Object-oriented programming in C#

Mirko Viroli, Giovanni Ciatto

C.D.L. Ingegneria e Scienze Informatiche
Alma Mater Studiorum—Università di Bologna, Cesena

a.a. 2022/2023

Outline

Brief introduction to C#

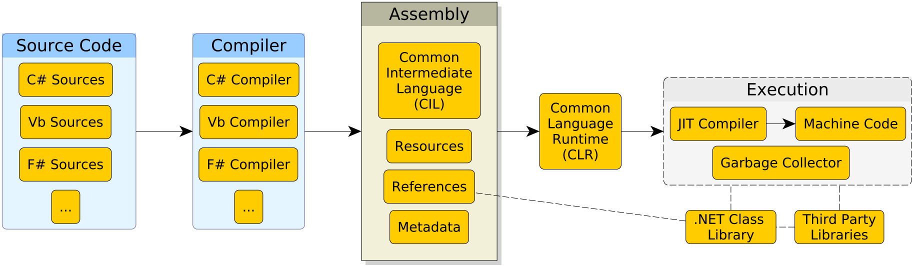
C# and .NET

- Designed by Anders Hejlsberg around 2000 at Microsoft
- Is part of the .NET initiative, designed to compile over the Common Language Infrastructure (CLI)
- Current version of C# is 11.0, released in 2022; current version of .NET is 7.0
- Mono is a free, open-source compiler and runtime environment
- C# Initially developed as very similar to Java, then somewhat diverged
- Essentially, C# took a different path than Java in following Scala
- Shall in these slides refer to "mainstream/standard OOP" to mean the intersection of Java/C#

.NET

Main elements

- .NET started as a polyglot framework since its beginning
- C# is by far the mostly used language
- Concepts replicate Java and JVM: CIL/bytecode, CLR/JVM, and so on
- As a key difference, .NET initially targeted only Microsoft Windows



.NET Platform – Present vs. Past

Past to Present

- Before .NET 5 there used to be three major implementations of the *class library*:

  .NET Framework — Windows-specific, full-featured, targetting desktop and web applications

  .NET Core — multi-platform (Win, Mac, Linux), less-featured, targetting desktop and web applications

  Xamarin — mobile-oriented (Android, iOS, Mac OS)

- Since .NET 5, implementations are aligned

In these slides

Stick to .NET 6

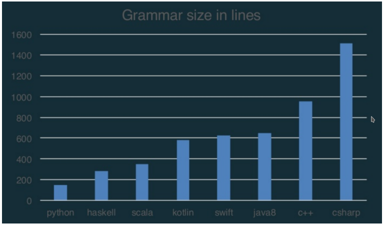
Features of C#

Ingredients

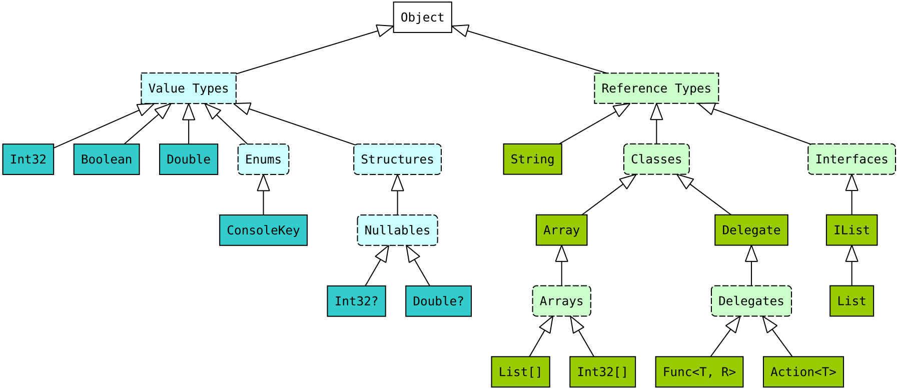
- C-like language: the imperative and structured parts are very similar
- Java-like language: essentially very similar to Java, specially at the beginning
- Static and strong typing: types are checked at run-time, preventing ill-typed operations
- Object-orientation: object by references, automatic garbage collection
- Functional-orientation: generics, delegates, lambdas

Philosophy

- aiming at high expressiveness and richness, though become a rather “large” language



C# types: we start with Simple Types and Class Types



Variables: initialisation and inference

On variables – essentially as in Java

- Same rules on scoping, and assignment
- Similar distinction between primitive and class types
- Similar naming conventions for variables
- `null` is assignable to variables of reference types
- Can use `var` to declare a variable with type to be inferred
- Keywords (`int`, `bool`, `string`, `object`) map to Library Value Types or Classes

```
String s = new String("aaa");
string s2 = "bbb";  // string and String are aliases
s2 = "ccc"; // reassignment
s2 = s; // s2 will contain a reference to the object of "aaa"

String ss; // Define name ss, without initialisation
// String s3 = ss; // this would not compile!
ss = "init"; // now assign ss

int i = 5 + 2; // int and Int32 are aliases
int j = i; // j and i both contain bits representing 7

Object o = null; // null is a special reference
object o2 = o; // object and Object are aliases

var x = 5; // by type inference, equivalent to int x = 5;
```

.NET Built-in Types

Name	Keyword	Category	Size	Description
`Boolean`	`bool`	*val*	1	either `true` or `false`
`Char`	`char`	*val*	2	UTF-16 characters '`U+0000`' ... '`U+FFFF`'
`Byte`	`byte`	*val*	1	integers in $0 \ldots (2^8 - 1)$
`SByte`	`sbyte`	*val*	1	integers in $-2^7 \ldots (2^7 - 1)$
`Int16`	`short`	*val*	2	integers in $-2^{15} \ldots (2^{15} - 1)$
`UInt16`	`ushort`	*val*	2	integers in $0 \ldots (2^{16} - 1)$
`Int32`	`int`	*val*	4	integers in $-2^{31} \ldots (2^{31} - 1)$
`UInt32`	`uint`	*val*	4	integers in $0 \ldots (2^{32} - 1)$
`Int64`	`long`	*val*	8	integers in $-2^{63} \ldots (2^{63} - 1)$
`UInt64`	`ulong`	*val*	8	integers in $0 \ldots (2^{64} - 1)$
`Float`	`float`	*val*	4	abs in $1.5 \times 10^{-45} \ldots 3.4 \times 10^{38}$
`Double`	`double`	*val*	8	abs in $5.0 \times 10^{-324} \ldots 1.7 \times 10^{308}$
`Decimal`	`decimal`	*val*	16	abs in $1.0 \times 10^{-28} \ldots 7.9228 \times 10^{28}$
`Object`	`object`	*ref*	$O(1)$	anything
`String`	`string`	*ref*	$O(n)$	sequences of n UTF-16 characters

(cf. `https://docs.microsoft.com/dotnet/csharp/language-reference/builtin-types/built-in-types`)

Outline

C# classes

The core of OOP is essentially as in Java

- Classes, methods, fields, and constructors have same syntax and semantics
- Class instantiation, method invocation, field access have same syntax and semantics
- Static, non-static fields/methods have same syntax/semantics
- Structured programming constructs (if/while) have same syntax/semantics
- A source file must define the namespace, similar to Java package but in a wrapping construct
- Syntax for calling a constructor from another constructor is different

Formatting

- Slightly different conventions on formatting braces
- Methods start with an uppercase, fields with an underscore
- `https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/inside-a-program/coding-conventions`

Point3D

```csharp

namespace Point3D
{
   public class UsePoint3D
   {
      public static void Main()
      {
         var p = new Point3D(10.0, 20.0, 30.0);    // istantiation with args
         // Point3D q = new Point3D(); // constructor with 0-args no more possible
         Console.WriteLine(p.GetSquareModulus());
      }
   }
   public class Point3D
   {
      private double _x;
      private double _y;
      private double _z;

      public Point3D(double x, double y, double z)
      {
         _x = x;
         _y = y;
         _z = z;
      }

      public double GetSquareModulus()
      {
         return _x * _x + _y * _y + _z * _z;
      }
   }
}
```

C# executable programs

Building blocks of C# software

- class libraries shipped with .NET
- possibly other external libraries
- a set of classes that make up the application we build (like `Point3D`)
- at least one of these classes has a special method `Main`
- a `Main` is the entry point of a program

The `Main` must have the following declaration:

- `public static void Main() {.. }`
- there could be variant with different inputs, outputs, and visibility, but we won't see them now
- it is key it is call `Main` and is `static`
- `static` means this method is "shared" among all objects, and is conceptually called to the class, not to the object

Structure of an executable "project" into a "solution"

Solution

- a folder with many projects, possibly with mutual dependencies
- IDEs work with solutions

Project

- has a name
- has one or many sources
- specify additional properties, and dependencies (references to other projects)

Entry point class

- it contains the `Main` method
- typically, it contains only that method

Other classes

- contain the various application classes

Source files

- have `.cs` extension
- start with "`using`" clauses to declare other classes they use
- declared one or more classes, enclosing them in `namespace`s
- a `namespace` is a "module" giving a context to the class

Working with command line

Creating a new project into your solution

- into a new folder...
- `dotnet new console` – creates a source with top-level hello-world print
- `dotnet new console --use-program-main` – creates a source with namespace, class and main method

Build the project

- `dotnet build`

Run the project

- namely, run the main method...
- `dotnet run`

A class Person

```
public class UsePerson
{
   static void Main(string[] args)
   {
      var p1 = new Person("John", 1980);
      var p2 = new Person("Michael", 1973);
      p2.GotMarried();
     //  Console.WriteLine(p1.ShowAsString()); // John 1980 False
     //  Console.WriteLine(p2.ShowAsString()); // Michael 1973 True
   }
}
public class Person
{
   private string _name;  // string is an alias for String...
   private int _birthYear;
   private bool _married = false;
   public Person(string name, int birthYear)
   {
      _name = name;
      _birthYear = birthYear;
   }
   public void GotMarried()
   {
      _married = true;
   }
   public string ShowAsString()
   {
      return _name + " " + _birthYear + " " + _married;
   }
}
```

Constructors chaining

```
public class UsePerson
{
   public static void Main(string[] args)
   {
      Console.WriteLine(new Person("Bill").ShowAsString());
      Console.WriteLine(new Person("Michael", 1973, true).ShowAsString());
   }
}
public class Person
{
   private string _name;
   private int _birthYear;
   private bool _married;

   public Person(string name, int birthYear, bool married)
   {
      Console.WriteLine("called first constructor");
      _name = name;
      _birthYear = birthYear;
      _married = married;
   }

   public Person(string name) : this(name, 1900, false)
   {
      Console.WriteLine("called second constructor... chaining to the first");
   }

   public string ShowAsString()
   {
      return _name + " " + _birthYear + " " + _married;
   }
}
```

Playing with libraries (namespace System)

```
using System;
// https://docs.microsoft.com/en-us/dotnet/api/system?view=net-6.0

namespace PlayWithLibraries
{
   class Program
   {
      public static void Main()
      {
         Console.WriteLine("The result of 8+2 is " + 10);
         Console.WriteLine("The result of 8+2 is " + (8+2));
         Console.WriteLine("The result of {0}+{1} is {2}",8,2,10);
         var res = 8 + 2;
         Console.WriteLine($"The result of 8+2 is {res}");
         Console.WriteLine($"The result of 8+2 is {2+8}");

         var date1 = new DateTime(2008, 3, 1, 7, 0, 0);
         Console.WriteLine(date1.ToString()); // 3/1/2008 7:00:00 AM
         var date2 = date1.AddMinutes(30);
         Console.WriteLine(date2.ToString()); // 3/1/2008 7:30:00 AM

         var rand = new Random();
         Console.WriteLine("{0},{1}", rand.Next(10), rand.Next(10));
         Console.WriteLine(rand.NextDouble()); // in [0..1]

         Console.Write("Input a number here: ");
         String str = Console.ReadLine(); // read from console
         int number = Int32.Parse(str);   // convert to int (if possible)
         Console.WriteLine(number);
      }
   }
}
```

State, Getters and Setters

An object state

- an object carries a state, in the form of a structure set of data
- internally this is represented by a set of named and typed fields, which are private
- externally this is represented by a set of named and typed "properties"
- such properties may or may not overlap with fields
- to make such properties accessible to clients, specific methods are needed

Getters and Setters

- a common solution in OOP (will see C# will improve it)
- a getter is method `GetXYZ` with 0-args, returning the property `XYZ`'s value, and typically causing no side-effect
- a getter is a method `SetXYZ` taking the property `XYZ`'s value and returning nothing
- properties that one only wants to read have no setter, and vice-versa for getters

Person with Getters and Setters

```
public class Person
{
    private string _name;  // string is an alias for String...
    private int _birthYear;
    private bool _married = false;
    public Person(string name, int birthYear)
    {
        _name = name;
        _birthYear = birthYear;
    }
    public string GetName()
    {
        return _name;
    }
    public int GetBirthYear()
    {
        return _birthYear;
    }
    public bool GetMarried()
    {
        return _married;
    }
    public void SetMarried(bool married)
    {
        _married = married;
    }
    public string GetStringRepresentation()
    {
        return _name + " " + _birthYear;
    }
}
```

Client code for Person

```
public static void Main(string[] args)
{
    var p1 = new Person("John", 1980);
    Console.WriteLine(p1.GetName());
    Console.WriteLine(p1.GetBirthYear());
    Console.WriteLine(p1.GetMarried());
    Console.WriteLine(p1.GetStringRepresentation());
    p1.SetMarried(true);
    Console.WriteLine(p1.GetMarried());
}
```

Expression-bodied members

Syntax: `<member> => expression;`

- can be used for methods and constructors
- when their body is a single return of an expression, or just a single statement...
- you can directly indicate the signature, =>, and that expression/statement
- it makes your programs more short and readable: use them!

Person with Expression-bodied methods

```
public class Person
{
    private readonly string _name;    // readonly field, cannot be changed
    private readonly int _birthYear; // readonly field, cannot be changed
    private bool _married = false;

    public Person(string name, int birthYear)
    {
        _name = name;
        _birthYear = birthYear;
    }

    public string GetName() => _name;

    public int GetBirthYear() => _birthYear;

    public bool GetMarried() => _married;

    public void SetMarried(bool married) => _married = married;

    public string GetStringRepresentation() => _name + " " + _birthYear;
}
```

Immutability

Design for immutability

- by choosing which property has a Setter we can decide that there is information that cannot be changed, and this is important to avoid clients to badly affect the behaviour of our objects

Readonly fields

- the same has to be done for fields: if a field is initialised at construction time and then never changed, we shall use modifier `readonly`
- this enhance clarity of programs, and the compiler check we do not alter such fields

Outline

Properties

Solving the getter/setter dilemma

- we know we never want to expose fields, not to break encapsulation
- still reading/writing "pieces" of an object is very frequent
- getters/setters are verbose and boring, and may hide intent
- need a mechanism with field syntax but getters/setter semantics

Properties: Improving over Get/Set accessors

- C# introduces a programming construct for properties
- a property is syntactically perceived by the client as a sort of field (starting with uppercase)
- semantically however, it has to be considered as a pair of getter and setter
- a `readonly` property is just a getter
- in the class, a property is defined by special convenient syntax

Properties syntax

Case 1: General notation

- `public <type> <name>{ get {...} set {...} }`
- the body of `get` should return a value, of `set` can use a special variable `value`
- for both we can use expression-bodied get/set
- `get` or `set` could be private

Case 2: Auto-implemented properties

- if the body of `get` and `set` are entirely skipped, a field with same name of the property is implicitly defined

Case 3: Expression-bodied getter

- an expression-bodied getter with no parenthesis is perceived as read-only property

Person with Properties (cases 2 and 3)

```
public class UsePerson
{
    public static void Main(string[] args)
    {
        var p1 = new Person("John", 1980);
        Console.WriteLine(p1.Name);
        Console.WriteLine(p1.BirthYear);
        Console.WriteLine(p1.Married);
        Console.WriteLine(p1.StringRepresentation);
        p1.Married = true;
        Console.WriteLine(p1.Married);
    }
}
public class Person
{
    public string Name { get; }     // auto-implemented readonly property
    public int BirthYear { get;  } // auto-implemented readonly property
    public bool Married { get; set; } // auto-implemented read/write property

    public Person(string name, int birthYear)
    {
        Name = name;
        BirthYear = birthYear;
        Married = false;
    }

    // expression-bodied property
    public string StringRepresentation => Name + " " + BirthYear;
}
```

Person with Properties: (cases 1,2 and 3)

```
public class Person
{
   public string Name { get; }    // auto-implemented readonly property
   public int BirthYear { get;  } // auto-implemented readonly property

   private bool _married;

   public bool Married
   {
      get => _married;
      set {
         if (_married && !value)
         {
            Console.WriteLine("can't unmarry!!");
         }
         else
         {
            _married = value;
         }
      }
   }

   public Person(string name, int birthYear)
   {
      Name = name;
      BirthYear = birthYear;
      Married = false;
   }

   public string StringRepresentation => Name + " " + BirthYear;
}
```

Playing with properties: client code

```
public static void Main(string[] args)
{
    var p1 = new Person("John", 1980);
    Console.WriteLine(p1.Name);
    Console.WriteLine(p1.BirthYear);
    Console.WriteLine(p1.Married); // false
    p1.Married = true;   // can marry
    Console.WriteLine(p1.Married); // true
    p1.Married = false;   // can't unmarry, message emitted
    Console.WriteLine(p1.Married); // still true
    Console.WriteLine(p1.StringRepresentation);
}
```

Outline

.NET Arrays

Array types

`T[]`	denotes the array of *T* type
`T[][]`	denotes the array of *arrays of* *T* type
`T[][][]`	denotes the array of *arrays of arrays of* *T* type
`T[][,]`	denotes the array of *2-dimensional arrays of* *T* type
`T[,,][]`	denotes the 3-dimensional array of *arrays of* *T* type

.NET Arrays

Arrays features

- All array types are reference types
  - arrays of value types are reference types as well
- All array types are subtypes of the `Array` class
- Arrays are constructed by sizes, i.e. $D_1, \ldots, D_N$ are user-provided
  - so memory can be contiguously allocated
  - items are initialised to their default values
- All array types come with 3 useful properties/methods:

  `Rank` returning the total amount of dimensions of the array (i.e. N)

  `Length` returning the total amount of items in the array (i.e. $D_1 \times \ldots \times D_N$)

  `GetLength(`i`)` returning the total amount of items along the i-th dimension (i.e. D_i)
- Access to items is performed via the indexed-access operator:

  '`array[index`$_1$`, ..., index`$_N$`]`'

Array Types Instantiation I

Constructors for N-dimensional Arrays of T

$$T\texttt{[,,...]}\ \langle \mathit{Var\ Name}\rangle\ \texttt{=}\ \texttt{new}\ T\texttt{[}D_1\texttt{,}\ D_2\texttt{,}\ \texttt{...];}$$

- Number of commas in the left-hand side: $N-1$
- Number of sizes in the right-hand side: N

Literal Array Expressions for N-dimensional Arrays of T

$$T\texttt{[,,...]}\ \langle \mathit{Var\ Name}\rangle\ \texttt{=}\ \texttt{new}\ T\texttt{[,,...]}\ \ \{\ldots\{\ \langle \mathit{Item}_1\rangle,\ \langle \mathit{Item}_2\rangle,\ \ldots\}\ldots\};$$

- Number of commas in the left-hand side: $N-1$
- Number of nesting levels of braces in the right-hand side: N
- Repeating T`[,,...]` may be avoided in the right-hand side

Array Types Instantiation II

```
Int32[] aLinearArrayOf10Ints = new Int32[10]; // all initialised to 0
Int32[] aLinearArrayOf4Ints = new Int32[] {1, 2, 3, 4};
Int32[] anotherLinearArrayOf4Ints = new [] {1, 2, 3, 4};
Int32[] yetAnotherLinearArrayOf4Ints = {1, 2, 3, 4};
Int32[,] aMatrixOf12Ints = new Int32[4,3]; // all initialised to 0
Int32[,] aMatrixOf6Ints = new Int32[,] {{1, 2, 3}, {4, 5, 6}};
Int32[,] anotherMatrixOf6Ints = {{1, 2}, {3, 4}, {5, 6}};
Int32[,,] a3DArrayOf8Ints = new Int32[2,2,2]; // all initialised to 0
Int32[,,] another3DArrayOf8Ints = {{{1, 2}, {3, 4}}, {{5, 6}, {7, 8}}};
String[] aLinearArrayOf10Strings = new string[10]; // all initialised to null
String[] aLinearArrayOf3Strings = new string[] {"a1", "b2", "c3"};
String[][] anArrayOf10ArraysOfStrings = new string[10][]; // all sub-arrays are to null
String[][] anArrayOf3ArraysOf2Strings = new string[][]
{
    new[] {"a", "b"}, new[] {"c", "d"}, new[] {"e", "f"}
};
```

Accessing Arrays Items I

```
public static void BubbleSort(Int32[] array)
{
    for (Int32 i = 0; i < array.Length; i++)
    {
        for (Int32 j = i - 1; j >= 0; j--)
        {
            if (array[j + 1] < array[j])
            {
                Int32 temp = array[j];     // read array item
                array[j] = array[j + 1];   // read and set array item
                array[j + 1] = temp;       // set array item
            }
        }
    }
}
```

```
public static void FillMatrixRandomly(Int32[,] matrix, Random random)
{
    for (Int32 i = 0; i < matrix.GetLength(0); i++)
    {
        for (Int32 j = 0; i < matrix.GetLength(1); i++)
        {
            matrix[i, j] = random.Next();
        }
    }
}
```

Outline

.NET Nullables

Nullable types definition

- a reference type (classes, interfaces, ...) is nullable, and also...
- let T by a value type of any sort, then T? denotes the nullable T type
- A nullable type T? can be defined as $T \cup \{$ `null` $\}$.
- A variable of type T? can be assigned with any admissible value of T, or with `null`

(cf. `https://docs.microsoft.com/dotnet/csharp/language-reference/builtin-types/nullable-value-types`)

Nullables features

- All nullable types are value types
- The notation T? is another way of writing `Nullable<T>`
- All nullable types come with some useful properties:

  `HasValue` returning null if the object is null

  `Value` returning the non-null value, if present
- When non-null, nullable-type variables behave like they non-nullable counterparts

Nullable Types Operators

- Operator ?? gets the value or a default if null
- Operator ?. calls a method on a nullable only if not null, otherwise it does nothing and yields null

```
    Int32? aNullableInt = null;
    Int32? anotherNullableInt = 1 + aNullableInt; // null

    Int32? aNullableInt2 = 5;
    Int32? anotherNullableInt2 = 1 + aNullableInt2; // 6

    Int32 i = anotherNullableInt ?? 0; // 0
    Int32 i2 = anotherNullableInt2 ?? 0; // 6

    String aString = aNullableInt?.ToString(); // null
    String aString2 = aNullableInt2?.ToString(); // "6"
    String aString3 = aString ?? ""; // ""
```

Outline

Encapsulation

Two crucial ingredients of OO programming

1. Packing data + functions to manipulate it
2. Information hiding via careful access control

Philosophy

- Each class declares `public` only those (few) methods/properties/constructors necessary to interact with (or create) its instances
- The rest (which therefore includes mere implementation aspects) is `private`
  - methods/constructors/properties for internal use only
  - **all** fields (i.e. internal status)

Encapsulation and dependencies

In this way the "client" is weakly influenced by possible future modifications concerning mere implementation aspects.

A basic case: class Counter

```
public class Counter
{
    // the field is made inaccessible
    private int _countValue;

    // it is the constructor that initialises fields
    public Counter()
    {
        _countValue = 0;
    }

    // the only method to observe state
    public int GetValue()
    {
        return _countValue;
    }

    // the only method to modify state
    public void Increment()
    {
        _countValue++;
    }
}
```

A basic case: usage of class `Counter`

```
public class UseCounter
{
    public static void Main(string[] args)
    {
        var counter = new Counter();
        Console.WriteLine(counter.GetValue()); // 0
        counter.Increment();
        counter.Increment();
        counter.Increment();
        Console.WriteLine(counter.GetValue()); // 3
    }
}
```

Encapsulation is preserved by properties!

```
public class UseCounter
{
    public static void Main(string[] args)
    {
        var counter = new Counter();
        Console.WriteLine(counter.Value); // 0
        counter.Increment();
        counter.Increment();
        Console.WriteLine(counter.Value); // 2
    }
}

public class Counter
{
    // an implicitly define field, not modifiable from outside
    // still a well encapsulated solution: one can change implementation
    // of get and set below if needed
    public int Value { get; private set; } = 0;

    public void Increment() => Value++;
}
```

A transparent modification to the Counter implementation

```
public class Counter
{
    // a modified implementation, without required changes in clients
    private int _value;
    private const int MaxValue = 100; // const: essentially a MACRO

    // note the property Value ensures MaxValue is never overcame
    public int Value
    {
        get => _value;
        private set => _value = value<=MaxValue ? value : _value;
    }

    public Counter() => Value = 0;

    public void Increment() => Value++;
}

```

A final bit on properties: object initializers

```
public class Student
{
   public int StudentID { get; set; }
   public string StudentName { get; set; }
   public int Age { get; set; }
   public string Address { get; set; }
}

class UseStudent
{
   static void Main(string[] args)
   {
      // need to have get-set properties
      var std = new Student() {  // object initializer
         StudentID = 1,
         StudentName = "Bill",
         Age = 20,
         Address = "New York"
      };

      // Essentially equivalent to...
      var std2 = new Student(); // default construct is needed
      std2.StudentID = 1;
      std2.StudentName = "Bill";
      std2.Age = 20;
      std2.Address = "New York";
   }
}
```

Properties vs methods vs fields: recap

"Properties are just methods"

- a read-only property is essentially a Getter
- a read-write property is essentially a pair of Getter and Setter method
- most discussions in the following focus on methods, and applicability to properties naturally derive

"Properties define a nice abstraction"

- from the design viewpoint, public properties are much nicer than Getters/Setters, which are still the OOP standard

"Properties can replace fields"

- as an implementation mechanism, auto-implemented properties are a good replacement for fields
- but this is just matter of internal implementation

Outline

C# interfaces

What is an interface

- It is a new declarable reference type (like classes)
- It has a name, and includes a set of method signatures (and properties)
- It cannot be used to create objects by the `new` operator

An interface I can be "implemented" by a class

- Through a class `C` that explicitly declares it (`class C : I {.. }`)
- `C` will define (the body of) all methods declared in `I`
- An instance object of `C`, will have the usual `C` type, but also `I`
- namely, type `C` is a subtype of `I`
- C# convention for interface names: `IDevice`, `IPerson`, ...
- later versions of C# provide default methods for interfaces

Substitutability

- As usual in OOP: an object created by a class implementing an interface can be passed to where an element of the interface is expected

Interface `IDevice`

IDevice introduce a contract for devices: they provide services to be switched on, switched off, and to check if they are on.

```
public interface IDevice
{
    void SwitchOn();

    void SwitchOff();

    bool On { get; }
}
```

Two lamp implementations of Device

```
public class Lamp : IDevice
{
    public void SwitchOn() => On = true;

    public void SwitchOff() => On = false;

    public bool On { get; private set; }

    // additional methods can be added if needed
}

// a lamp with erratic switches
public class ErraticLamp : IDevice
{
    private bool _on = false; // here, want to use a field
    private readonly Random _random = new Random();

    public void SwitchOn() => _on = _random.NextDouble() < 0.95;

    public void SwitchOff() => _on = _random.NextDouble() < 0.05;

    public bool On =>_on;

    // additional methods can be added if needed
}
```

Multiple implementation

Multiple implementation

Possible declaration: `class C : I1, I2, I3 {.. }`

- A class `C` implements `I1` and `I2` and `I3`
- The class `C` ust provide a body for all methods of `I1`, all those of `I2`, all those of `I3`
  - if `I1`, `I2`, `I3` had common methods there would be no problem, each one should be implemented only once
- Instances of `C` have type `C`, but also types `I1`, `I2` and `I3`

Extension

Possible declaration: `interface I : I1, I2, I3 {.. }`

- An interface `I` defines certain methods, in addition to those of `I1`, `I2`, `I3`
- A class `C` that implements `I` must provide a body for all methods indicated in `I`, plus all those of `I1`, all those of `I2`, and all those of `I3`
- Instances of `C` have type `C`, but also types `I`, `I1`, `I2` and `I3`

Outline

Inheritance

It is a mechanism that allows you to define a new class specialising an existing one, that is, “inheriting” its members (the private ones are not directly visible), possibly modifying / adding new members, and therefore reusing code already written and tested.

Inheritance is a key concept of OOP

- It is related to the interface mechanism
- It is one of the key elements along with encapsulation and interfaces
- It not only affects code reuse, but also the resulting polymorphism

Abstract classes

- dealt with in a completely standard way, we won’t consider them further here

Basic example: Counter

```
class UseCounter
{
    public static void Main(string[] args)
    {
        var counter = new Counter(5);
        Console.WriteLine(counter.Value); // 5
        counter.Increment();
        counter.Increment();
        counter.Increment();
        Console.WriteLine(counter.Value); // 8
    }
}

public class Counter
{
    public int Value { get; private set; }

    public Counter(int initialValue) => Value = initialValue;

    public void Increment() => Value++;
}
```

The need to extend and modify

The inheritance mechanism can be used for a multicounter

- Definition: `class C : D {.. }`
- The new `C` class inherits all members of `D`
  - The private members are not directly accessible from within `C`
  - The constructors of `D` must always be rewritten, and should properly call `C`'s
  - The constructor of a subclass should have the `base` statement, which calls a (non-private) constructor of the parent class

```
// Reuse with inheritance: a terser solution!
public class MultiCounter : Counter
{
    public MultiCounter(int initialValue) : base(initialValue)
    {
    }

    public void MultiIncrement(int n)
    {
        for (var i = 0; i < n; i++) Increment();
    }
}
```

`protected` access level

Usable for the members of a class

- It is an intermediate level between `public` and `private`
- Indicates that the member (field, method, constructor, property) is accessible from the current class, from a subclass, and from subclasses of subclasses (recursively)

What is it for?

- It allows subclasses to access supra-class information that you don't want clients to see
- Most often used in retrospect replacing a `private`
- Using `protected` fields is to be avoided – it somewhat breaks encapsulation; should better use `protected` properties/methods

Example class `BiCounter` - bidirectional counter

- A counter with also the `Decrement` method
- Impossible without making the `Value` accessible also for modification

ExtendibleCounter and BiCounter

```
class UseBiCounter
{
    public static void Main(string[] args)
    {
        var counter = new BiCounter(5);
        Console.WriteLine(counter.Value); // 5
        counter.Increment();
        counter.Decrement();
        counter.Decrement();
        Console.WriteLine(counter.Value); // 4
    }
}

public class ExtendibleCounter
{
    public int Value { get; protected set; }

    public ExtendibleCounter(int initialValue) => Value = initialValue;

    public void Increment() => Value++;
}

public class BiCounter : ExtendibleCounter
{
    public BiCounter(int initialValue) : base(initialValue)
    {
    }

    public void Decrement() => Value--;
}
```

Analogous solution with fields

```
public class ExtendibleCounter
{
    private int _value;

    public ExtendibleCounter(int initialValue) => _value = initialValue;

    public int GetValue() => _value;

    protected void SetValue(int value) => _value = value;

    public void Increment() => _value++;
}

public class BiCounter : ExtendibleCounter
{
    public BiCounter(int initialValue) : base(initialValue)
    {
    }

    public void Decrement() => SetValue(GetValue()-1);
}
```

Overriding

Extension and modification

- When creating a new class by extension, it is very often not enough to add new functionality
- Sometimes it is also necessary to modify some of those available, possibly even distorting a bit their original functioning
- This can be done by rewriting in the subclass one (or more) of the methods/properties of the superclass – called an override
- To do so, methods/properties in the base class must be declared `virtual`, and those in the subclass `override`
- If necessary, the rewritten method can invoke the version of the parent using the special receiver `base`
- It is possible to "hide" a method of the superclass that is not `virtual`, by the modifier `new` – but this mechanis is optional, and arguably with limited use
- A class can be declared `sealed` to prevent extension

Example `LimitCounter`

- Create a (sealed) counter which, having reached a certain limit, no longer continues
- It is necessary to override the `Increment()` method
- An additional getter method inspects when the limit is reached

Using the LimitCounter class

```
public class UseLimitCounter
{
    public static void Main(string[] args)
    {
        var limitCounter = new LimitCounter(3);
        Console.WriteLine(limitCounter.Value); // 0
        limitCounter.Increment();
        limitCounter.Increment();
        Console.WriteLine(limitCounter.Value); // 2
        limitCounter.Increment();
        limitCounter.Increment();
        Console.WriteLine(limitCounter.Value); // 3
    }
}
```

Class LimitCounter

```
public class Counter
{
    public int Value { get; protected set; }

    public Counter(int initialValue) => Value = initialValue;

    public virtual void Increment() => Value++;
}

public sealed class LimitCounter : Counter
{
    private readonly int _limit;

    public LimitCounter(int limit) : base(0) => _limit = limit;

    public bool IsOver() => Value >= _limit;

    public override void Increment()
    {
        if (!IsOver()) base.Increment();
    }
}
```

A summary of access modifiers (for types and members)

Recall that `internal` means "visible only in this assembly"

Who can access?

- `public`: any other code in the same assembly or another assembly that references it
- `private`: only by code in the same class
- `protected`: only by code in the same class, or in a class that is derived from that class
- `internal`: by any code in the same assembly, but not from another assembly
- `protected internal`: by any code in the assembly in which it's declared, or from within a derived class in another assembly
- `private protected`: only within its declaring assembly, by code in the same class or in a type that is derived from that class

Class `Object`

Implicit extension of `Object`

- when a class extends nothing, it is like extending `System.Object`
- transitively, this means all classes inherit from `Object`
- it provides low-level services for all objects
- we will in the following explain some of them

Method `String ToString()`

- it can be overriden to provide a canonical string representation of an object
- `Console.Write` uses it if you try to write an object

Outline

Polymorphism with `object`

```csharp
class Program
{
    static void Main(string[] args)
    {
        var objects = new object[5];
        objects[0] = "hello!";
        objects[1] = new object();
        objects[2] = 10; // boxing to a System.Int32
        objects[3] = DateTime.Now;
        objects[4] = new int[] {10, 20, 30};

        PrintAll(objects);
    }

    private static void PrintAll(object[] objects)
    {
        foreach (var obj in objects)
        {
            // Polymorphic call to ToString
            Console.WriteLine(obj.ToString());
            // Giving a representation of obj's type
            Console.WriteLine(obj.GetType());
        }
    }
}
```

UsePerson and Person

```
public class UsePerson
{
    public static void Main(string[] args)
    {
        var people = new Person[]
        {
            new Teacher("Mirko", 521, new string[] {"oop", "softeng"}),
            new Student("Mario", 1001, 2019),
            new Student("Carla", 1002, 2020),
            new Teacher("Giovanni", 522, new string[] {"sistdist"})
        };
        // note the polymorphic call to virtual method ToString...
        foreach (var person in people)
        {
            Console.WriteLine($"{person.Name}; {person.ToString()}");
        }
    }
}
public class Person
{
    public string Name { get; }

    public int Id { get; }

    public Person(string name, int id)
    {
        Name = name;
        Id = id;
    }
}
```

Specialisations of Person

```
public class Student : Person
{
    public int MatriculationYear { get; }

    public Student(string name, int id, int matriculationYear) : base(name, id)
    {
        MatriculationYear = matriculationYear;
    }

    public override string ToString() => $"S[{Name}, {Id}, m:{MatriculationYear}]";
}

public class Teacher : Person
{
    public string[] Courses { get; }

    public Teacher(string name, int id, string[] courses) : base(name, id)
    {
        Courses = courses;
    }

    public override string ToString() => $"T[{Name}, {Id}, c:{string.Join(",",Courses)}]";
}
```

Static type and run-time type

A duality introduced by subtyping (inclusive polymorphism)

- Static type: the data type of an expression that can be inferred by the compiler
- Run-time type: the data type of the value (/ object) actually present (could be a subtype of the static one, and can be inspected with `GetType()`)
  - in this case virtual method calls rely on late-binding

Example in `PrintAll()` code, inside the `foreach`

- Static type of `obj` is `Object`
- Run-time type of `obj` varies from time to time: `String`, `Int32`, ...

Type inspection at run-time

- In some cases it is necessary to inspect the type at run-time
- The case of the `is` and `as` operators
- However, using them is bad practice: it means you have poorly used polymorphism

Type check and conversion

```
public class UsePerson
{
   public static void Main(string[] args)
   {
      var people = new Person[]
      {
         new Teacher("Mirko", 521, new string[] {"oop", "softeng"}),
         new Student("Mario", 1001, 2019),
         new Student("Carla", 1002, 2020),
         new Teacher("Giovanni", 522, new string[] {"sistdist"})
      };
      // printing only teachers
      foreach (var person in people)
      {
         if (person is Teacher) // check for run-time type
         {
            Console.WriteLine($"{person.Name}; {person.ToString()}");
         }
         else
         {
            Student student = person as Student; // type conversion, same reference!
            // Teacher teacher = person as Teacher; // this would yield null!
            Console.WriteLine($"{student.Name}; {student.MatriculationYear}");
         }
      }
   }
}
```

Outline

Uniform abstractions with classes

Uniform abstractions for recurring problems

- During the development of various systems, recurrent problems are encountered that can find a common solution
- In some cases these solutions are factorisable (by abstraction) into a single highly reusable class

A fundamental case: the collection

- A collection is an object whose task is to store the reference to a (typically unspecified) number of other objects
- Among its tasks is to allow modifications and quick access to the set of elements of this collection
- Various strategies can be used, following the theory/practice of algorithms and data structures

Preliminary UseIntVector

```
public class UseIntVectors
{
    public static void Main(string[] args)
    {
        var fibonacci = new IntVector();
        Console.WriteLine(fibonacci); // []
        fibonacci.AddElement(1);
        fibonacci.AddElement(5);
        Console.WriteLine(fibonacci); // [1,5]
        fibonacci.SetElementAt(1,1);
        Console.WriteLine(fibonacci); // [1,1]
        for (var i=2; i< 10; i++)
            fibonacci.AddElement(fibonacci.GetElementAt(i-1)+
                                 fibonacci.GetElementAt(i-2));
        Console.WriteLine(fibonacci); // [1,1,2,3,5,8,13,21,34,55]
    }
}
```

Preliminary IntVector

```
public class IntVector
{
    private const int InitialCapacity = 10;
    private const int MultiplicationFactor = 2;
    private int[] _elements = new int[InitialCapacity];
    public int Size { get; private set; } = 0;

    public void AddElement(int element)
    {
        if (Size == _elements.Length) Expand();
        _elements[Size++] = element;
    }

    private void Expand()
    {
        var old = _elements;
        _elements = new int[old.Length * MultiplicationFactor];
        Array.Copy(old,_elements,Size);
    }

    public int GetElementAt(int position) => _elements[position];

    public void SetElementAt(int position, int element) => _elements[position] = element;

    public override string ToString()
    {
        var s = "[";
        for (var i = 0; i < Size; i++) s += _elements[i] + (i < Size - 1 ? "," : "");
        return s + "]";
    }
}
```

C# Indexers

Let's improve this design a bit

- we have getters/setters that are parametric, namely, they depend on an indexer
- we would like to handle them as we would with properties

C# Indexer

- a sort of parametric property
- it has no name
- it supports the array access notation for reading and/or writing elements
- basic syntax:
  ```
  public <type> this[<type> <name>]{ get{...} set{...}}
  ```
- could have many parameters of the indexer
- could have many indexers in a class, with different parameters

IntVector (with indexer)

```
public class IntVector
{
   private const int InitialCapacity = 10;
   private const int MultiplicationFactor = 2;
   private int[] _elements = new int[InitialCapacity];
   public int Size { get; private set; } = 0;

   public int this[int i]
   {
      get => _elements[i];
      set => _elements[i] = value;
   }

   public void AddElement(int element)
   {
      if (Size == _elements.Length) Expand();
      _elements[Size++] = element;
   }

   private void Expand()
   {
      var old = _elements;
      _elements = new int[old.Length * MultiplicationFactor];
      Array.Copy(old,_elements,Size);
   }

   public override string ToString()
   {
      var s = "[";
      for (var i = 0; i < Size; i++) s += _elements[i] + (i < Size - 1 ? ", " : "");
      return s + "]";
   }
}
```

UseIntVector

```
public class UseIntVectors
{
    public static void Main(string[] args)
    {
        var fibonacci = new IntVector();
        fibonacci.AddElement(1);
        fibonacci.AddElement(5);
        fibonacci[1] = 1;
        for (var i=2; i< 10; i++)
            fibonacci.AddElement(fibonacci[i-1]+fibonacci[i-2]);
        Console.WriteLine(fibonacci); // [1,1,2,3,5,8,13,21,34,55]
    }
}
```

In this lesson we shall assume we never “escape” boundaries of arrays and collections, which would result in exceptions

A first step towards uniformity

Only vectors of `int`?

- Experience would immediately lead to the need to design vectors of `double`, `bool`, ... that is, of any value type
- And then, also vectors of `String`, `DateTime`, and so on
- The implementation would be similar, but without the possibility of reuse.

The idea of "monomorphic" collections

- A first solution to the problem is obtained by exploiting inclusive polymorphism and the "everything is an object" philosophy (including the use of autoboxing)
- Only a `ObjectVector` is created, simply by replacing `int` with `object`
- Any element is inserted (via implicit upcast conversions)
- When you get a value back you need an explicit downcast with `as` operator
- Working with such interfaces gets bloated, and low-level

UseObjectVector

```
public class UseObjectVectors
{
    public static void Main(string[] args)
    {
        // used to store strings... needs annoying "as" operator to retrieve
        var strings = new ObjectVector();
        strings.AddElement("hello");
        strings.AddElement("world!");
        string first = strings[0] as string; // could be null
        string second = strings[1] as string;
        Console.WriteLine(first?.Substring(2)); // "llo"
        // cannot guarantee it contains just strings
        strings.AddElement(DateTime.Now);


        var fibonacci = new ObjectVector();
        fibonacci.AddElement(1); // actually adding a int?
        fibonacci.AddElement(5);
        fibonacci[1] = 1;
        for (var i = 2; i < 10; i++)
        {
            // working with int? is not very optimal, it's "viscose"
            var next = (fibonacci[i - 1] as int?) + (fibonacci[i - 2] as int?);
            if (next!=null) fibonacci.AddElement(next);
        }
        Console.WriteLine(fibonacci); // [1,1,2,3,5,8,13,21,34,55]
    }
}
```

ObjectVector

```
public class ObjectVector
{
   private const int InitialCapacity = 10;
   private const int MultiplicationFactor = 2;
   private object[] _elements = new object[InitialCapacity];
   public int Size { get; private set; } = 0;

   public object this[int i]
   {
      get => _elements[i];
      set => _elements[i] = value;
   }

   public void AddElement(object element)
   {
      if (Size == _elements.Length) Expand();
      _elements[Size++] = element;
   }

   private void Expand()
   {
      var old = _elements;
      _elements = new object[old.Length * MultiplicationFactor];
      Array.Copy(old,_elements,Size);
   }

   public override string ToString()
   {
      var s = "[";
      for (var i = 0; i < Size; i++) s += _elements[i] + (i < Size - 1 ? ", " : "");
      return s + "]";
   }
}
```

Another example of collection: `ObjectList`

```
public class ObjectList
{
    public object Head { get; }
    public ObjectList Tail { get; }

    public ObjectList(object head, ObjectList tail)
    {
        Head = head;
        Tail = tail;
    }

    public object this[int i]  => i == 0 ? Head : Tail[i - 1];

    public int Length => 1 + (Tail?.Length ?? 0);

    public override string ToString()
    {
        var s = "[";
        for (var i = 0; i < Length; i++)
            s += this[i] + (i < Length - 1 ? "," : "");
        return s + "]";
    }
}
```

UseObjectList

```
class UseObjetList
{
   static void Main(string[] args)
   {
      var strings =
         new ObjectList("Hello!",
         new ObjectList("my",
            new ObjectList("World", null)));
      Console.WriteLine(strings.Head+ " " + strings.Tail);
      Console.WriteLine(strings); // [Hello!,my,World]

      // usual problems with the "monomorphic collection" mechanisms
      var numbers =
         new ObjectList(10, new ObjectList(20, null));
      int? sum = (numbers.Head as int?) + (numbers.Tail.Head as int?);
      Console.WriteLine(sum);
   }
}
```

The need for a parametric polymorphism approach

In C# 1.0

- This was the standard approach to building collections

Problem

- With this approach, C# code resulted in many uses of objects similar to `ObjectVector` or `ObjectList`
- It was very easy to lose track of what the content was ...
  - which objects a collection contain? only `int`? only `strings`?
- The code often contained bad conversions

More generally

The problem arises every time I want to collect objects whose type is not known a priori, but could be subject to inclusive polymorphism

Parametric polymorphism

Basic idea: generification

- Given a code snippet `F` that works on a certain type, say `string`, if it could also work uniformly with others...
- ...you make it parametric by replacing `string` with a sort of `T` variable (called type-variable, i.e. a variable that contains a type)
- At this point, when you need the code fragment instantiated on the strings, you use `F<String>`, that is, it is required that `T` becomes `string`
- When you need the code snippet instantiated on integers, use `F<int>`

C# Generics

- Generic classes / interfaces / methods
- Fully integrated in the type system and run-time (differently from other frameworks, like the JVM)
- Typical application with collections

Generic vector

```
public class GenericVector<T>
{
   private const int InitialCapacity = 10;
   private const int MultiplicationFactor = 2;
   private T[] _elements = new T[InitialCapacity];
   public int Size { get; private set; } = 0;

   public T this[int i]
   {
      get => _elements[i];
      set => _elements[i] = value;
   }

   public void AddElement(T element)
   {
      if (Size == _elements.Length) Expand();
      _elements[Size++] = element;
   }

   private void Expand()
   {
      var old = _elements;
      _elements = new T[old.Length * MultiplicationFactor];
      Array.Copy(old,_elements,Size);
   }

   public override string ToString()
   {
      var s = "[";
      for (var i = 0; i < Size; i++) s += _elements[i] + (i < Size - 1 ? "," : "");
      return s + "]";
   }
}
```

Using generic vector

```
public class UseVectors
{
    public static void Main(string[] args)
    {
        // we have an explicit type used to store strings...
        // no annoying "as" operator and nulls to deal with
        GenericVector<string> strings = new GenericVector<string>();
        strings.AddElement("hello");
        strings.AddElement("world!");
        string first = strings[0];
        string second = strings[1];
        Console.WriteLine(strings[0].Substring(2)); // "llo"
        // strings.AddElement(DateTime.Now); // would not compile!


        var fibonacci = new GenericVector<int>();
        fibonacci.AddElement(1); // actually adding a int, not an int?
        fibonacci.AddElement(5);
        fibonacci[1] = 1;
        for (var i = 2; i < 10; i++)
        {
            fibonacci.AddElement(fibonacci[i-2]+fibonacci[i-1]);
        }
        Console.WriteLine(fibonacci); // [1,1,2,3,5,8,13,21,34,55]

    }
}
```

Outline

Generic Methods

Basic idea

- generify a single method in the type(s) of some of its arguments/return
- syntactically: add type parameter after method name
- at the call side: specify type parameter after method name
- at the call side: use type inference, simply avoiding any specification

Two typical applications

- generic static methods: as helpers working on generic structures
- in generic classes: as a helper to mix differen instantiations

Syntax is ad-hoc

- type parameters come after method name in declarations/invocations
- type parameters in invocations can be inferred

Generic static methods

```
class Program
{
   // A method generified in the type of element to create the vector
   public static GenericVector<T> CreateAndFill<T>(int size, T elem)
   {
      var v = new GenericVector<T>();
      for (var i=0; i<size; i++) v.AddElement(elem);
      return v;
   }
   // A method generified in the type of elements of the vector to show
   public static void ShowAll<T>(GenericVector<T> vector)
   {
      for (int i=0; i<vector.Size;i++) Console.Write(vector[i]+" ");
      Console.WriteLine();
   }
   static void Main(string[] args)
   {
      // version with explicit indication of type
      GenericVector<string> vs = CreateAndFill<string>(5, "a");
      // version with type inference
      GenericVector<int> vi = CreateAndFill(5, 10);

      // version with explicit indication of type
      ShowAll<int>(vi);
      // version with type inference
      ShowAll(vs);
   }
}
```

Generic instance methods: the case of class Pair<TA,TB>

```
public class Pair<TA, TB>
{
    public TA First { get; }
    public TB Second { get; }

    public Pair(TA a, TB b)
    {
        First = a;
        Second = b;
    }

    public Pair<TA, TC> ChangeSecond<TC>(TC c) => new Pair<TA, TC>(First, c);

    public Pair<TC, TB> ChangeFirst<TC>(TC c) => new Pair<TC, TB>(c, Second);

    public override string ToString() => $"({First}:{Second})";
}
```

Using pairs

```csharp
class UsePairs
{
   static void Main(string[] args)
   {
      var pair = new Pair<string, int>("ciao", 2);
      Console.WriteLine(pair);
      Console.WriteLine(pair.First.Substring(pair.Second));

      var archive = new GenericVector<Pair<int, string>>();
      archive.AddElement(new Pair<int, string>(1001,"mirko"));
      archive.AddElement(new Pair<int, string>(800,"carla"));
      archive.AddElement(new Pair<int, string>(1003,"mario"));
      Console.WriteLine(archive);

      // name of people with id>1000
      var searchResults = new GenericVector<string>();
      for (var i = 0; i < archive.Size; i++)
      {
         if (archive[i].First > 1000)
            searchResults.AddElement(archive[i].Second);
      }
      Console.WriteLine(searchResults);

      // showcasing generic methods
      Console.WriteLine(pair.ChangeFirst(10)); // a Pair<int,int>
      Console.WriteLine(pair.ChangeSecond("10")); // a Pair<string,string>
   }
}
```

Outline

Generic interfaces

What is a generic interface

- It is an interface that declares type-variables: `interface I <T1, T2> {.. }`
- The type-variables appear in the methods signatures defined by the interface
- When a class implements it, it must instantiate the type variables (or assign them to other type-variables if it is generic)

Uses

To create uniform contracts that do not have to depend on the types used

Example 1

- An `IGenericVector<T>` would be used to abstract over `GenericVector<T>`

Example 2: a new case, Iterators

- An iterator is an object used to access a sequence of elements
- We will now look at a simplified version - different from that of the Java libraries

IGenericVector<T>

```
public interface IGenericVector<T>
{
    int Size { get; }

    T this[int i] { get; set; }

    void AddElement(T element);
}
```

Implementing IGenericVector

```
public class GenericVector<T> : IGenericVector<T>
{
    private const int InitialCapacity = 10;
    private const int MultiplicationFactor = 2;
    private T[] _elements = new T[InitialCapacity];
    public int Size { get; private set; } = 0;

    public T this[int i]
    {
        get => _elements[i];
        set => _elements[i] = value;
    }

    public void AddElement(T element)
    {
        if (Size == _elements.Length) Expand();
        _elements[Size++] = element;
    }

    private void Expand()
    {
        var old = _elements;
        _elements = new T[old.Length * MultiplicationFactor];
        Array.Copy(old,_elements,Size);
    }

    public override string ToString()
    {
        var s = "[";
        for (var i = 0; i < Size; i++) s += _elements[i] + (i < Size - 1 ? "," : "");
        return s + "]";
    }
}
```

Outline

The Iterator pattern

Idea

- Assume an object represents a set of values
  - a collection, a source of information, a mathematical set,. . .
- . . . how could it give a service to let clients retrieve all such values?
- Iterator pattern: the object gives to requestors a so-called iterator
- An iterator is an object with a method to extract the "next" element of the set, to be called iteratively until there are othe objects available
- various implementations possible

Iterator core support in C#

- interface `IEnumerable<T>`: the root of the collection library
  - can use it in `foreach` construct
- interface `IEnumerator<T>`: the actual iterator you can ask to a `IEnumerable`
- both interfaces are connected with the `yield return` construct

IEnumerator<T>, and Range example

```
// namespace System.Collections.Generic
public interface IEnumerator<T> : IDisposable, System.Collections.
  IEnumerator
{
    T Current { get; } // gets current element

    bool MoveNext();  // move cursor to next position, returns if not "over"

    void Reset();     // could be called to get back at the beginning

    void Dispose();   // could be called at the end to release resources
}
```

```
public class UseEnumerator
{
    public static void TestEnumerator()
    {
        IEnumerator<int> enumerator = new RangeEnumerator(0, 5);
        while (enumerator.MoveNext())
        {
            Console.Write(enumerator.Current+" ");
        }
        enumerator.Dispose(); // optional
    }
}
```

RangeEnumerator implementation

```
public class RangeEnumerator : IEnumerator<int>
{
   private readonly int _stop;
   public int Current { get; private set; }

   public RangeEnumerator(int start, int stop)
   {
      Current = start-1;
      _stop = stop;
   }

   public bool MoveNext()
   {
      Current++;
      return Current < _stop;
   }

   // other non-important methods

   public void Reset() // not interested in implementing it
   {
      throw new NotImplementedException();
   }

   public void Dispose() // nothing to do to dispose
   {
   }

   object? IEnumerator.Current => Current; // needed by IEnumerator superinterface
}
```

The actual enumerable: Range

```
public class Range : IEnumerable<int>
{
   public int Start {get; set; }
   public int Stop {get; set; }

   public IEnumerator<int> GetEnumerator() =>
      new RangeEnumerator(Start, Stop);

   // a "dirty" method, due to IEnumerable non-generic superinterface
   IEnumerator IEnumerable.GetEnumerator() =>
      GetEnumerator();
}
```

Using Range with IEnumerator

```
class UseEnumerables
{
    public static void TestEnumerable()
    {
        IEnumerable<int> range = new Range{ Start = 0, Stop = 5};
        var enumerator = range.GetEnumerator();
        while (enumerator.MoveNext())
        {
            Console.Write(enumerator.Current+" ");
        }
        enumerator.Dispose();
        Console.WriteLine();

        var enumerator2 = range.GetEnumerator();
        while (enumerator2.MoveNext())
        {
            if (enumerator2.Current > 2)
            {
                Console.Write(enumerator2.Current+" ");
            }
        }
        enumerator2.Dispose();
    }
}
```

foreach is compatile with IEnumerables!

```
static void TestForeach()
{
    var range = new Range(); // default construction
    range.Start = 0;
    range.Stop = 5;
    foreach (var i in range){
        Console.Write(i+" ");
    }
    // note that foreach also calls Dispose
    Console.WriteLine("");

    // using object initializer
    foreach (var i in new Range{ Start = 0, Stop = 5}){
        Console.Write(i+" ");
    }
}
```

The `yield return` construct

```
public class Helpers
{
    public IEnumerable<int> RangeWithDelta(int start, int stop, int delta)
    {
        for (var i = start; i != stop; i += delta) yield return i;
    }

    public IEnumerable<string> Directions()
    {
        yield return "NORTH";
        yield return "EAST";
        yield return "SOUTH";
        yield return "WEST";
    }

    public IEnumerator<int> FibonacciInfiniteEnumerator()
    {
        yield return 1;
        int a = 1;
        int b = 1;
        while (true)
        {
            yield return b;
            int sum = a + b;
            a = b;
            b = sum;
        }
    }
}
```

The yield return construct

```
class UseYieldReturn
{
    public static void Main(string[] args)
    {
        var helpers = new Helpers();

        foreach (var i in helpers.RangeWithDelta(0, -10, -2))
        {
            Console.Write(i + " ");
        }
        Console.WriteLine("");
        foreach (var d in helpers.Directions())
        {
            Console.Write(d + " ");
        }
        Console.WriteLine("");
        var fib = helpers.FibonacciInfiniteEnumerator();
        while (true)
        {
            fib.MoveNext();
            Console.Write(fib.Current + " ");
            if (fib.Current > 1000) break;
        }

        fib.Dispose();
    }
}
```

Creating a vector as an IEnumerable<T>

```
public class EnumerableVector<T> : IEnumerable<T>
{
    private const int InitialCapacity = 10;
    private const int MultiplicationFactor = 2;
    private T[] _elements = new T[InitialCapacity];
    public int Size { get; private set; } = 0;
    public T this[int i] => _elements[i];

    public void AddElement(T element)
    {
        if (Size == _elements.Length) Expand();
        _elements[Size++] = element;
    }

    private void Expand()
    {
        var old = _elements;
        _elements = new T[old.Length * MultiplicationFactor];
        Array.Copy(old,_elements,Size);
    }

    public IEnumerator<T> GetEnumerator()
    {
        for (var i = 0; i < Size; i++) yield return _elements[i];
    }

    IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
}
```

Enumerating our vector

```
public class UseEnumerableVectors
{
    public static void Main(string[] args)
    {
        var fibonacci = new EnumerableVector<int>();
        fibonacci.AddElement(1);
        fibonacci.AddElement(1);
        for (var i = 2; i < 10; i++)
        {
            fibonacci.AddElement(fibonacci[i-2]+fibonacci[i-1]);
        }

        foreach (var i in fibonacci) Console.WriteLine(i);
    }
}
```

IEnumerable<T> and collections

The collection framework very briefly (will be explored next)

- various implementations available for collecting data
- all implement interface `IEnumerable<T>`
- all have standard methods `Add`, indexers, and so on

```
public class ShowCollections
{
    public static void Main(string[] args)
    {
        // IList and List are from System.Collections.Generic

        IList<int> fibonacci = new List<int>();
        fibonacci.Add(1);
        fibonacci.Add(1);
        for (var i = 2; i < 10; i++)
        {
            fibonacci.Add(fibonacci[i-2]+fibonacci[i-1]);
        }

        foreach (var i in fibonacci) Console.WriteLine(i);
    }
}
```

An example application: ClassManagement

```
class Program
{
   static void Main(string[] args)
   {
      var classManagement = new ClassManagement();
      classManagement.AddStudent(new Student{ Id = 100, Name = "Maria"});
      classManagement.AddStudent(new Student{ Id = 101, Name = "Gino", Evaluation = 30});
      classManagement.AddStudent(new Student{ Id = 102, Name = "Marco", Evaluation = 18});

      foreach (var student in classManagement.GetAll()) Console.WriteLine(student);
      Console.WriteLine("---");
      foreach (var name in classManagement.GetNames()) Console.WriteLine(name);
      Console.WriteLine("---");
      foreach (var student in classManagement.GetStudentsWithEvaluation())
         Console.WriteLine(student);
   }
}

public class Student
{
   public int Id { get; set; }
   public string Name { get; set; }
   public int? Evaluation { get; set; }

   public override string ToString()
   {
      return $"Id: {Id}, Name: {Name}, Evaluation: {Evaluation}";
   }
}
```

An example application: ClassManagement

```
interface IClassManagement
{
   void AddStudent(Student student);

   IEnumerable<Student> GetAll();
   IEnumerable<string> GetNames();
   IEnumerable<Student> GetStudentsWithEvaluation();
}

public class ClassManagement : IClassManagement
{
   private IList<Student> _students = new List<Student>();

   public void AddStudent(Student student) => _students.Add(student);

   public IEnumerable<Student> GetAll()
   {
      foreach (var student in _students) yield return student;
   }

   public IEnumerable<string> GetNames()
   {
      foreach (var student in _students) yield return student.Name;
   }

   public IEnumerable<Student> GetStudentsWithEvaluation()
   {
      foreach (var student in _students)
      {
         if (student.Evaluation != null) yield return student;
      }
   }
}
```

Outline

Constrained Polymorphism

Consider a generic class `C<T>` or method `M<T>(...)`

- what operations are we allowed to perform on elements of type `T`?
- we can only assume it is an `object`, hence e.g. `ToString`
- how can we express something more?

`where` clauses in type parameters

- `where T : class`: used to mean that `T` should be a reference type
- `where T : new()`: used to mean that `T` must have a 0-ary constructor
- `where T : Lamp`: used to mean that `T` must be a subtype of `Lamp` class
- `where T : ILamp`: used to mean that `T` must be a subtype of `ILamp` interface
- `where T : U`: used to mean that `T` must be a subtype of another type parameter `U`

Examples of Constrained Polymorphism

```
class Program
{
   static void Main(string[] args)
   {
      ShowAll(Create<object>(5));
      ShowAll(Create<DateTime>(5));
      // ShowAll(Create<IList>(5)); // would notwork!

      var list1 = new List<object>(new object[]{1, true, new DateTime()});
      var list2 = new List<int>(new int[]{1, 2,3,4,5});
      Copy(list2, list1);
      ShowAll(list1);
      // Copy(list1, list2); // would not work!
   }

   public static void ShowAll<T>(IEnumerable<T> enumerable)
   {
      foreach(var t in enumerable) Console.Write(t+" ");
      Console.WriteLine();
   }

   public static List<T> Create<T>(int size) where T: new()
   {
      var l = new List<T>();
      for (var i=0; i<10; i++) l.Add(new T());
      return l;
   }

   public static void Copy<TFrom,TTo>(List<TFrom> from, List<TTo> to) where TFrom: TTo
   {
      foreach (var t in from) to.Add(t);
   }
}
```

Outline

Deepening: on the substitutability of generics

Question: `List<string>` is a subtype of `List<object>`?

That is, can we think of passing a `List<string>` in all contexts where a `List<object>` is expected instead?

Answer: no!! It would seem so .. but:

what happens if in the method below we pass a `List<string>`?
$\Rightarrow$ we could easily compromise the integrity of the list

```
static void Main(string[] args)
{
    var list = new List<string>(new[] {"10", "20", "30"});
    // AddAString(list,-1); // This code should not work, otherwise...
    String s = list[list.Count - 1];
}

public static void AddAnElement(List<object> list, object last)
{
    list.Add(last);
}
```

Unsafety with C# arrays

C# arrays are treated as covariants!

- Arrays look a lot like a generic type
- string[] ~ List<string>, T[] ~ List<T>
- And so we know it wouldn't be safe to handle them with covariance
- But in C# it is exactly like this!! E.g. string[] <: object[]
- So any write to array could potentially fail .... throwing an exception

```
static void Main(string[] args)
{
   string[] s = new[] {"a", "b", "c"};
   SetToArray(s,10); // It works!
   string str = s[0]; // It throws a: System.ArrayTypeMismatchException:
}

public static void SetToArray(object[] array, object element)
{
   array[0] = element;
}
```

Covariance and access operations

Covariance (`C<T> <: C<S>` with `T<:S`) would be admissible if:

- The `C<X>` class had no operations receiving `X` objects
- That is, it has only private or readonly fields and no methods with `X` as argument

Contravariance (`C<S> <: C<T>` with `T <: S`) would be admissible if:

- The `C<X>` class had no operations that produce `X` objects
- That is, it has only private fields and no method with return type `X`

In practice:

- Most of the generic `C<X>` classes have fields of type `X` (composition) and getter and setter operations, and therefore their covariance and contravariance would not work
- C# allows indication of covariance or contravariance in generic interfaces for which it is safe to do so, by keywords `in` and `out`
- this allows to nicely deal with reusability of generic methods, as will see in the collection framework

An advanced example of “variant” modelling

```
public interface IBaseVector
{
    int Size { get;}
}

public interface IReadVector<out T> : IBaseVector, IEnumerable<T>
{
    T this[int i] { get; }
}

public interface IWriteVector<in T>
{
    void AddElement(T t);
}

public interface IVector<T> : IReadVector<T>, IWriteVector<T>
{
}
```

Expectation

```
class Program
{
    static void Main(string[] args)
    {
        IVector<string> vs = new GenericVector<string>();
        AddManyStrings(vs, 5);
        IVector<object> vo = new GenericVector<object>();
        AddManyStrings(vo, 5); // IWriteVector<object> <: IWriteVector<string>
        IVector<Pair<string, string>> vp = new GenericVector<Pair<string, string>>();
        vp.AddElement(new Pair<string, string>("a","b"));
        Copy(vs, vo); // same as Copy<object>(vs, vo), or even Copy<string>(vs, vo)
        Copy(vp, vo); // IReadVector<Pair<string,string>> <: IReadVector<object>
        foreach (var o in vo) Console.WriteLine(o);
    }
```

Straightforward implementation

```
public class GenericVector<T>: IVector<T>
{
   private const int InitialCapacity = 10;
   private const int MultiplicationFactor = 2;
   private T[] _elements = new T[InitialCapacity];
   public int Size { get; private set; } = 0;

   public T this[int i] => _elements[i];


   public void AddElement(T element)
   {
      if (Size == _elements.Length) Expand();
      _elements[Size++] = element;
   }

   private void Expand()
   {
      var old = _elements;
      _elements = new T[old.Length * MultiplicationFactor];
      Array.Copy(old,_elements,Size);
   }

   public IEnumerator<T> GetEnumerator()
   {
      for (var i = 0; i < Size; i++) yield return _elements[i];
   }

   IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
}
```

Outline

.NET Exceptions

Exceptions as types

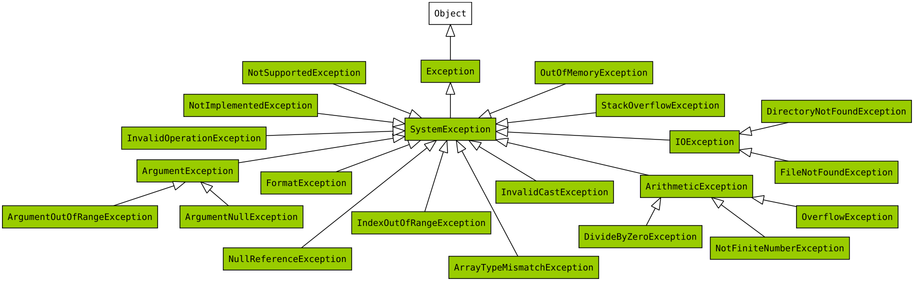
- .NET exceptions are *reference* types
  - in particular, they are sub-classes of `System.Exception`
- whose instances can be thrown to denote
  - an illegal operation/input/state—situation, in general
  - an unexpected value
  - some invariant of a class being violated
  - an exceptional result

Exception-related activities and **who** performs them

throw — performed by libraries *implementors* (standard mechanism)
catch — optionally performed by libraries *users* (standard mechanism)
design — (rarely) performed by libraries *designers*

! .NET class library is full of re-usable exceptions

.NET Exception Type Hierarchy (non-exaustive)



The throw Statement

What happens on a throw?

1. The control flow is interrupted
2. The control flow is given to any method in the current call stack which is capable of catching current exception
3. If none is found, the program execution is interrupted
4. . . . and the exception's stack trace is shown.

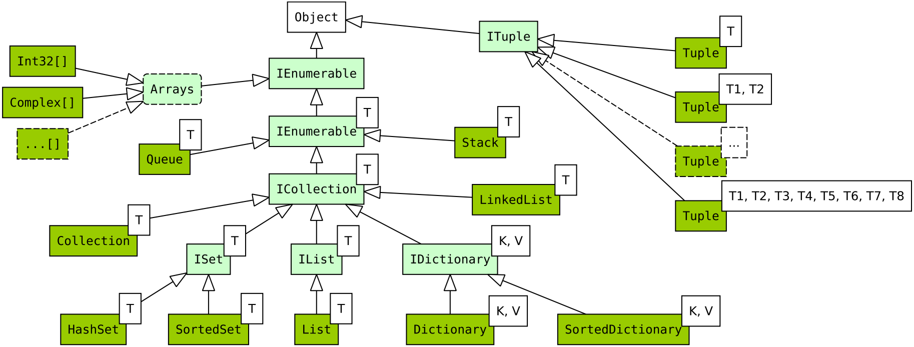
```
static void Recursive(int x)
{
    if (x < 5)
    {
        Recursive(x + 1);
        Console.WriteLine(x);
    }
    else
    {
        throw new Exception("" + x);
    }
}
```

The try-catch-finally Construct

```
int readLines = 0;
while (true)
{
    String line = null;
    try
    {
        Console.Write("> ");
        line = Console.ReadLine();
        int number = int.Parse(line);
        Console.WriteLine("Valid integer: " + number);
    }
    catch (FormatException e) { Console.WriteLine("Not a valid integer: " + line); }
    catch (OverflowException e) { Console.WriteLine("Out of range integer: " + line); }
    catch (ArgumentNullException e) { Console.WriteLine("Bye bye!"); break; }
    finally { Console.WriteLine($"(So far I read {++readLines} lines)"); }
}
```

Outline

Generic Collections and their Type Hierarchy I



- Notice the many type parameters (white rectangles)
- All dark-green boxes represent classes which can be exploited in everyday programming

Generic Collections and their Type Hierarchy II

`IEnumerable` is the type of all data structures...

...containing a number of `objects`, which can be iterated

`IEnumerable<T>` is the type of all `IEnumerables`...

...containing a number of instances of `T`, which can be iterated

`ICollection<T>` is the super-type of all collections

Most notably, it is the super-type of:

- `ISet<T>` i.e. the type of all sets whose items are of type `T`
- `IList<T>` i.e. the type of all lists whose items are of type `T`
- `IDictionary<K, V>` i.e. the type of all maps whose keys (resp. values) are of type `K` (resp. `V`)
  - in this case `T` ≡ `KeyValuePair<K, V>`

    ie a structure defined in `System.Collections.Generic`

Generic Collections and their Type Hierarchy III

`ITuple` is the super-type of all sorts of tuples

Most notably, it is the super-type of:

`Tuple<T1, T2>` i.e. the type of all pairs whose first item is of type `T1` and whose second item is of type `T2`

`Tuple<T1, T2, T3>` i.e. the type of all triples whose first item is of type `T1`, whose second item is of type `T2`, whose third item is of type `T3`

⋮

Generic Collections and their Type Hierarchy IV

Remarks

- Most of these types are in `System.Collections.Generic`
- All sorts of arrays and collections are enumerable
- Tuples are not enumerable
- Surprisingly, `Queues`, `Stacks`, and `LinkedLists` are not `ILists`
- Another hierachy exists, rooted in `ICollection`, for non-generic types
  - these are contained into `System.Collections`
- Another hierachy exists, rooted in `IReadOnlyCollection<out T>`, for read-only types
  - these are contained into `System.Collections.Generic`
- Other types exist, named `IImmutable*<T>`, for immutables types
  - these are contained into `System.Collections.Immutable`

The ICollection<T> Interface I

The System.Collections.Generic.ICollection<T> Interface

```
// Generic container of items of type T
public interface ICollection<T> : IEnumerable<T>, IEnumerable
{
    // Gets the amount of items in the collection
    int Count { get; }

    // Adds an item to the collection
    void Add(T item);

    // Removes all the items from the collection, emptying it
    void Clear();

    // Checks whether item is contained in the collection or not
    bool Contains(T item);

    // Inserts all the items of the collection into array, starting from arrayIndex
    void CopyTo(T[] array, int arrayIndex);

    // Removes item from the collection
    bool Remove(T item);

    // RECALL THAT GetEnumerator() IS INHERITED!
}
```

The `ICollection<T>` Interface II

Creation

- Use the following syntax to provide items on the fly:

  new ⟨*Collection Class*⟩() { ⟨*Item*$_1$⟩, ..., ⟨*Item*$_N$⟩ }

  - where all ⟨*Item*$_i$⟩ are expressions returning instances of `T`

Usage

- The metods of `ICollection<T>` support all basic operations
  - except directly reading a particular item
- By default, they support a mutable operation approach
- Converting a collection into string does not shows its items!

The ICollection<T> Interface III

Creating/Usage example for the Collection class

```
ICollection<ExampleItem> collection = new Collection<ExampleItem>() {
    new ExampleItem(2),
    new ExampleItem(3),
    new ExampleItem(1),
    new ExampleItem(4)
};

Console.WriteLine(collection.Count); // 4

Console.WriteLine(collection.Contains(new ExampleItem(5))); // false

collection.Add(new ExampleItem(5)); // item 5 is added
Console.WriteLine(collection.Contains(new ExampleItem(5))); // true
Console.WriteLine(collection.Count); // 5

Console.WriteLine(collection.Contains(new ExampleItem(1))); // true
collection.Remove(new ExampleItem(1)); // item 1 is removed
Console.WriteLine(collection.Count); // 4
Console.WriteLine(collection.Contains(new ExampleItem(1))); // false

foreach (var item in collection)
{
    Console.WriteLine(item); // 2, 3, 4, 5
}

Console.WriteLine(collection.ToString()); // ???
```

The IList<T> Interface I

The System.Collections.Generic.IList<T> Interface

```
// Collection storing items in an orderly fashion
// (indexes are 0-based)
public interface IList<T> : ICollection<T>, IEnumerable<T>, IEnumerable
{
    // Get or sets the index-th item in the list
    T this[int index] { get; set; }

    // Gets the index of an item in the list, or -1 if missing
    int IndexOf(T item);

    // Adds item in position index (subsequent items are shifted right)
    void Insert(int index, T item);

    // Removes the item in position index (subsequent items are shifted left)
    void RemoveAt(int index);

    // RECALL THAT METHODS FROM Object, ICollection<T>, and IEnumerable<T> ARE INHERITED!
}
```

The IList<T> Interface II

Creating/Usage example for the List class

```
IList<ExampleItem> list = new List<ExampleItem>() {
    new ExampleItem(2),
    new ExampleItem(3),
    new ExampleItem(1),
    new ExampleItem(4)
};

foreach (var item in list)
{
    Console.WriteLine(item); // 2, 3, 1, 4
}

Console.WriteLine(list[1]); // 3
list[1] = new ExampleItem(5);
Console.WriteLine(list[1]); // 5
Console.WriteLine(list[2]); // 1

list.Insert(2, new ExampleItem(6));
Console.WriteLine(list[2]); // 6
Console.WriteLine(list[3]); // 1

Console.WriteLine(list.IndexOf(new ExampleItem(1))); // 3
list.RemoveAt(3); // removing item in position 3, i.e. 1
Console.WriteLine(list.IndexOf(new ExampleItem(1))); // -1

Console.WriteLine(list.ToString()); // ???
```

The ISet<T> Interface I

The System.Collections.Generic.ISet<T> Interface

```
// Collection storing items with no duplicates, regardless of their ordering
public interface ISet<T> : ICollection<T>, IEnumerable<T>, IEnumerable
{
    // Adds an item to the set, returning false if it was already present, true otherwise
    bool Add(T item);

    // Removes all elements in the enumerable from the current set
    void ExceptWith(IEnumerable<T> other);

    // Modifies the current set so that it contains only elements
    // that are contained in both the current set and the provided enumerable
    void IntersectWith(IEnumerable<T> other);

    // Returns true if all the items of the set are contained in the enumerable,
    // and the enumerable contains other items as well
    bool IsProperSubsetOf(IEnumerable<T> other);

    // Returns true if all the items of the enumerable are contained in current set,
    // and the current set contains other items as well
    bool IsProperSupersetOf(IEnumerable<T> other);

    // Returns true if all the items of the set are contained in the enumerable
    bool IsSubsetOf(IEnumerable<T> other);

    // Returns true if all the items of the enumerable are contained in current set
    bool IsSupersetOf(IEnumerable<T> other);

```

The ISet<T> Interface II

```
    // Returns true if some item of the current set is in the enumerable as well
    bool Overlaps(IEnumerable<T> other);

    // Returns true if the current set and the specified collection contain the same
  elements
    bool SetEquals(IEnumerable<T> other);

    // Modifies the current set so that it contains only those items that
    // are present either in the current set or in the enumerable, but not both
    void SymmetricExceptWith(IEnumerable<T> other);

    // Modifies the current set so that it contains both its original items and the ones
    // in the provided enumerable, without repetitions
    void UnionWith(IEnumerable<T> other);

    // RECALL THAT METHODS FROM Object, ICollection<T>, and IEnumerable<T> ARE INHERITED!
}
```

The ISet<T> Interface III

Creating/Usage example for the HashSet class

```
ISet<ExampleItem> set = new HashSet<ExampleItem>() {
    new ExampleItem(2), new ExampleItem(3), new ExampleItem(2),
    new ExampleItem(1), new ExampleItem(2)
};

foreach (var item in set) Console.WriteLine(item); // 1, 2, 3 (in some order)

Console.WriteLine(set.Count); // 3
Console.WriteLine(set.Add(new ExampleItem(3))); // false
Console.WriteLine(set.Count); // 3

var anotherSet = new HashSet<ExampleItem>() { new ExampleItem(1), new ExampleItem(2) };

Console.WriteLine(set.IsProperSupersetOf(anotherSet)); // true
Console.WriteLine(anotherSet.IsProperSubsetOf(set)); // true
anotherSet.Add(new ExampleItem(4));
Console.WriteLine(set.IsProperSupersetOf(anotherSet)); // false
Console.WriteLine(anotherSet.IsProperSubsetOf(set)); // false
Console.WriteLine(set.Overlaps(anotherSet)); // true
set.UnionWith(anotherSet);

foreach (var item in set) Console.WriteLine(item); // 1, 2, 3, 4 (in some order)

Console.WriteLine(set.ToString()); // ???
```

- consider retrying the same test with the SortedSet<T> class

The IDictionary<K, V> Interface I

The System.Collections.Generic.IDictionary<K, V> Interface

```
// A collection containing a number of key-value pairs
// where keys are of type K (and cannot be null) and values are of type V
public interface IDictionary<K, V> : ICollection<KeyValuePair<K, V>> where K : notnull
{
    // Gets or sets the element with the specified key
    V this[K key] { get; set; }

    // Returns a collection containing all the keys used in the current dictionary
    ICollection<K> Keys { get; }

    // Returns a collection containing all the values used in the current dictionary
    ICollection<V> Values { get; }

    // Adds a new key-value pair. Throws an exception if key is already present
    void Add(K key, V value);

    // Returns true if a pair indexed by key is present, false otherwise
    bool ContainsKey(K key);

    // Removes a key and its corresponding pair. Returns null if the key is missing
    bool Remove(K key);

    // RECALL THAT METHODS FROM Object, ICollection<T>, and IEnumerable<T> ARE INHERITED!
}
```

Creating/Usage example for the Dictionary class

```
IDictionary<string, ExampleItem> map = new Dictionary<string, ExampleItem>() {
    ["giovanni"] = new ExampleItem(1),
    ["mirko"] = new ExampleItem(2),
    ["andrea"] = new ExampleItem(3)
};

foreach (KeyValuePair<string, ExampleItem> item in map)
{
    Console.WriteLine(item.Key); // giovanni, mirko, andrea
    Console.WriteLine(item.Value); // 1, 2, 3
}

foreach (var (key, val) in map)
{
    Console.WriteLine(val); // 1, 2, 3
    Console.WriteLine(key); // giovanni, mirko, andrea
}

map["andrea"] = new ExampleItem(0); // here the value of key "andrea" is being replaced
map.Add("matteo", new ExampleItem(3)); // here a novel key-value pair is being added
map["danilo"] = new ExampleItem(4); // here a novel key-value pair is being added

try
{
    map.Add("matteo", new ExampleItem(4)); // fails as key "matteo" is already present
}
catch (ArgumentException e) { /* ignore */ }
```

The IDictionary<K, V> Interface III

```
Console.WriteLine(map["giovanni"]); // 1
Console.WriteLine(map["mirko"]); // 2
Console.WriteLine(map["andrea"]); // 0
Console.WriteLine(map["matteo"]); // 3
Console.WriteLine(map["danilo"]); // 4

foreach (string key in map.Keys)
    Console.WriteLine(key); // giovanni, mirko, andrea, matteo, danilo (in some order)
foreach (ExampleItem val in map.Values)
    Console.WriteLine(val); // 1, 2, 0, 3, 4 (in some order)

Console.WriteLine(map.ToString()); // ???
```

- consider retrying the same test with the SortedDictionary<K, V> class

The Tuple<T1, T2, ...> Classes I

The System.Tuple<T1, T2, ...> Class

```
// A type for IMMUTABLE triplets of items where the 1st one is of type T1,
// the 2nd one is of type T2, and the 3rd one is of type T3
public class Tuple<T1, T2, T3>
{
    private readonly object[] _items;

    public Tuple(T1 x, T2 y, T3 z) { _items = new object[] {x, y, z}; }

    public T1 Item1 => (T1) _items[0]; // Gets the first item
    public T2 Item2 => (T2) _items[1]; // Gets the second item
    public T3 Item3 => (T3) _items[2]; // Gets the third item

    public object this[int index] => _items[index];

    int Length => _items.Length;

    // method Equals compares triplets by value
    // method GetHashCode is coherent w.r.t. equals

    // method ToString prints the items
}
```

Creating/Usage example for the Dictionary class

```
Tuple<string, int> pair = Tuple.Create("giovanni", 29);
Tuple<string, int, DateTime> triplet = Tuple.Create("mirko", 45, DateTime.Now);

var number = 30;
Tuple<string, int> anotherPair = Tuple.Create("giovanni", number - 1);
Tuple<string, int, DateTime> anotherTriplet = Tuple.Create("mirko", 45, DateTime.Now);

Console.WriteLine(pair.ToString()); // (giovanni, 29)
Console.WriteLine(anotherPair.ToString()); // (giovanni, 29)
Console.WriteLine(triplet.ToString()); // (mirko, 45, 19/03/2021 16:33:30)
Console.WriteLine(anotherTriplet.ToString()); // (mirko, 45, 19/03/2021 16:33:30)

Console.WriteLine(pair.Equals(anotherPair)); // true
Console.WriteLine(triplet.Equals(anotherTriplet)); // false ...... WHY? :)

Console.WriteLine(triplet.Item1 == anotherTriplet.Item1); // true
Console.WriteLine(triplet.Item2 == anotherTriplet.Item2); // true
Console.WriteLine(triplet.Item3 == anotherTriplet.Item3); // false ...... WHY? :)
```

Outline

Improving functional strategies

Functional strategies

- a specific case of the so-called Strategy pattern
- methods/constructors accept arguments of an interface with just one method, one or more implementing classes, the need of creating an object when the strategy is to be defined
- the whole idea is actually simply that of a "function"

The C# path across time: how to instantiate a delegate

- wrapping method references (C# 1.0)
- anonymous functions (C# 2.0)
- lambdas (C# 3.0) – the one suggested now

Consider a `BankAccount`

With methods for:

- withdrawing
- checking balance

Abstracting a `IFeeCalculator`

- the hypothesis that withdrawal fee is 1 appears weak
- surely this has to be generalise
- it could vary across different accounts
- it could vary dependning on the actual amount
- could introduce an interface `IFeeCalculator`

Abstracting a `IWithdrawalAction`

- what should we do if withdrawal fail?
- surely that task is better left to a different class
- in this case, perhaps many different behaviour have to be defined
- shuold abstract from that specific behaviour
- could introduce an interface `IWithdrawalAction`

Interfaces

```
public interface IFeeCalculator
{
   int Fee(int amount);
}

public interface IWithdrawAction
{
   void Handler(int balance, int amount);
}

public class StandardFee : IFeeCalculator
{
   public int Fee(int amount) => 1;
}

public class BusinessFee : IFeeCalculator
{
   public int Fee(int amount) => amount > 100 ? 1 : 0;
}

public class OnConsoleAction : IWithdrawAction
{
   public void Handler(int balance, int amount) =>
      Console.WriteLine("Could not withdraw "+amount);
}

public class ErrorAction : IWithdrawAction
{
   public void Handler(int balance, int amount) =>
      Console.Error.WriteLine($"ERROR on Withdrawal: {balance}:{amount}");
}
```

FlexibleBankAccount

```
public class FlexibleBankAccount : IBankAccount
{
    public int Balance { get; private set; }
    public string Name { get; }

    private IFeeCalculator _feeCalculator;

    private ISet<IWithdrawAction> _actions;

    public FlexibleBankAccount(string name, IFeeCalculator feeCalculator, ISet<
   IWithdrawAction> actions)
    {
        _feeCalculator = feeCalculator;
        _actions = actions;
        Name = name;
    }

    public void Deposit(int amount) => Balance += amount;

    public void Withdraw(int amount)
    {
        var fee = _feeCalculator.Fee(amount);
        if (Balance < amount + fee)
        {
            foreach (var wa in _actions) wa.Handler(Balance, amount);
        }
        else Balance -= (amount + fee);
    }

    public override string ToString() => $"Name: {Name}, Balance: {Balance}";
}
```

UseFlexibleBankAccount

```
class UseFlexibleBankAccounts
{
   static void Main(string[] args)
   {
      var actions1 = new HashSet<IWithdrawAction>(new []{new OnConsoleAction()});
      var ba1 = new FlexibleBankAccount("a", new BusinessFee(), actions1);
      ba1.Deposit(1000); // 1000
      ba1.Withdraw(50);  // 950
      ba1.Withdraw(150); // 699
      ba1.Withdraw(1000); // 699, + console output
      Console.WriteLine(ba1.Balance); // 699

      var actions2 = new HashSet<IWithdrawAction>(
         new IWithdrawAction[]{new OnConsoleAction(), new ErrorAction()});
      var ba2 = new FlexibleBankAccount("a", new StandardFee(), actions2);
      ba2.Deposit(1000); // 1000
      ba2.Withdraw(50);  // 949
      ba2.Withdraw(150); // 798
      ba2.Withdraw(1000); // 798, + console/error outputs
      Console.WriteLine(ba2.Balance); // 700
   }
}
```

C# delegates

Delegate

- a delegate is a wrapper for a method (reference) of a specific type (input/output arguments)
- defining a delegate `D` means to define one such type and giving it a name
- syntax: `delegate <ret-type> D(<parameters list>);`
- "under the hood" this is just a subclass of `System.Delegate`

Instantiation of a delegate by method reference

- if there is a method `M` (static or instance) accessible in scope
- you can define a variable of type `D` assigned to method `M`
- syntax: `D del = new D(M);`, or simply passing `M` where a `D` is expected

Call of a delegate

- given the above definition, simply `del` is like a method to be called

Multichannel delegates

- given two delegates that return void (also called even delegates), they can be combined by operators +, −, +=, −=

Definition and use of delegates

```
public delegate int ComputeFee(int amount);
public delegate void WithdrawAction(int balance, int amount);

public class FlexibleBankAccount : IBankAccount
{
   public int Balance { get; private set; }
   public string Name { get; }

   private ComputeFee _computeFee;

   private WithdrawAction _action;

   public FlexibleBankAccount(string name, ComputeFee computeFee, WithdrawAction action)
   {
      _computeFee = computeFee;
      _action = action;
      Name = name;
   }

   public void Deposit(int amount) => Balance += amount;

   public void Withdraw(int amount)
   {
      var fee = _computeFee(amount);
      if (Balance < amount + fee) _action(Balance, amount);
      else Balance -= (amount + fee);
   }

   public override string ToString() => $"Name: {Name}, Balance: {Balance}";
}
```

Instantiation of delegates

```
class UseFlexibleBankAccounts
{
   private static int StandardFee(int amount) => 1;
   private static int BusinessFee(int amount) => amount > 100 ? 1 : 0;
   private static void OnConsoleAction(int balance, int amount) =>
         Console.WriteLine("Could not withdraw "+amount);
   private static void ErrorAction(int balance, int amount) =>
         Console.Error.WriteLine($"ERROR on Withdrawal: {balance}:{amount}");

   static void Main(string[] args)
   {
      var fee1 = new ComputeFee(BusinessFee);
      var act1 = new WithdrawAction(OnConsoleAction);
      var ba1 = new FlexibleBankAccount("a", fee1, act1);
      // var ba1 = new FlexibleBankAccount("a", BusinessFee, OnConsoleAction);
      ba1.Deposit(1000); // 1000
      ba1.Withdraw(50);  // 950
      ba1.Withdraw(150); // 699
      ba1.Withdraw(1000); // 699, + console output
      Console.WriteLine(ba1.Balance); // 699

      var fee2 = new ComputeFee(StandardFee);
      var act2 = new WithdrawAction(OnConsoleAction) + new WithdrawAction(ErrorAction);
      var ba2 = new FlexibleBankAccount("a", fee2, act2);
      ba2.Deposit(1000); // 1000
      ba2.Withdraw(50);  // 949
      ba2.Withdraw(150); // 798
      ba2.Withdraw(1000); // 798, + console/error outputs
      Console.WriteLine(ba2.Balance); // 700
   }
}
```

Anonymous functions

Pros and cons of delegates

- surely reduce boilerplace code, and are hence more convenient than using interfaces
- still there's boilerplate code, since a method is needed somewhere to implement your functional strategy, even if short
- recalling that a functional strategy is just a "function" (in a mathematical/computational interpretation)...

Anonynous functions

- a mechanism to express "in-line" a function to be passed where a delegate is expected
- syntax of expression: `delegate(<parameters list>){<body>}`, to be passed where a compatible delegate is expected
- essentially, it is a function without name and with keyword `delegate` upfront

Using anonymous function

```
static void Main(string[] args)
{
   // use of anonymous function to fill a variable
   WithdrawAction act1 = delegate(int balance, int amount)
   {
      Console.WriteLine("Could not withdraw " + amount);
   };
   // use of anonymous function in-line where needed
   var ba1 = new FlexibleBankAccount("a", delegate(int amount) { return 1; }, act1);
   ba1.Deposit(1000); // 1000
   ba1.Withdraw(50); // 950
   ba1.Withdraw(150); // 699
   ba1.Withdraw(1000); // 699, + console output
   Console.WriteLine(ba1.Balance); // 699

   ComputeFee fee2 = delegate(int amount) { return amount > 100 ? 1 : 0; };
   // cannot use them to chain delegates
   var act2 = new WithdrawAction(OnConsoleAction) + new WithdrawAction(ErrorAction);
   var ba2 = new FlexibleBankAccount("a", fee2 , act2);
   ba2.Deposit(1000); // 1000
   ba2.Withdraw(50); // 949
   ba2.Withdraw(150); // 798
   ba2.Withdraw(1000); // 798, + console/error outputs
   Console.WriteLine(ba2.Balance); // 700

   // Use in libraries...
   List<int> list = new List<int>(new int[]{10,30,20,40,5});
   list.Sort(delegate(int i, int i1) { return i - i1; });
   foreach(var i in list) Console.WriteLine(i); // 5,10,20,30,..
   list.Sort(delegate(int i, int i1) { return i1 - i; });
   foreach(var i in list) Console.WriteLine(i); // 40,30,20,..
}
```

Outline

Lambda expressions

Pros and cons of anonymous functions

- surely further reduce boilerplate code, and are hence more convenient than using method references
- still there's boilerplate code, since the syntax is still rather long: one would more heavility rely on inference and on single-expression body
- lambda-expressions have been invented in 1930 by Alonso Church
- Scala language started using them in OOP languages, and now also Java have them

Lambda-expression

- Complete syntax: `(T1 x1,...,Tn xn) => {<body>}`
- With inference: `(x1,...,xn) => {<body>}`
- With single-expression body: `(T1 x1,...,Tn xn) => <exp>`
- With single argument and inference: `x => {<body>}`
- With unused inputs: `(_,...,_) => {<body>}`
- ...and combinations

⇒ should generally use the shortest version possible

Using lambdas

```
static void Main(string[] args)
{
   // use of anonymous function in-line where needed
   var ba1 = new FlexibleBankAccount(
      "a",
      _ => 1, // lambda of the form x=><exp>
      (_, a) => Console.WriteLine("Could not withdraw " + a));
   ba1.Deposit(1000); // 1000
   ba1.Withdraw(50); // 950
   ba1.Withdraw(150); // 699
   ba1.Withdraw(1000); // 699, + console output
   Console.WriteLine(ba1.Balance); // 699

   WithdrawAction act2 = (_, a) => Console.WriteLine("Could not withdraw " + a);
   act2 +=  (b, a) => Console.Error.WriteLine($"ERROR on Withdrawal: {b}:{a}");
   var ba2 = new FlexibleBankAccount("a", amount => amount > 100 ? 1 : 0 , act2);
   ba2.Deposit(1000); // 1000
   ba2.Withdraw(50); // 949
   ba2.Withdraw(150); // 798
   ba2.Withdraw(1000); // 798, + console/error outputs
   Console.WriteLine(ba2.Balance); // 700

   // Use in libraries...
   List<int> list = new List<int>(new int[]{11,33,22,49,5});
   list.Sort((i,j) => i%10 - j%10);
   foreach(var i in list) Console.WriteLine(i); // 5,10,20,30,..
   list.Sort((i,j) => j-i);
   foreach(var i in list) Console.WriteLine(i); // 40,30,20,..
```

Reusable delegates in libraries

Generic functions in namespace `System`

- `delegate TResult Func<out TResult>();`
- `delegate TResult Func<in T, out TResult(T arg);`
- `delegate TResult Func<in T1, in T2, out TResult(T1 arg1, T2 arg2);`
- `...`
- `delegate bool Predicare<in T>(T obj)`

Generic actions in namespace `System`

- `delegate void Action();`
- `delegate void Action<in T>(T obj);`
- `delegate void Action<in T1, in T2>(T1 obj1, T2 obj2);`
- `...`

Guideline

- define your own delegates only if domain-specific
- otherwise, consider using library delegates

BankAccount with Func and Action

```
public class FlexibleBankAccount : IBankAccount
{
    public int Balance { get; private set; }
    public string Name { get; }

    private Func<int,int> _computeFee;

    private Action<int,int> _action;

    public FlexibleBankAccount(string name, Func<int,int> computeFee, Action<int,int>
   action)
    {
        _computeFee = computeFee;
        _action = action;
        Name = name;
    }

    public void Deposit(int amount) => Balance += amount;

    public void Withdraw(int amount)
    {
        var fee = _computeFee(amount);
        if (Balance < amount + fee) _action(Balance, amount);
        else Balance -= (amount + fee);
    }

    public override string ToString() => $"Name: {Name}, Balance: {Balance}";
}
```

Use BankAccount with Func and Action

```
class UseFlexibleBankAccounts
{
   static void Main(string[] args)
   {
      var ba1 = new FlexibleBankAccount(
         "a",
         _ => 1,
         (_, a) => Console.WriteLine("Could not withdraw " + a));
      // var ba1 = new FlexibleBankAccount("a", BusinessFee, OnConsoleAction);
      ba1.Deposit(1000); // 1000
      ba1.Withdraw(50);  // 950
      ba1.Withdraw(150); // 699
      ba1.Withdraw(1000); // 699, + console output
      Console.WriteLine(ba1.Balance); // 699

      Action<int, int> act2 = (_, a) => Console.WriteLine("Could not withdraw " + a);
      act2 += (b, a) => Console.Error.WriteLine($"ERROR on Withdrawal: {b}:{a}");
      var ba2 = new FlexibleBankAccount("a", a => a > 100 ?1 : 0, act2);
      ba2.Deposit(1000); // 1000
      ba2.Withdraw(50);  // 949
      ba2.Withdraw(150); // 798
      ba2.Withdraw(1000); // 798, + console/error outputs
      Console.WriteLine(ba2.Balance); // 700
   }
}
```

Exercise with Func, Predicate and Action

```
class Helpers {

   public static void ShowAll(IEnumerable<int> elems)
   {
      foreach (var e in elems) Console.WriteLine(e);
   }

   public static IEnumerable<int> GetAllPositive(IEnumerable<int> elems)
   {
      foreach (var e in elems) if (e>0) yield return e;
   }

   public static IEnumerable<int> IncrementAll(IEnumerable<int> elems)
   {
      foreach (var e in elems) yield return e+1;
   }

   public static int SumAll(IEnumerable<int> elems)
   {
      var sum = 0;
      foreach (var e in elems) sum += e;
      return sum;
   }

   public static IEnumerable<int> Iterate(int size)
   {
      for (var i=0; i<size; i++) yield return i;
   }
}
```

Solution for the first three

```
class GeneralizedHelpers
{
   // Generalizing ShowAll
   public static void ForEach<T>(IEnumerable<T> elems, Action<T> action)
   {
      foreach (var e in elems) action(e);
   }

   // Generalizing GetAllPositive
   public static IEnumerable<T> Filter<T>(IEnumerable<T> elems, Predicate<T> pred)
   {
      foreach (var e in elems)
         if (pred(e)) yield return e;
   }

   // Generalizing Increment
   public static IEnumerable<TResult> Map<TResult, T>(IEnumerable<T> elems, Func<T,
  TResult> map)
   {
      foreach (var e in elems) yield return map(e);
   }

   // Generalize the others by yourself!
}
```

Expectations

```
class UseGeneralizedHelpers
{
    static void Main(string[] args)
    {
        var list = new List<string>(new[] {"a", "bb", "ccc", "dddd"});
        ForEach(list, s => Console.Write(s+"; ")); // a; bb; ccc; dddd;
        Console.WriteLine();

        var list2 = Filter(list, s => s.Length < 4);
        ForEach(list2, s => Console.Write(s+"; ")); // a; bb; ccc;
        Console.WriteLine();

        var list3 = Map(list, s => s.Length);
        ForEach(list3, s => Console.Write(s+"; ")); // 1; 2; 3;
        Console.WriteLine();

        // Generalize SumAll to extract from the above list: "abbcccdddd"
        // Use same generalization above to extract from the above list the shortest string
        // Generalize Increment to produce enumeration "a", "aa", "aaa",...
    }
}
```

A preview of standard LINQ library

```
class Program
{
   static void Main(string[] args)
   {
      var archive = new List<Person>(new Person[]
      {
         new Person( "Mario", "Rossi", new DateTime(1990, 1, 18), false),
         new Person( "Gino", "Bianchi", new DateTime(1980, 2, 20), false),
         new Person( "Carla", "Neri", new DateTime(1992, 12, 2), true),
         new Person( "Rosa", "Rosa", new DateTime(1970, 3, 1), false),
         new Person( "Italo", "Casadei", new DateTime(1990, 12, 25), true)
      });

      var marriedPeople = archive.FindAll(p => p.Married).Select(p => p.ToString());
      var toShow = string.Join(" /// ", marriedPeople.Select(p => p.ToString()));
      Console.WriteLine(toShow);

      // count people born later than 1/1/1990
      Predicate<Person> young = p => p.Birth.CompareTo(new DateTime(1990, 1, 1)) > 0;
      var toShow2 = archive.FindAll(p => young(p)).Count;

      Console.WriteLine(toShow2);

      var marriedPeople2 =
         from person in archive
         where person.Married select person.ToString();
      Console.WriteLine(string.Join(" /// ",marriedPeople2.Select(p => p.ToString())));

   }
```

The used Person class

```
class Person
{
    public string Name { get; }
    public string Surname { get; }
    public DateTime Birth { get; }
    public Boolean Married { get; }

    public Person(string name, string surname, DateTime birth, bool married)
    {
        Name = name;
        Surname = surname;
        Birth = birth;
        Married = married;
    }

    public override string ToString()
    {
        return $"Name: {Name}, Surname: {Surname}, Birth: {Birth}, Married: {Married}";
    }
}
```

Outline

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`

→ 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`
  - → .NET enums are value-types
  - → .NET enums are integers

Enums II

Syntax

$$\texttt{enum}\ \langle \textit{Name} \rangle\ [\texttt{:}\ \langle \textit{Integer Type} \rangle]\ \{\ \langle \textit{Constants} \rangle\ \}$$

- 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

Outline

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 overlead 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 $\langle T_3 \rangle$ are equal to the hosting type

Operators Overloading III

Syntax – Binary Operator

```
public static ⟨T3⟩ operator ⟨Symbol⟩(⟨T1⟩ ⟨N1⟩, ⟨T2⟩ ⟨N2⟩)
                    { ⟨Code⟩ }
```

- 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 $\langle Symbol \rangle$

  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 ⟨T2⟩ (⟨T1⟩ ⟨N1⟩) { ⟨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

```
    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);
}
```

Notice that:

- 1 unary operator (i.e. `-`), negating both components of a `Complex`
- 4 binary operators (i.e. `+`, `-`, `/`, `*`) are defined to accept and return `Complex`es
  - 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

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> GetTripledFirstNEvenNumbers1(IEnumerable<int> items, int n)
{
    var list = new List<int>();
    foreach (var item in items)
    {
        if (item % 2 == 0)
        {
            list.Add(item * 3);
        }
    }
    return list.GetRange(0, Math.Min(n, list.Count));
}
```

- this implementation wastes a lot of memory and computational efforts!

The Need for LINQ III

We may then make it more efficient by short-circuiting the algorithm as soon as *N* items have been found:

```
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 IV

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 V

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 VI

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:

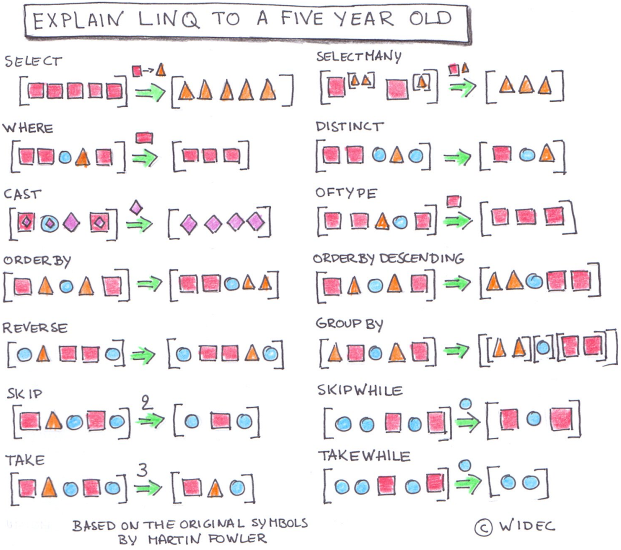
  provisioning — a (possibly long/infinite) stream of data is lazily generated / read from some source

  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));
```