals`
- implicit casts from `double` to `Complex` are allowed
  - or from anything that can be implicitly casted to double, e.g. `int`
- explicit casts from `Complex` to `double` are allowed
  - but only work if the imaginary part is 0

Operators Overloading – Example IV

Usage of Complex Numers with Operators

```
int one = 1;
Complex c = one + Complex.I; // implicit cast from int to double and then to Complex
Console.WriteLine(c); // 1 + 1*i
c *= 2; // implicit cast from int to double and then to Complex, before multiplication
Console.WriteLine(c); // 2,0000000000000004 + 2*i
c = 1 / c; // "inverse" operator is somewhat implicitly defined
Console.WriteLine(c); // 0,25 + -0,24999999999999994*i
c += Complex.I * 0.25; // "multiply by scalar" is somewhat implicitly defined
Console.WriteLine(c); // 0,25 + 5,551115123125783E-17*i
c = (double) c; // InvalidCastException: Not a real: 0,25 + 5,551115123125783E-17*i
Console.WriteLine(c); // NOT EXECUTED
```

Operators Overloading – Remarks

Beware of Languages supporting Operator Overloading

- You never know what's the meaning of an operator until you read the doc
- Nobody constrains developers to implement meaningful operators
- Do now endow your types with operators unless their meaning is obvious!

Reference Comparison vs Value Comparison

- Operators == and `!=` test identity by default
- By they may be overloadaded to test for equality
- When this is the case, how can identity be tested?
- This is the purposed of the `Object.ReferenceEquals` static method

Outline

The Need for LINQ I

Consider the algorithm `GetTripledFirstNEvenNumbers` which

- accepts an enumerable of integers as input
- and returns an enumerable containing no more than N numbers...
- and these numbers are tripled w.r.t. the first N even numbers in the input enumerable

eg the algorithm applied to `[7, 6, 2, 9, 10, 4, 2, ...]`

- should return `[18, 6, 30, 12]`
- provided that $N = 4$

The Need for LINQ II

We may implement the algorithm as follows:

```
static IEnumerable<int> GetTripledFirstNEvenNumbers2(IEnumerable<int> items, int n)
{
    var list = new List<int>();
    foreach (var item in items)
    {
        if (item % 2 == 0)
        {
            list.Add(item * 3);
            n--;
        }
        if (n == 0) break;
    }
    return list;
}
```

- yet, this code steps through the unnecessary construction of an intermediate list
  - this may be inefficient, e.g. in case of large N

The Need for LINQ III

We may then use `yield` to make the algorithm totally lazy:

```
static IEnumerable<int> GetTripledFirstNEvenNumbers3(IEnumerable<int> items, int n)
{
    foreach (var item in items)
    {
        if (item % 2 == 0)
        {
            yield return item * 3;
            n--;
        }
        if (n == 0) yield break;
    }
}
```

- this is technically ok, but still very verbose
- you need to carefully read it to understand what's going on

Computational laziness

No computation is actually performed until the very last useful moment

The Need for LINQ IV

We may rewrite the same algorithm in functional style, to make it more declarative:

```
static IEnumerable<int> GetTripledFirstNEvenNumbers4(IEnumerable<int> items, int n) =>
    items.Where(item => item % 2 == 0)
        .Select(even => even * 3)
        .Take(n);
```

- laziness is retained
- the code is more consise and declarative
- "phases" of computation are made evident

! this is the essence of LINQ

The Need for LINQ V

We may also consider of re-writing the algorithm in SQL-like syntax:

```
static IEnumerable<int> GetTripledFirstNEvenNumbers5(IEnumerable<int> items, int n) =>
    (
        from item in items
        where item % 2 == 0
        select item * 3
    ).Take(n);
```

- this implies interpreting the input enumerable as an abstract database
- more practical, if you are confident with SQL

LINQ – Language-INtegrated Query I

What is LINQ

- A portion of the .NET framework
- Aimed at manipulating any sort of data-source which can be enumerated
  - ranging from in-memory collections, to remote databases, stepping through files
- Via a rich library of high-order functions
- and syntactical tricks aimed at making data manipulation very quick (to write)
  - eg an (optional) SQL-like syntax

LINQ – Language-INtegrated Query II

How does LINQ work

- Via a bunch of extension methods defined in `System.Linq.Enumerable`
- Allowing several sorts of operations on any sort of `IEnumerable<T>`
- Most notable sorts of operations:

  provisioning — a (possibly long/infinite) stream of data is lazily generated / read from some source

  transformation — an enumerable is transformed into another enumerable

  reduction — a value is computed out of an enumerable

- Operations are pipelined
  - each operation is lazy, and it performs as less computations as possible

LINQ – Language-INtegrated Query III

Example of provisioning operations

```
// Generates an infinite stream of values by calling a function over and over again
static IEnumerable<T> Generate<T>(Func<T> provider)
{
    while (true)
        provider();
}

// Generates a stream of integers ranging from min to max, incremented by delta at each
step
static IEnumerable<int> Range<T>(int min, int max, int delta)
{
    for (; min < max; min += delta)
        yield return min;
}
```

LINQ – Language-INtegrated Query V

Example of transformation operations

```
// Transforms the enumerable by applying a function to each item
static IEnumerable<R> Select<T, R>(this IEnumerable<T> items, Func<T, R> transform)
{
    foreach (var item in items)
        yield return transform(item);
}

// Filters out from the stream those items for which a predicate does not hold
static IEnumerable<T> Where<T>(this IEnumerable<T> items, Func<T, bool> filter)
{
    foreach (var item in items)
        if (filter(item))
            yield return item;
}

// Only takes the first n items in the input enumerable
static IEnumerable<T> Take<T>(this IEnumerable<T> items, int n)
{
    foreach (var item in items)
    {
        if (n > 0)
        {
            yield return item;
            n--;
        }
        else yield break;
    }
}
```

LINQ – Language-INtegrated Query VI

Example of reduction operations

```
// Gets the maximum value in a stream, given a comparer
static T Max<T>(this IEnumerable<T> items, Func<T, T, int> comparer) where T : class
{
    T max = null;
    foreach (var item in items)
        if (comparer(item, max) > 0)
            max = item;
    return max;
}

// Gets the minimum value in a stream, given a comparer
static T Min<T>(this IEnumerable<T> items, Func<T, T, int> comparer) where T : class =>
    items.Max((a, b) => -comparer(a, b));
```