

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. 2021/2022

Outline

- 1 Basic OO in C#
 - C# and .NET
 - Basic OOP
- 2 Some specific C# mechanisms
- 3 Encapsulation, interfaces
- 4 Inheritance
- 5 Generics
- 6 Exceptions and some key C# libraries
- 7 Functional programming in C#

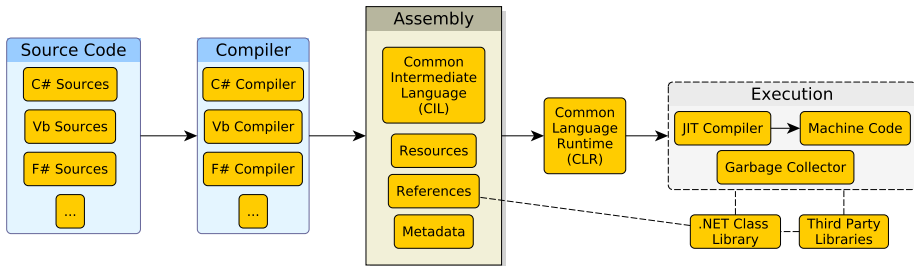
Brief introduction to C#

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 is 10.0, released in 2021 for .NET 6.0
- Mono is a free, open-source compiler and runtime environment
- 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#

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 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 Core 3.1**, to maximise interoperability and to avoid compatibility issues

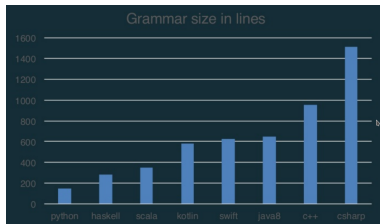
Features of C#

Ingredients

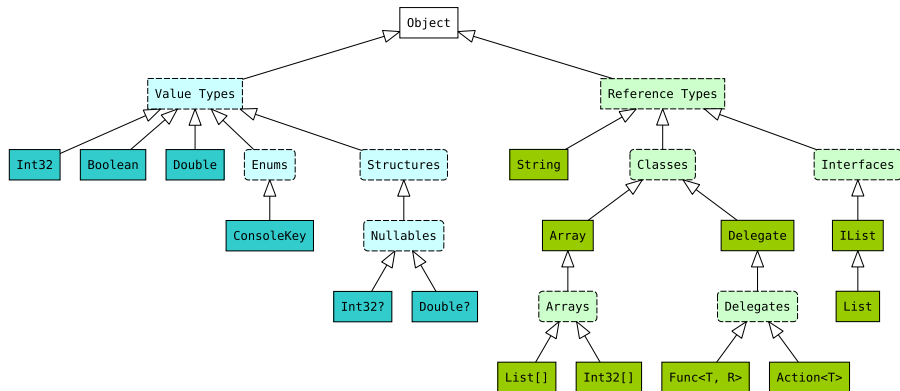
- C-like language: the imperative and structured parts are very similar
- Java-like language: essentially very similar to Java, specially at the beginning
- Static and strong typing: types are checked at run-time, preventing ill-typed operations
- Object-orientation: object by references, automatic garbage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

```
1 String s = new String("aaa");
2 string s2 = "bbb"; // string and String are aliases
3 s2 = "ccc"; // reassignment
4 s2 = s; // s2 will contain a reference to the object of "aaa"
5
6 String ss; // Define name ss, without initialisation
7 // String s3 = ss; // this would not compile!
8 ss = "init"; // now assign ss
9
10 int i = 5 + 2; // int and Int32 are aliases
11 int j = i; // j and i both contain bits representing 7
12
13 Object o = null; // null is a special reference
14 object o2 = o; // object and Object are aliases
15
16 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 $0 \dots (2^8 - 1)$ |
| SByte | sbyte | val | 1 | integers in $-2^7 \dots (2^7 - 1)$ |
| Int16 | short | val | 2 | integers in $-2^{15} \dots (2^{15} - 1)$ |
| UInt16 | ushort | val | 2 | integers in $0 \dots (2^{16} - 1)$ |
| Int32 | int | val | 4 | integers in $-2^{31} \dots (2^{31} - 1)$ |
| UInt32 | uint | val | 4 | integers in $0 \dots (2^{32} - 1)$ |
| Int64 | long | val | 8 | integers in $-2^{63} \dots (2^{63} - 1)$ |
| UInt64 | ulong | val | 8 | integers in $0 \dots (2^{64} - 1)$ |
| Float | float | val | 4 | abs in $1.5 \times 10^{-45} \dots 3.4 \times 10^{38}$ |
| Double | double | val | 8 | abs in $5.0 \times 10^{-324} \dots 1.7 \times 10^{308}$ |
| Decimal | decimal | val | 16 | abs in $1.0 \times 10^{-28} \dots 7.9228 \times 10^{28}$ |
| 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

- 1 Basic OO in C#
 - C# and .NET
 - Basic OOP
- 2 Some specific C# mechanisms
- 3 Encapsulation, interfaces
- 4 Inheritance
- 5 Generics
- 6 Exceptions and some key C# libraries
- 7 Functional programming in C#

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

```
1 namespace Point3D
2 {
3     public class UsePoint3D
4     {
5         public static void Main()
6         {
7             var p = new Point3D(10.0, 20.0, 30.0);    // instantiation with args
8             // Point3D q = new Point3D(); // constructor with 0-args no more possible
9             Console.WriteLine(p.GetSquareModulus());
10        }
11    }
12    public class Point3D
13    {
14        private double _x;
15        private double _y;
16        private double _z;
17
18        public Point3D(double x, double y, double z)
19        {
20            _x = x;
21            _y = y;
22            _z = z;
23        }
24
25        public double GetSquareModulus()
26        {
27            return _x * _x + _y * _y + _z * _z;
28        }
29    }
30 }
31 }
```

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

Typical structure of an executable project

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 namespaces
- a namespace is a “module” giving a context to the class

Project

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

A class Person

```
1 public class UsePerson
2 {
3     static void Main(string[] args)
4     {
5         var p1 = new Person("John", 1980);
6         var p2 = new Person("Michael", 1973);
7         p2.GotMarried();
8         Console.WriteLine(p1.ShowAsString()); // John 1980 False
9         Console.WriteLine(p2.ShowAsString()); // Michael 1973 True
10    }
11 }
12 public class Person
13 {
14     private string _name; // string is an alias for String...
15     private int _birthYear;
16     private bool _married = false;
17     public Person(string name, int birthYear)
18     {
19         _name = name;
20         _birthYear = birthYear;
21     }
22     public void GotMarried()
23     {
24         _married = true;
25     }
26     public string ShowAsString()
27     {
28         return _name + " " + _birthYear + " " + _married;
29     }
30 }
```

Constructors chaining

```
1 public class UsePerson
2 {
3     public static void Main(string[] args)
4     {
5         Console.WriteLine(new Person("Bill").ShowAsString());
6         Console.WriteLine(new Person("Michael", 1973, true).ShowAsString());
7     }
8 }
9 public class Person
10 {
11     private string _name;
12     private int _birthYear;
13     private bool _married;
14
15     public Person(string name, int birthYear, bool married)
16     {
17         Console.WriteLine("called first constructor");
18         _name = name;
19         _birthYear = birthYear;
20         _married = married;
21     }
22
23     public Person(string name) : this(name, 1900, false)
24     {
25         Console.WriteLine("called second constructor... chaining to the first");
26     }
27
28     public string ShowAsString()
29     {
30         return _name + " " + _birthYear + " " + _married;
31     }
32 }
33
```


Playing with libraries (namespace System)

```
1 using System;
2 // https://docs.microsoft.com/en-us/dotnet/api/system?view=net-5.0
3
4 namespace PlayWithLibraries
5 {
6     class Program
7     {
8         public static void Main()
9         {
10             Console.WriteLine("The result of 8+2 is " + 10);
11             Console.WriteLine("The result of 8+2 is " + (8+2));
12             Console.WriteLine("The result of {0}+{1} is {2}",8,2,10);
13             var res = 8 + 2;
14             Console.WriteLine($"The result of 8+2 is {res}");
15             Console.WriteLine($"The result of 8+2 is {2+8}");
16
17             var date1 = new DateTime(2008, 3, 1, 7, 0, 0);
18             Console.WriteLine(date1.ToString()); // 3/1/2008 7:00:00 AM
19             var date2 = date1.AddMinutes(30);
20             Console.WriteLine(date2.ToString()); // 3/1/2008 7:30:00 AM
21
22             var rand = new Random();
23             Console.WriteLine("{0},{1}", rand.Next(10), rand.Next(10));
24             Console.WriteLine(rand.NextDouble()); // in [0..1]
25
26             Console.Write("Input a number here: ");
27             String str = Console.ReadLine(); // read from console
28             int number = Int32.Parse(str);    // convert to int (if possible)
29             Console.WriteLine(number);
30         }
31     }
32 }
```

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 setter 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 setters

Person with Getters and Setters

```
1 public class Person
2 {
3     private string _name; // string is an alias for String...
4     private int _birthYear;
5     private bool _married = false;
6     public Person(string name, int birthYear)
7     {
8         _name = name;
9         _birthYear = birthYear;
10    }
11    public string GetName()
12    {
13        return _name;
14    }
15    public int GetBirthYear()
16    {
17        return _birthYear;
18    }
19    public bool GetMarried()
20    {
21        return _married;
22    }
23    public void SetMarried(bool married)
24    {
25        _married = married;
26    }
27    public string GetStringRepresentation()
28    {
29        return _name + " " + _birthYear;
30    }
31 }
```

Client code for Person

```
1 public static void Main(string[] args)
2 {
3     var p1 = new Person("John", 1980);
4     Console.WriteLine(p1.GetName());
5     Console.WriteLine(p1.GetBirthYear());
6     Console.WriteLine(p1.GetMarried());
7     Console.WriteLine(p1.GetStringRepresentation());
8     p1.SetMarried(true);
9     Console.WriteLine(p1.GetMarried());
10 }
```

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

```
1 public class Person
2 {
3     private readonly string _name;    // readonly field, cannot be changed
4     private readonly int _birthYear;  // readonly field, cannot be changed
5     private bool _married = false;
6
7     public Person(string name, int birthYear)
8     {
9         _name = name;
10        _birthYear = birthYear;
11    }
12
13    public string GetName() => _name;
14
15    public int GetBirthYear() => _birthYear;
16
17    public bool GetMarried() => _married;
18
19    public void SetMarried(bool married) => _married = married;
20
21    public string GetStringRepresentation() => _name + " " + _birthYear;
22 }
```

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

Properties

Improving over Get/Set accessors

- C# introduces a programming construct for properties
- a property is directly perceived by the client as a sort of field (starting with uppercase)
- internally to a class, a property is actually a getter and/or setter with special syntax

Notation

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

The special case of auto-implemented properties

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

Person with Properties

```
1 public class UsePerson
2 {
3     public static void Main(string[] args)
4     {
5         var p1 = new Person("John", 1980);
6         Console.WriteLine(p1.Name);
7         Console.WriteLine(p1.BirthYear);
8         Console.WriteLine(p1.Married);
9         Console.WriteLine(p1.StringRepresentation);
10        p1.Married = true;
11        Console.WriteLine(p1.Married);
12    }
13 }
14 public class Person
15 {
16     public string Name { get; } // auto-implemented readonly property
17     public int BirthYear { get; } // auto-implemented readonly property
18     public bool Married { get; set; } // auto-implemented read/write property
19
20     public Person(string name, int birthYear)
21     {
22         Name = name;
23         BirthYear = birthYear;
24         Married = false;
25     }
26
27     // expression-bodied property
28     public string StringRepresentation => Name + " " + BirthYear;
29 }
```

Person with Properties: playing with properties

```
1 public class Person
2 {
3     public string Name { get; } // auto-implemented readonly property
4     public int BirthYear { get; } // auto-implemented readonly property
5
6     private bool _married;
7
8     public bool Married
9     {
10         get => _married;
11         set {
12             if (_married && !value)
13             {
14                 Console.WriteLine("can't unmarry!!");
15             }
16             else
17             {
18                 _married = value;
19             }
20         }
21     }
22
23     public Person(string name, int birthYear)
24     {
25         Name = name;
26         BirthYear = birthYear;
27         Married = false;
28     }
29
30     public string StringRepresentation => Name + " " + BirthYear;
31 }
```

Playing with properties: client code

```
1 public static void Main(string[] args)
2 {
3     var p1 = new Person("John", 1980);
4     Console.WriteLine(p1.Name);
5     Console.WriteLine(p1.BirthYear);
6     Console.WriteLine(p1.Married); // false
7     p1.Married = true; // can marry
8     Console.WriteLine(p1.Married); // true
9     p1.Married = false; // can't unmarry, message emitted
10    Console.WriteLine(p1.Married); // still true
11    Console.WriteLine(p1.StringRepresentation);
12 }
```

Outline

- 1 Basic OO in C#
- 2 Some specific C# mechanisms
 - Arrays
 - Nullables
- 3 Encapsulation, interfaces
- 4 Inheritance
- 5 Generics
- 6 Exceptions and some key C# libraries
- 7 Functional programming in C#

.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, \dots, 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. $D_1 \times \dots \times D_N$)
 - 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[index1, ..., indexN']`

Array Types Instantiation I

Constructors for N -dimensional Arrays of T

$T[, \dots]$ $\langle Var\ Name \rangle = \text{new } T[D_1, D_2, \dots];$

- 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

$T[, \dots]$ $\langle Var\ Name \rangle = \text{new } T[, \dots] \{ \dots \{ \langle Item_1 \rangle, \langle Item_2 \rangle, \dots \} \dots \};$

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

Array Types Instantiation II

```
1  Int32[] aLinearArrayOf10Ints = new Int32[10]; // all initialised to 0
2  Int32[] aLinearArrayOf4Ints = new Int32[] {1, 2, 3, 4};
3  Int32[] anotherLinearArrayOf4Ints = new [] {1, 2, 3, 4};
4  Int32[] yetAnotherLinearArrayOf4Ints = {1, 2, 3, 4};
5  Int32[,] aMatrixOf12Ints = new Int32[4,3]; // all initialised to 0
6  Int32[,] aMatrixOf6Ints = new Int32[,] {{1, 2, 3}, {4, 5, 6}};
7  Int32[,] anotherMatrixOf6Ints = {{1, 2}, {3, 4}, {5, 6}};
8  Int32[,,] a3DArrayOf8Ints = new Int32[2,2,2]; // all initialised to 0
9  Int32[,,] another3DArrayOf8Ints = {{{1, 2}, {3, 4}}, {{5, 6}, {7, 8}}};
10 String[] aLinearArrayOf10Strings = new string[10]; // all initialised to null
11 String[] aLinearArrayOf3Strings = new string[] {"a1", "b2", "c3"};
12 String[][] anArrayOf10ArraysOfStrings = new string[10][]; // all sub-arrays are to null
13 String[][] anArrayOf3ArraysOf2Strings = new string[][]
14 {
15     new[] {"a", "b"}, new[] {"c", "d"}, new[] {"e", "f"}
16 };
```


Accessing Arrays Items I

```
1 public static void BubbleSort(Int32[] array)
2 {
3     for (Int32 i = 0; i < array.Length; i++)
4     {
5         for (Int32 j = i - 1; j >= 0; j--)
6         {
7             if (array[j + 1] < array[j])
8             {
9                 Int32 temp = array[j];    // read array item
10                array[j] = array[j + 1];  // read and set array item
11                array[j + 1] = temp;       // set array item
12            }
13        }
14    }
15 }
```

```
1 public static void FillMatrixRandomly(Int32[,] matrix, Random random)
2 {
3     for (Int32 i = 0; i < matrix.GetLength(0); i++)
4     {
5         for (Int32 j = 0; i < matrix.GetLength(1); i++)
6         {
7             matrix[i, j] = random.Next();
8         }
9     }
10 }
```

Outline

- 1 Basic OO in C#
- 2 Some specific C# mechanisms
 - Arrays
 - **Nullable**
- 3 Encapsulation, interfaces
- 4 Inheritance
- 5 Generics
- 6 Exceptions and some key C# libraries
- 7 Functional programming in C#

.NET Nullables

Nullable types definition

- let T be a **value** type of any sort, then $T?$ denotes the **nullable** T type
- A nullable type $T?$ can be defined as $T \cup \{ \text{null} \}$.
- 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>)

Nullable 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

```
1  Int32? aNullableInt = null;
2  Int32? anotherNullableInt = 1 + aNullableInt; // null
3
4  Int32? aNullableInt2 = 5;
5  Int32? anotherNullableInt2 = 1 + aNullableInt2; // 6
6
7  Int32 i = anotherNullableInt ?? 0; // 0
8  Int32 i2 = anotherNullableInt2 ?? 0; // 6
9
10 String aString = aNullableInt?.ToString(); // null
11 String aString2 = aNullableInt2?.ToString(); // "6"
```

Outline

- 1 Basic OO in C#
- 2 Some specific C# mechanisms
- 3 **Encapsulation, interfaces**
 - **Encapsulation, and properties**
 - Interfaces
- 4 Inheritance
- 5 Generics
- 6 Exceptions and some key C# libraries
- 7 Functional programming in C#

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

```
1 public class Counter
2 {
3     // the field is made inaccessible
4     private int _countValue;
5
6     // it is the constructor that initialises fields
7     public Counter()
8     {
9         _countValue = 0;
10    }
11
12    // the only method to observe state
13    public int GetValue()
14    {
15        return _countValue;
16    }
17
18    // the only method to modify state
19    public void Increment()
20    {
21        _countValue++;
22    }
23 }
```

A basic case: usage of class Counter

```
1 public class UseCounter
2 {
3     public static void Main(string[] args)
4     {
5         var counter = new Counter();
6         Console.WriteLine(counter.GetValue()); // 0
7         counter.Increment();
8         counter.Increment();
9         counter.Increment();
10        Console.WriteLine(counter.GetValue()); // 3
11    }
12 }
```


Encapsulation is preserved by properties!

```
1 public class UseCounter
2 {
3     public static void Main(string[] args)
4     {
5         var counter = new Counter();
6         Console.WriteLine(counter.Value); // 0
7         counter.Increment();
8         counter.Increment();
9         Console.WriteLine(counter.Value); // 2
10    }
11 }
12
13 public class Counter
14 {
15     // an implicitly define field, not modifiable from outside
16     // still a well encapsulated solution: one can change implementation
17     // of get and set below if needed
18     public int Value { get; private set; } = 0;
19
20     public void Increment() => Value++;
21 }
```

A transparent modification to the Counter implementation

```
1 public class Counter
2 {
3     // a modified implementation, without required changes in clients
4     private int _value;
5     private const int MaxValue = 100; // const: essentially a MACRO
6
7     // note the property Value ensures MaxValue is never overcome
8     public int Value
9     {
10         get => _value;
11         private set => _value = value <= MaxValue ? value : _value;
12     }
13
14     public Counter() => Value = 0;
15
16     public void Increment() => Value++;
17 }
18
```

A final bit on properties: object initializers

```
1 public class Student
2 {
3     public int StudentID { get; set; }
4     public string StudentName { get; set; }
5     public int Age { get; set; }
6     public string Address { get; set; }
7 }
8
9 class UseStudent
10 {
11     static void Main(string[] args)
12     {
13         // need to have get-set properties
14         var std = new Student() { // object initializer
15             StudentID = 1,
16             StudentName = "Bill",
17             Age = 20,
18             Address = "New York"
19         };
20
21         // Essentially equivalent to...
22         var std2 = new Student(); // default construct is needed
23         std2.StudentID = 1;
24         std2.StudentName = "Bill";
25         std2.Age = 20;
26         std2.Address = "New York";
27     }
28 }
```

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

- 1 Basic OO in C#
- 2 Some specific C# mechanisms
- 3 Encapsulation, interfaces**
 - Encapsulation, and properties
 - Interfaces**
- 4 Inheritance
- 5 Generics
- 6 Exceptions and some key C# libraries
- 7 Functional programming in C#

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 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.

```
1 public interface IDevice
2 {
3     void SwitchOn();
4
5     void SwitchOff();
6
7     bool IsOn { get; }
8 }
```

Two lamp implementations of Device

```
1 public class Lamp : IDevice
2 {
3     public void SwitchOn() => IsOn = true;
4
5     public void SwitchOff() => IsOn = false;
6
7     public bool IsOn { get; private set; }
8
9     // additional methods can be added if needed
10 }
11
12 // a lamp with erratic switches
13 public class ErraticLamp : IDevice
14 {
15     private bool _on = false; // here, want to use a field
16     private readonly Random _random = new Random();
17
18     public void SwitchOn() => _on = _random.NextDouble() < 0.95;
19
20     public void SwitchOff() => _on = _random.NextDouble() < 0.05;
21
22     public bool IsOn { get => _on; }
23
24     // additional methods can be added if needed
25 }
```


Multiple implementation

Multiple implementation

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

- A class C implements I1 and I2 and I3
- The class C must 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

- 1 Basic OO in C#
- 2 Some specific C# mechanisms
- 3 Encapsulation, interfaces
- 4 Inheritance**
 - **Class extension**
 - Runtime types
- 5 Generics
- 6 Exceptions and some key C# libraries
- 7 Functional programming in C#

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

```
1 class UseCounter
2 {
3     public static void Main(string[] args)
4     {
5         var counter = new Counter(5);
6         Console.WriteLine(counter.Value); // 5
7         counter.Increment();
8         counter.Increment();
9         counter.Increment();
10        Console.WriteLine(counter.Value); // 8
11    }
12 }
13
14 public class Counter
15 {
16     public int Value { get; private set; }
17
18     public Counter(int initialValue) => Value = initialValue;
19
20     public void Increment() => Value++;
21 }
```

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

```
1 // Reuse with inheritance: a terser solution!
2 public class MultiCounter : Counter
3 {
4     public MultiCounter(int initialValue) : base(initialValue)
5     {
6     }
7
8     public void MultiIncrement(int n)
9     {
10         for (var i = 0; i < n; i++) Increment();
11     }
12 }
```

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

```
1 class UseBiCounter
2 {
3     public static void Main(string[] args)
4     {
5         var counter = new BiCounter(5);
6         Console.WriteLine(counter.Value); // 5
7         counter.Increment();
8         counter.Decrement();
9         counter.Decrement();
10        Console.WriteLine(counter.Value); // 4
11    }
12 }
13
14 public class ExtendibleCounter
15 {
16     public int Value { get; protected set; }
17
18     public ExtendibleCounter(int initialValue) => Value = initialValue;
19
20     public void Increment() => Value++;
21 }
22
23 public class BiCounter : ExtendibleCounter
24 {
25     public BiCounter(int initialValue) : base(initialValue)
26     {
27     }
28
29     public void Decrement() => Value--;
30 }
```

Analogous solution with fields

```
1 public class ExtendibleCounter
2 {
3     private int _value;
4
5     public ExtendibleCounter(int initialValue) => _value = initialValue;
6
7     public int GetValue() => _value;
8
9     protected void SetValue(int value) => _value = value;
10
11    public void Increment() => _value++;
12 }
13
14 public class BiCounter : ExtendibleCounter
15 {
16     public BiCounter(int initialValue) : base(initialValue)
17     {
18     }
19
20     public void Decrement() => SetValue(GetValue()-1);
21 }
```


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 in the superclass that is not **virtual**, by the modifier **new** – but this mechanism 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

```
1 public class UseLimitCounter
2 {
3     public static void Main(string[] args)
4     {
5         var limitCounter = new LimitCounter(3);
6         Console.WriteLine(limitCounter.Value); // 0
7         limitCounter.Increment();
8         limitCounter.Increment();
9         Console.WriteLine(limitCounter.Value); // 2
10        limitCounter.Increment();
11        limitCounter.Increment();
12        Console.WriteLine(limitCounter.Value); // 3
13    }
14 }
```

Class LimitCounter

```
1 public class Counter
2 {
3     public int Value { get; protected set; }
4
5     public Counter(int initialValue) => Value = initialValue;
6
7     public virtual void Increment() => Value++;
8 }
9
10 public sealed class LimitCounter : Counter
11 {
12     private readonly int _limit;
13
14     public LimitCounter(int limit) : base(0) => _limit = limit;
15
16     public bool IsOver() => Value >= _limit;
17
18     public override void Increment()
19     {
20         if (!IsOver()) base.Increment();
21     }
22 }
```

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 overridden to provide a canonical string representation of an object
- `Console.WriteLine` uses it if you try to write an object

Outline

- 1 Basic OO in C#
- 2 Some specific C# mechanisms
- 3 Encapsulation, interfaces
- 4 Inheritance**
 - Class extension
 - Runtime types**
- 5 Generics
- 6 Exceptions and some key C# libraries
- 7 Functional programming in C#

Polymorphism with object

```
1 class Program
2 {
3     static void Main(string[] args)
4     {
5         var objects = new object[5];
6         objects[0] = "hello!";
7         objects[1] = new object();
8         objects[2] = 10; // boxing to a System.Int32
9         objects[3] = DateTime.Now;
10        objects[4] = new int[] {10, 20, 30};
11
12        PrintAll(objects);
13    }
14
15    private static void PrintAll(object[] objects)
16    {
17        foreach (var obj in objects)
18        {
19            // Polymorphic call to ToString
20            Console.WriteLine(obj.ToString());
21            // Giving a representation of obj's type
22            Console.WriteLine(obj.GetType());
23        }
24    }
25 }
```

UsePerson and Person

```
1 public class UsePerson
2 {
3     public static void Main(string[] args)
4     {
5         var people = new Person[]
6         {
7             new Teacher("Mirko", 521, new string[] {"oop", "softeng"}),
8             new Student("Mario", 1001, 2019),
9             new Student("Carla", 1002, 2020),
10            new Teacher("Giovanni", 522, new string[] {"sistdist"})
11        };
12        // note the polymorphic call to virtual method ToString...
13        foreach (var person in people)
14        {
15            Console.WriteLine($"{person.Name}; {person.ToString()}");
16        }
17    }
18 }
19 public class Person
20 {
21     public string Name { get; }
22
23     public int Id { get; }
24
25     public Person(string name, int id)
26     {
27         Name = name;
28         Id = id;
29     }
30 }
```


Specialisations of Person

```
1 public class Student : Person
2 {
3     public int MatriculationYear { get; }
4
5     public Student(string name, int id, int matriculationYear) : base(name, id)
6     {
7         MatriculationYear = matriculationYear;
8     }
9
10    public override string ToString() => $"S[{Name}, {Id}, m:{MatriculationYear}]";
11 }
12
13 public class Teacher : Person
14 {
15     public string[] Courses { get; }
16
17     public Teacher(string name, int id, string[] courses) : base(name, id)
18     {
19         Courses = courses;
20     }
21
22    public override string ToString() => $"T[{Name}, {Id}, c:{string.Join(", ", Courses)}]";
23 }
```

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

```
1 public class UsePerson
2 {
3     public static void Main(string[] args)
4     {
5         var people = new Person[]
6         {
7             new Teacher("Mirko", 521, new string[] {"oop", "softeng"}),
8             new Student("Mario", 1001, 2019),
9             new Student("Carla", 1002, 2020),
10            new Teacher("Giovanni", 522, new string[] {"sistdist"})
11        };
12        // printing only teachers
13        foreach (var person in people)
14        {
15            if (person is Teacher) // check for run-time type
16            {
17                Console.WriteLine($"{person.Name}; {person.ToString()}");
18            }
19            else
20            {
21                Student student = person as Student; // type conversion, same reference!
22                // Teacher teacher = person as Teacher; // this would yield null!
23                Console.WriteLine($"{student.Name}; {student.MatriculationYear}");
24            }
25        }
26    }
27 }
```

Outline

- 1 Basic OO in C#
- 2 Some specific C# mechanisms
- 3 Encapsulation, interfaces
- 4 Inheritance
- 5 Generics**
 - **Generic classes**
 - Generic methods
 - Generic interfaces
 - Iterators and collections
 - Constrained polymorphism
 - Variance
- 6 Exceptions and some key C# libraries
- 7 Functional programming in C#

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

```
1 public class UseIntVectors
2 {
3     public static void Main(string[] args)
4     {
5         var fibonacci = new IntVector();
6         Console.WriteLine(fibonacci); // []
7         fibonacci.AddElement(1);
8         fibonacci.AddElement(5);
9         Console.WriteLine(fibonacci); // [1,5]
10        fibonacci.SetElementAt(1,1);
11        Console.WriteLine(fibonacci); // [1,1]
12        for (var i=2; i< 10; i++)
13            fibonacci.AddElement(fibonacci.GetElementAt(i-1)+
14                                fibonacci.GetElementAt(i-2));
15        Console.WriteLine(fibonacci); // [1,1,2,3,5,8,13,21,34,55]
16    }
17 }
```

Preliminary IntVector

```
1 public class IntVector
2 {
3     private const int InitialCapacity = 10;
4     private const int MultiplicationFactor = 2;
5     private int[] _elements = new int[InitialCapacity];
6     public int Size { get; private set; } = 0;
7
8     public void AddElement(int element)
9     {
10         if (Size == _elements.Length) Expand();
11         _elements[Size++] = element;
12     }
13
14     private void Expand()
15     {
16         var old = _elements;
17         _elements = new int[old.Length * MultiplicationFactor];
18         Array.Copy(old, _elements, Size);
19     }
20
21     public int GetElementAt(int position) => _elements[position];
22
23     public void SetElementAt(int position, int element) => _elements[position] = element;
24
25     public override string ToString()
26     {
27         var s = "[";
28         for (var i = 0; i < Size; i++) s += _elements[i] + (i < Size - 1 ? ", " : "");
29         return s + "]";
30     }
31 }
```

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)

```
1 public class IntVector
2 {
3     private const int InitialCapacity = 10;
4     private const int MultiplicationFactor = 2;
5     private int[] _elements = new int[InitialCapacity];
6     public int Size { get; private set; } = 0;
7
8     public int this[int i]
9     {
10         get => _elements[i];
11         set => _elements[i] = value;
12     }
13
14     public void AddElement(int element)
15     {
16         if (Size == _elements.Length) Expand();
17         _elements[Size++] = element;
18     }
19
20     private void Expand()
21     {
22         var old = _elements;
23         _elements = new int[old.Length * MultiplicationFactor];
24         Array.Copy(old, _elements, Size);
25     }
26
27     public override string ToString()
28     {
29         var s = "[";
30         for (var i = 0; i < Size; i++) s += _elements[i] + (i < Size - 1 ? ", " : "");
31         return s + "]";
32     }
33 }
```

UseIntVector

```
1 public class UseIntVectors
2 {
3     public static void Main(string[] args)
4     {
5         var fibonacci = new IntVector();
6         fibonacci.AddElement(1);
7         fibonacci.AddElement(5);
8         fibonacci[1] = 1;
9         for (var i=2; i< 10; i++)
10             fibonacci.AddElement(fibonacci[i-1]+fibonacci[i-2]);
11         Console.WriteLine(fibonacci); // [1,1,2,3,5,8,13,21,34,55]
12     }
13 }
```

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

```
1 public class UseObjectVectors
2 {
3     public static void Main(string[] args)
4     {
5         // used to store strings... needs annoying "as" operator to retrieve
6         var strings = new ObjectVector();
7         strings.AddElement("hello");
8         strings.AddElement("world!");
9         string first = strings[0] as string; // could be null
10        string second = strings[1] as string;
11        Console.WriteLine(first?.Substring(2)); // "llo"
12        // cannot guarantee it contains just strings
13        strings.AddElement(DateTime.Now);
14
15
16        var fibonacci = new ObjectVector();
17        fibonacci.AddElement(1); // actually adding a int?
18        fibonacci.AddElement(5);
19        fibonacci[1] = 1;
20        for (var i = 2; i < 10; i++)
21        {
22            // working with int? is not very optimal, it's "viscose"
23            var next = (fibonacci[i - 1] as int?) + (fibonacci[i - 2] as int?);
24            if (next != null) fibonacci.AddElement(next);
25        }
26        Console.WriteLine(fibonacci); // [1,1,2,3,5,8,13,21,34,55]
27    }
28 }
```

ObjectVector

```
1 public class ObjectVector
2 {
3     private const int InitialCapacity = 10;
4     private const int MultiplicationFactor = 2;
5     private object[] _elements = new object[InitialCapacity];
6     public int Size { get; private set; } = 0;
7
8     public object this[int i]
9     {
10         get => _elements[i];
11         set => _elements[i] = value;
12     }
13
14     public void AddElement(object element)
15     {
16         if (Size == _elements.Length) Expand();
17         _elements[Size++] = element;
18     }
19
20     private void Expand()
21     {
22         var old = _elements;
23         _elements = new object[old.Length * MultiplicationFactor];
24         Array.Copy(old, _elements, Size);
25     }
26
27     public override string ToString()
28     {
29         var s = "[";
30         for (var i = 0; i < Size; i++) s += _elements[i] + (i < Size - 1 ? ", " : "");
31         return s + "]";
32     }
33 }
```

Another example of collection: ObjectList

```
1 public class ObjectList
2 {
3     public object Head { get; }
4     public ObjectList Tail { get; }
5
6     public ObjectList(object head, ObjectList tail)
7     {
8         Head = head;
9         Tail = tail;
10    }
11
12    public object this[int i] => i == 0 ? Head : Tail[i - 1];
13
14    public int Length => 1 + (Tail?.Length ?? 0);
15
16    public override string ToString()
17    {
18        var s = "[";
19        for (var i = 0; i < Length; i++)
20            s += this[i] + (i < Length - 1 ? ", " : "");
21        return s + "]";
22    }
23 }
```

UseObjectList

```
1 class UseObjetList
2 {
3     static void Main(string[] args)
4     {
5         var strings =
6             new ObjectList("Hello!",
7                             new ObjectList("my",
8                                             new ObjectList("World", null)));
9         Console.WriteLine(strings.Head + " " + strings.Tail);
10        Console.WriteLine(strings); // [Hello!,my,World]
11
12        // usual problems with the "monomorphic collection" mechanisms
13        var numbers =
14            new ObjectList(10, new ObjectList(20, null));
15        int? sum = (numbers.Head as int?) + (numbers.Tail.Head as int?);
16        Console.WriteLine(sum);
17    }
18 }
```

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

```
1 public class GenericVector<T>
2 {
3     private const int InitialCapacity = 10;
4     private const int MultiplicationFactor = 2;
5     private T[] _elements = new T[InitialCapacity];
6     public int Size { get; private set; } = 0;
7
8     public T this[int i]
9     {
10         get => _elements[i];
11         set => _elements[i] = value;
12     }
13
14     public void AddElement(T element)
15     {
16         if (Size == _elements.Length) Expand();
17         _elements[Size++] = element;
18     }
19
20     private void Expand()
21     {
22         var old = _elements;
23         _elements = new T[old.Length * MultiplicationFactor];
24         Array.Copy(old, _elements, Size);
25     }
26
27     public override string ToString()
28     {
29         var s = "[";
30         for (var i = 0; i < Size; i++) s += _elements[i] + (i < Size - 1 ? "," : "");
31         return s + "]";
32     }
33 }
```

Using generic vector

```
1 public class UseVectors
2 {
3     public static void Main(string[] args)
4     {
5         // we have an explicit type used to store strings...
6         // no annoying "as" operator and nulls to deal with
7         GenericVector<string> strings = new GenericVector<string>();
8         strings.AddElement("hello");
9         strings.AddElement("world!");
10        string first = strings[0];
11        string second = strings[1];
12        Console.WriteLine(strings[0].Substring(2)); // "llo"
13        // strings.AddElement(DateTime.Now); // would not compile!
14
15
16        var fibonacci = new GenericVector<int>();
17        fibonacci.AddElement(1); // actually adding a int, not an int?
18        fibonacci.AddElement(5);
19        fibonacci[1] = 1;
20        for (var i = 2; i < 10; i++)
21        {
22            fibonacci.AddElement(fibonacci[i-2]+fibonacci[i-1]);
23        }
24        Console.WriteLine(fibonacci); // [1,1,2,3,5,8,13,21,34,55]
25
26    }
27 }
```

Outline

- 1 Basic OO in C#
- 2 Some specific C# mechanisms
- 3 Encapsulation, interfaces
- 4 Inheritance
- 5 Generics**
 - Generic classes
 - Generic methods**
 - Generic interfaces
 - Iterators and collections
 - Constrained polymorphism
 - Variance
- 6 Exceptions and some key C# libraries
- 7 Functional programming in C#

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 different instantiations

Syntax is ad-hoc

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

Generic static methods

```
1 class Program
2 {
3     // A method generified in the type of element to create the vector
4     public static GenericVector<T> CreateAndFill<T>(int size, T elem)
5     {
6         var v = new GenericVector<T>();
7         for (var i=0; i<size; i++) v.AddElement(elem);
8         return v;
9     }
10    // A method generified in the type of elements of the vector to show
11    public static void ShowAll<T>(GenericVector<T> vector)
12    {
13        for (int i=0; i<vector.Size;i++) Console.Write(vector[i]+" ");
14        Console.WriteLine();
15    }
16    static void Main(string[] args)
17    {
18        // version with explicit indication of type
19        GenericVector<string> vs = CreateAndFill<string>(5, "a");
20        // version with type inference
21        GenericVector<int> vi = CreateAndFill(5, 10);
22
23        // version with explicit indication of type
24        ShowAll<int>(vi);
25        // version with type inference
26        ShowAll(vs);
27    }
28 }
```

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

```
1 public class Pair<TA, TB>
2 {
3     public TA First { get; }
4     public TB Second { get; }
5
6     public Pair(TA a, TB b)
7     {
8         First = a;
9         Second = b;
10    }
11
12    public Pair<TA, TC> ChangeSecond<TC>(TC c) => new Pair<TA, TC>(First, c);
13
14    public Pair<TC, TB> ChangeFirst<TC>(TC c) => new Pair<TC, TB>(c, Second);
15
16    public override string ToString() => $"({First}:{Second})";
17 }
```

Using pairs

```
1 class UsePairs
2 {
3     static void Main(string[] args)
4     {
5         var pair = new Pair<string, int>("ciao", 2);
6         Console.WriteLine(pair);
7         Console.WriteLine(pair.First.Substring(pair.Second));
8
9         var archive = new GenericVector<Pair<int, string>>();
10        archive.AddElement(new Pair<int, string>(1001, "mirko"));
11        archive.AddElement(new Pair<int, string>(800, "carla"));
12        archive.AddElement(new Pair<int, string>(1003, "mario"));
13        Console.WriteLine(archive);
14
15        // name of people with id>1000
16        var searchResults = new GenericVector<string>();
17        for (var i = 0; i < archive.Size; i++)
18        {
19            if (archive[i].First > 1000)
20                searchResults.AddElement(archive[i].Second);
21        }
22        Console.WriteLine(searchResults);
23
24        // showcasing generic methods
25        Console.WriteLine(pair.ChangeFirst(10)); // a Pair<int,int>
26        Console.WriteLine(pair.ChangeSecond("10")); // a Pair<string,string>
27    }
28 }
```


Outline

- 1 Basic OO in C#
- 2 Some specific C# mechanisms
- 3 Encapsulation, interfaces
- 4 Inheritance
- 5 Generics**
 - Generic classes
 - Generic methods
 - Generic interfaces**
 - Iterators and collections
 - Constrained polymorphism
 - Variance
- 6 Exceptions and some key C# libraries
- 7 Functional programming in C#

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>

```
1 public interface IGenericVector<T>
2 {
3     int Size { get; }
4
5     T this[int i] { get; set; }
6
7     void AddElement(T element);
8 }
```

Implementing IGenericVector

```
1 public class GenericVector<T> : IGenericVector<T>
2 {
3     private const int InitialCapacity = 10;
4     private const int MultiplicationFactor = 2;
5     private T[] _elements = new T[InitialCapacity];
6     public int Size { get; private set; } = 0;
7
8     public T this[int i]
9     {
10         get => _elements[i];
11         set => _elements[i] = value;
12     }
13
14     public void AddElement(T element)
15     {
16         if (Size == _elements.Length) Expand();
17         _elements[Size++] = element;
18     }
19
20     private void Expand()
21     {
22         var old = _elements;
23         _elements = new T[old.Length * MultiplicationFactor];
24         Array.Copy(old, _elements, Size);
25     }
26
27     public override string ToString()
28     {
29         var s = "[";
30         for (var i = 0; i < Size; i++) s += _elements[i] + (i < Size - 1 ? "," : "");
31         return s + "]";
32     }
33 }
```

Outline

- 1 Basic OO in C#
- 2 Some specific C# mechanisms
- 3 Encapsulation, interfaces
- 4 Inheritance
- 5 Generics**
 - Generic classes
 - Generic methods
 - Generic interfaces
 - Iterators and collections**
 - Constrained polymorphism
 - Variance
- 6 Exceptions and some key C# libraries
- 7 Functional programming in C#

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 other 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

```
1 // namespace System.Collections.Generic
2 public interface IEnumerator<T> : IDisposable, System.Collections.
   IEnumerator
3 {
4     T Current { get; } // gets current element
5
6     bool MoveNext(); // move cursor to next position, returns if not "over"
7
8     void Reset(); // could be called to get back at the beginning
9
10    void Dispose(); // could be called at the end to release resources
11 }
```

```
1 public class UseEnumerator
2 {
3     public static void TestEnumerator()
4     {
5         IEnumerator<int> enumerator = new RangeEnumerator(0, 5);
6         while (enumerator.MoveNext())
7         {
8             Console.Write(enumerator.Current+" ");
9         }
10        enumerator.Dispose(); // optional
11    }
12 }
```

RangeEnumerator implementation

```
1 public class RangeEnumerator : IEnumerator<int>
2 {
3     private readonly int _stop;
4     public int Current { get; private set; }
5
6     public RangeEnumerator(int start, int stop)
7     {
8         Current = start-1;
9         _stop = stop;
10    }
11
12    public bool MoveNext()
13    {
14        Current++;
15        return Current < _stop;
16    }
17
18    // other non-important methods
19
20    public void Reset() // not interested in implementing it
21    {
22        throw new NotImplementedException();
23    }
24
25    public void Dispose() // nothing to do to dispose
26    {
27    }
28
29    object? IEnumerator.Current => Current; // needed by IEnumerator superinterface
30 }
```


The actual enumerable: Range

```
1 public class Range : IEnumerable<int>
2 {
3     public int Start {get; set; }
4     public int Stop {get; set; }
5
6     public IEnumerator<int> GetEnumerator() =>
7         new RangeEnumerator(Start, Stop);
8
9     // a "dirty" method, due to IEnumerable non-generic superinterface
10    IEnumerator IEnumerable.GetEnumerator() =>
11        GetEnumerator();
12 }
```

Using Range with IEnumerator

```
1 class UseEnumerables
2 {
3     public static void TestEnumerable()
4     {
5         IEnumerable<int> range = new Range{ Start = 0, Stop = 5};
6         var enumerator = range.GetEnumerator();
7         while (enumerator.MoveNext())
8         {
9             Console.Write(enumerator.Current+" ");
10        }
11        enumerator.Dispose();
12        Console.WriteLine();
13
14        var enumerator2 = range.GetEnumerator();
15        while (enumerator2.MoveNext())
16        {
17            if (enumerator2.Current > 2)
18            {
19                Console.Write(enumerator2.Current+" ");
20            }
21        }
22        enumerator2.Dispose();
23    }
24 }
```

foreach is compatible with IEnumerable!

```
1 static void TestForeach()
2 {
3     var range = new Range(); // default construction
4     range.Start = 0;
5     range.Stop = 5;
6     foreach (var i in range){
7         Console.Write(i+" ");
8     }
9     // note that foreach also calls Dispose
10    Console.WriteLine("");
11
12    // using object initializer
13    foreach (var i in new Range{ Start = 0, Stop = 5}){
14        Console.Write(i+" ");
15    }
16 }
```

The yield return construct

```
1 public class Helpers
2 {
3     public IEnumerable<int> RangeWithDelta(int start, int stop, int delta)
4     {
5         for (var i = start; i != stop; i += delta) yield return i;
6     }
7
8     public IEnumerable<string> Directions()
9     {
10         yield return "NORTH";
11         yield return "EAST";
12         yield return "SOUTH";
13         yield return "WEST";
14     }
15
16     public IEnumerator<int> FibonacciInfiniteEnumerator()
17     {
18         yield return 1;
19         int a = 1;
20         int b = 1;
21         while (true)
22         {
23             yield return b;
24             int sum = a + b;
25             a = b;
26             b = sum;
27         }
28     }
29 }
```

The yield return construct

```
1 class UseYieldReturn
2 {
3     public static void Main(string[] args)
4     {
5         var helpers = new Helpers();
6
7         foreach (var i in helpers.RangeWithDelta(0, -10, -2))
8         {
9             Console.Write(i + " ");
10        }
11        Console.WriteLine("");
12        foreach (var d in helpers.Directions())
13        {
14            Console.Write(d + " ");
15        }
16        Console.WriteLine("");
17        var fib = helpers.FibonacciInfiniteEnumerator();
18        while (true)
19        {
20            fib.MoveNext();
21            Console.Write(fib.Current + " ");
22            if (fib.Current > 1000) break;
23        }
24
25        fib.Dispose();
26    }
27 }
```

Creating a vector as an IEnumerable<T>

```
1 public class EnumerableVector<T> : IEnumerable<T>
2 {
3     private const int InitialCapacity = 10;
4     private const int MultiplicationFactor = 2;
5     private T[] _elements = new T[InitialCapacity];
6     public int Size { get; private set; } = 0;
7     public T this[int i] => _elements[i];
8
9     public void AddElement(T element)
10    {
11        if (Size == _elements.Length) Expand();
12        _elements[Size++] = element;
13    }
14
15    private void Expand()
16    {
17        var old = _elements;
18        _elements = new T[old.Length * MultiplicationFactor];
19        Array.Copy(old, _elements, Size);
20    }
21
22    public IEnumerator<T> GetEnumerator()
23    {
24        for (var i = 0; i < Size; i++) yield return _elements[i];
25    }
26
27    IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
28 }
```

Enumerating our vector

```
1 public class UseEnumerableVectors
2 {
3     public static void Main(string[] args)
4     {
5         var fibonacci = new EnumerableVector<int>();
6         fibonacci.AddElement(1);
7         fibonacci.AddElement(1);
8         for (var i = 2; i < 10; i++)
9         {
10             fibonacci.AddElement(fibonacci[i-2]+fibonacci[i-1]);
11         }
12
13         foreach (var i in fibonacci) Console.WriteLine(i);
14     }
15 }
```

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

```
1 public class ShowCollections
2 {
3     public static void Main(string[] args)
4     {
5         // IList and List are from System.Collections.Generic
6
7         IList<int> fibonacci = new List<int>();
8         fibonacci.Add(1);
9         fibonacci.Add(1);
10        for (var i = 2; i < 10; i++)
11        {
12            fibonacci.Add(fibonacci[i-2]+fibonacci[i-1]);
13        }
14
15        foreach (var i in fibonacci) Console.WriteLine(i);
16    }
17 }
```


An example application: ClassManagement

```
1 class Program
2 {
3     static void Main(string[] args)
4     {
5         var classManagement = new ClassManagement();
6         classManagement.AddStudent(new Student{ Id = 100, Name = "Maria"});
7         classManagement.AddStudent(new Student{ Id = 101, Name = "Gino", Evaluation = 30});
8         classManagement.AddStudent(new Student{ Id = 102, Name = "Marco", Evaluation = 18});
9
10        foreach (var student in classManagement.GetAll()) Console.WriteLine(student);
11        Console.WriteLine("---");
12        foreach (var name in classManagement.GetNames()) Console.WriteLine(name);
13        Console.WriteLine("---");
14        foreach (var student in classManagement.GetStudentsWithEvaluation())
15            Console.WriteLine(student);
16    }
17 }
18
19 public class Student
20 {
21     public int Id { get; set; }
22     public string Name { get; set; }
23     public int? Evaluation { get; set; }
24
25     public override string ToString()
26     {
27         return $"Id: {Id}, Name: {Name}, Evaluation: {Evaluation}";
28     }
29 }
```

An example application: ClassManagement

```
1 interface IClassManagement
2 {
3     void AddStudent(Student student);
4
5     IEnumerable<Student> GetAll();
6     IEnumerable<string> GetNames();
7     IEnumerable<Student> GetStudentsWithEvaluation();
8 }
9
10 public class ClassManagement : IClassManagement
11 {
12     private IList<Student> _students = new List<Student>();
13
14     public void AddStudent(Student student) => _students.Add(student);
15
16     public IEnumerable<Student> GetAll()
17     {
18         foreach (var student in _students) yield return student;
19     }
20
21     public IEnumerable<string> GetNames()
22     {
23         foreach (var student in _students) yield return student.Name;
24     }
25
26     public IEnumerable<Student> GetStudentsWithEvaluation()
27     {
28         foreach (var student in _students)
29         {
30             if (student.Evaluation != null) yield return student;
31         }
32     }
33 }
```

Outline

- 1 Basic OO in C#
- 2 Some specific C# mechanisms
- 3 Encapsulation, interfaces
- 4 Inheritance
- 5 Generics**
 - Generic classes
 - Generic methods
 - Generic interfaces
 - Iterators and collections
 - Constrained polymorphism**
 - Variance
- 6 Exceptions and some key C# libraries
- 7 Functional programming in C#

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

```
1 class Program
2 {
3     static void Main(string[] args)
4     {
5         ShowAll(Create<object>(5));
6         ShowAll(Create<DateTime>(5));
7         // ShowAll(Create<IList>(5)); // would not work!
8
9         var list1 = new List<object>(new object[]{1, true, new DateTime()});
10        var list2 = new List<int>(new int[]{1, 2,3,4,5});
11        Copy(list2, list1);
12        ShowAll(list1);
13        // Copy(list1, list2); // would not work!
14    }
15
16    public static void ShowAll<T>(IEnumerable<T> enumerable)
17    {
18        foreach(var t in enumerable) Console.Write(t+" ");
19        Console.WriteLine();
20    }
21
22    public static List<T> Create<T>(int size) where T: new()
23    {
24        var l = new List<T>();
25        for (var i=0; i<10; i++) l.Add(new T());
26        return l;
27    }
28
29    public static void Copy<TFrom,TTo>(List<TFrom> from, List<TTo> to) where TFrom: TTo
30    {
31        foreach (var t in from) to.Add(t);
32    }
33 }
```

Outline

- 1 Basic OO in C#
- 2 Some specific C# mechanisms
- 3 Encapsulation, interfaces
- 4 Inheritance
- 5 Generics**
 - Generic classes
 - Generic methods
 - Generic interfaces
 - Iterators and collections
 - Constrained polymorphism
 - Variance**
- 6 Exceptions and some key C# libraries
- 7 Functional programming in C#

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>`?

⇒ we could easily compromise the integrity of the list

```
1 static void Main(string[] args)
2 {
3     var list = new List<string>(new[] { "10", "20", "30" });
4     // AddAString(list,-1); // This code should not work, otherwise...
5     String s = list[list.Count - 1];
6 }
7
8 public static void AddAnElement(List<object> list, object last)
9 {
10     list.Add(last);
11 }
```

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

```
1 static void Main(string[] args)
2 {
3     string[] s = new[] {"a", "b", "c"};
4     SetToArray(s,10); // It works!
5     string str = s[0]; // It throws a: System.ArrayTypeMismatchException:
6 }
7
8 public static void SetToArray(object[] array, object element)
9 {
10     array[0] = element;
11 }
```


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

```
1 public interface IBaseVector
2 {
3     int Size { get; }
4 }
5
6 public interface IReadVector<out T> : IBaseVector, IEnumerable<T>
7 {
8     T this[int i] { get; }
9 }
10
11 public interface IWriteVector<in T>
12 {
13     void AddElement(T t);
14 }
15
16 public interface IVector<T> : IReadVector<T>, IWriteVector<T>
17 {
18 }
```

Expectation

```
1 class Program
2 {
3     static void Main(string[] args)
4     {
5         IVector<string> vs = new GenericVector<string>();
6         AddManyStrings(vs, 5);
7         IVector<object> vo = new GenericVector<object>();
8         AddManyStrings(vo, 5); // IWriteVector<object> <: IWriteVector<string>
9         IVector<Pair<string, string>> vp = new GenericVector<Pair<string, string>>();
10        vp.AddElement(new Pair<string, string>("a","b"));
11        Copy(vs, vo); // same as Copy<object>(vs, vo), or even Copy<string>(vs, vo)
12        Copy(vp, vo); // IReadVector<Pair<string,string>> <: IReadVector<object>
13        foreach (var o in vo) Console.WriteLine(o);
14    }
```

Straightforward implementation

```
1 public class GenericVector<T>: IVector<T>
2 {
3     private const int InitialCapacity = 10;
4     private const int MultiplicationFactor = 2;
5     private T[] _elements = new T[InitialCapacity];
6     public int Size { get; private set; } = 0;
7
8     public T this[int i] => _elements[i];
9
10
11     public void AddElement(T element)
12     {
13         if (Size == _elements.Length) Expand();
14         _elements[Size++] = element;
15     }
16
17     private void Expand()
18     {
19         var old = _elements;
20         _elements = new T[old.Length * MultiplicationFactor];
21         Array.Copy(old, _elements, Size);
22     }
23
24     public IEnumerator<T> GetEnumerator()
25     {
26         for (var i = 0; i < Size; i++) yield return _elements[i];
27     }
28
29     IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
30 }
```

Outline

- 1 Basic OO in C#
- 2 Some specific C# mechanisms
- 3 Encapsulation, interfaces
- 4 Inheritance
- 5 Generics
- 6 Exceptions and some key C# libraries**
 - Exceptions**
 - Generic collections
- 7 Functional programming in C#

.NET Exceptions

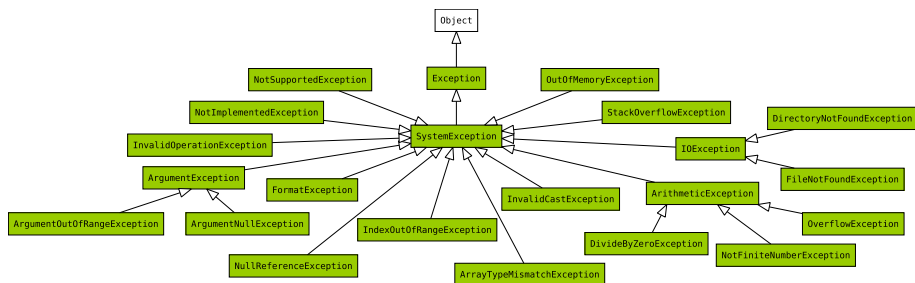
Exceptions as types

- .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-exhaustive)



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.

```
1 static void Recursive(int x)
2 {
3     if (x < 5)
4     {
5         Recursive(x + 1);
6         Console.WriteLine(x);
7     }
8     else
9     {
10        throw new Exception(" " + x);
11    }
12 }
```

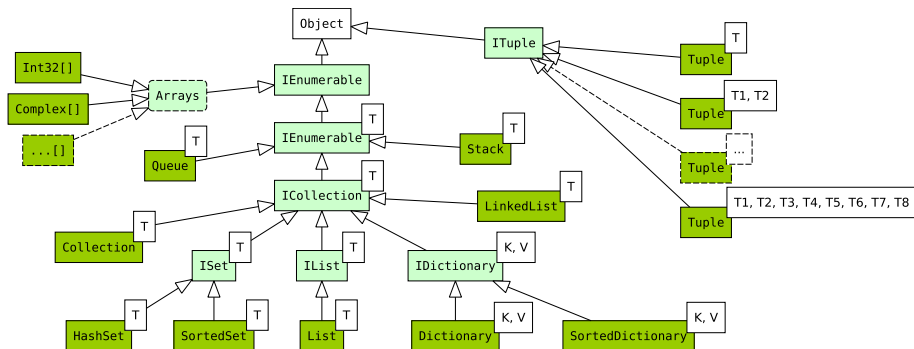

The try-catch-finally Construct

```
1  int readLines = 0;
2  while (true)
3  {
4      String line = null;
5      try
6      {
7          Console.Write("> ");
8          line = Console.ReadLine();
9          int number = int.Parse(line);
10         Console.WriteLine("Valid integer: " + number);
11     }
12     catch (FormatException e) { Console.WriteLine("Not a valid integer: " + line); }
13     catch (OverflowException e) { Console.WriteLine("Out of range integer: " + line); }
14     catch (ArgumentNullException e) { Console.WriteLine("Bye bye!"); break; }
15     finally { Console.WriteLine($"(So far I read {++readLines} lines)"); }
16 }
```

Outline

- 1 Basic OO in C#
- 2 Some specific C# mechanisms
- 3 Encapsulation, interfaces
- 4 Inheritance
- 5 Generics
- 6 Exceptions and some key C# libraries**
 - Exceptions
 - Generic collections**
- 7 Functional programming in C#

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*

`ICollection<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 $T \equiv \text{KeyValuePair}\langle K, V \rangle$

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 hierarchy exists, rooted in `ICollection`, for **non-generic** types
 - ▶ these are contained into `System.Collections`
- Another hierarchy exists, rooted in `ICollection<out T>`, for **read-only** types
 - ▶ these are contained into `System.Collections.Generic`
- Other types exist, named `IImmutable*<T>`, for **immutable** types
 - ▶ these are contained into `System.Collections.Immutable`

The ICollection<T> Interface I

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

```
1 // Generic container of items of type T
2 public interface ICollection<T> : IEnumerable<T>, IEnumerable
3 {
4     // Gets the amount of items in the collection
5     int Count { get; }
6
7     // Adds an item to the collection
8     void Add(T item);
9
10    // Removes all the items from the collection, emptying it
11    void Clear();
12
13    // Checks whether item is contained in the collection or not
14    bool Contains(T item);
15
16    // Inserts all the items of the collection into array, starting from arrayIndex
17    void CopyTo(T[] array, int arrayIndex);
18
19    // Removes item from the collection
20    bool Remove(T item);
21
22    // RECALL THAT GetEnumerator() IS INHERITED!
23 }
```

The ICollection<T> Interface II

Creation

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

```
new <Collection Class>() { <Item1>, ..., <ItemN> }
```

- ▶ where all $\langle Item_i \rangle$ are expressions returning instances of **T**

Usage

- The methods 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

```
1 ICollection<ExampleItem> collection = new Collection<ExampleItem>() {  
2     new ExampleItem(2),  
3     new ExampleItem(3),  
4     new ExampleItem(1),  
5     new ExampleItem(4)  
6 };  
7  
8 Console.WriteLine(collection.Count); // 4  
9  
10 Console.WriteLine(collection.Contains(new ExampleItem(5))); // false  
11  
12 collection.Add(new ExampleItem(5)); // item 5 is added  
13 Console.WriteLine(collection.Contains(new ExampleItem(5))); // true  
14 Console.WriteLine(collection.Count); // 5  
15  
16 Console.WriteLine(collection.Contains(new ExampleItem(1))); // true  
17 collection.Remove(new ExampleItem(1)); // item 1 is removed  
18 Console.WriteLine(collection.Count); // 4  
19 Console.WriteLine(collection.Contains(new ExampleItem(1))); // false  
20  
21 foreach (var item in collection)  
22 {  
23     Console.WriteLine(item); // 2, 3, 4, 5  
24 }  
25  
26 Console.WriteLine(collection.ToString()); // ???
```

The IList<T> Interface I

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

```
1 // Collection storing items in an orderly fashion
2 // (indexes are 0-based)
3 public interface IList<T> : ICollection<T>, IEnumerable<T>, IEnumerable
4 {
5     // Get or sets the index-th item in the list
6     T this[int index] { get; set; }
7
8     // Gets the index of an item in the list, or -1 if missing
9     int IndexOf(T item);
10
11     // Adds item in position index (subsequent items are shifted right)
12     void Insert(int index, T item);
13
14     // Removes the item in position index (subsequent items are shifted left)
15     void RemoveAt(int index);
16
17     // RECALL THAT METHODS FROM Object, ICollection<T>, and IEnumerable<T> ARE INHERITED!
18 }
```

The IList<T> Interface II

Creating/Usage example for the List class

```
1  IList<ExampleItem> list = new List<ExampleItem>() {  
2      new ExampleItem(2),  
3      new ExampleItem(3),  
4      new ExampleItem(1),  
5      new ExampleItem(4)  
6  };  
7  
8  foreach (var item in list)  
9  {  
10     Console.WriteLine(item); // 2, 3, 1, 4  
11 }  
12  
13 Console.WriteLine(list[1]); // 3  
14 list[1] = new ExampleItem(5);  
15 Console.WriteLine(list[1]); // 5  
16 Console.WriteLine(list[2]); // 1  
17  
18 list.Insert(2, new ExampleItem(6));  
19 Console.WriteLine(list[2]); // 6  
20 Console.WriteLine(list[3]); // 1  
21  
22 Console.WriteLine(list.IndexOf(new ExampleItem(1))); // 3  
23 list.RemoveAt(3); // removing item in position 3, i.e. 1  
24 Console.WriteLine(list.IndexOf(new ExampleItem(1))); // -1  
25  
26 Console.WriteLine(list.ToString()); // ???
```

The ISet<T> Interface I

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

```
1 // Collection storing items with no duplicates, regardless of their ordering
2 public interface ISet<T> : ICollection<T>, IEnumerable<T>, IEnumerable
3 {
4     // Adds an item to the set, returning false if it was already present, true otherwise
5     bool Add(T item);
6
7     // Removes all elements in the enumerable from the current set
8     void ExceptWith(IEnumerable<T> other);
9
10    // Modifies the current set so that it contains only elements
11    // that are contained in both the current set and the provided enumerable
12    void IntersectWith(IEnumerable<T> other);
13
14    // Returns true if all the items of the set are contained in the enumerable,
15    // and the enumerable contains other items as well
16    bool IsProperSubsetOf(IEnumerable<T> other);
17
18    // Returns true if all the items of the enumerable are contained in current set,
19    // and the current set contains other items as well
20    bool IsProperSupersetOf(IEnumerable<T> other);
21
22    // Returns true if all the items of the set are contained in the enumerable
23    bool IsSubsetOf(IEnumerable<T> other);
24
25    // Returns true if all the items of the enumerable are contained in current set
26    bool IsSupersetOf(IEnumerable<T> other);
27
```

The ISet<T> Interface II

```
28 // Returns true if some item of the current set is in the enumerable as well
29 bool Overlaps(IEnumerable<T> other);
30
31 // Returns true if the current set and the specified collection contain the same
32 // elements
33 bool SetEquals(IEnumerable<T> other);
34
35 // Modifies the current set so that it contains only those items that
36 // are present either in the current set or in the enumerable, but not both
37 void SymmetricExceptWith(IEnumerable<T> other);
38
39 // Modifies the current set so that it contains both its original items and the ones
40 // in the provided enumerable, without repetitions
41 void UnionWith(IEnumerable<T> other);
42
43 // RECALL THAT METHODS FROM Object, ICollection<T>, and IEnumerable<T> ARE INHERITED!
44 }
```

The ISet<T> Interface III

Creating/Usage example for the HashSet class

```
1 ISet<ExampleItem> set = new HashSet<ExampleItem>() {  
2     new ExampleItem(2), new ExampleItem(3), new ExampleItem(2),  
3     new ExampleItem(1), new ExampleItem(2)  
4 };  
5  
6 foreach (var item in set) Console.WriteLine(item); // 1, 2, 3 (in some order)  
7  
8 Console.WriteLine(set.Count); // 3  
9 Console.WriteLine(set.Add(new ExampleItem(3))); // false  
10 Console.WriteLine(set.Count); // 3  
11  
12 var anotherSet = new HashSet<ExampleItem>() { new ExampleItem(1), new ExampleItem(2) };  
13  
14 Console.WriteLine(set.IsProperSupersetOf(anotherSet)); // true  
15 Console.WriteLine(anotherSet.IsProperSubsetOf(set)); // true  
16 anotherSet.Add(new ExampleItem(4));  
17 Console.WriteLine(set.IsProperSupersetOf(anotherSet)); // false  
18 Console.WriteLine(anotherSet.IsProperSubsetOf(set)); // false  
19 Console.WriteLine(set.Overlaps(anotherSet)); // true  
20 set.UnionWith(anotherSet);  
21  
22 foreach (var item in set) Console.WriteLine(item); // 1, 2, 3, 4 (in some order)  
23  
24 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

```
1 // A collection containing a number of key-value pairs
2 // where keys are of type K (and cannot be null) and values are of type V
3 public interface IDictionary<K, V> : ICollection<KeyValuePair<K, V>> where K : notnull
4 {
5     // Gets or sets the element with the specified key
6     V this[K key] { get; set; }
7
8     // Returns a collection containing all the keys used in the current dictionary
9     ICollection<K> Keys { get; }
10
11     // Returns a collection containing all the values used in the current dictionary
12     ICollection<V> Values { get; }
13
14     // Adds a new key-value pair. Throws an exception if key is already present
15     void Add(K key, V value);
16
17     // Returns true if a pair indexed by key is present, false otherwise
18     bool ContainsKey(K key);
19
20     // Removes a key and its corresponding pair. Returns null if the key is missing
21     bool Remove(K key);
22
23     // RECALL THAT METHODS FROM Object, ICollection<T>, and IEnumerable<T> ARE INHERITED!
24 }
```

The IDictionary<K, V> Interface II

Creating/Usage example for the Dictionary class

```
1  IDictionary<string, ExampleItem> map = new Dictionary<string, ExampleItem>() {
2      ["giovanni"] = new ExampleItem(1),
3      ["mirko"] = new ExampleItem(2),
4      ["andrea"] = new ExampleItem(3)
5  };
6
7  foreach (KeyValuePair<string, ExampleItem> item in map)
8  {
9      Console.WriteLine(item.Key); // giovanni, mirko, andrea
10     Console.WriteLine(item.Value); // 1, 2, 3
11 }
12
13 foreach (var (key, val) in map)
14 {
15     Console.WriteLine(val); // 1, 2, 3
16     Console.WriteLine(key); // giovanni, mirko, andrea
17 }
18
19 map["andrea"] = new ExampleItem(0); // here the value of key "andrea" is being replaced
20 map.Add("matteo", new ExampleItem(3)); // here a novel key-value pair is being added
21 map["daniilo"] = new ExampleItem(4); // here a novel key-value pair is being added
22
23 try
24 {
25     map.Add("matteo", new ExampleItem(4)); // fails as key "matteo" is already present
26 }
27 catch (ArgumentException e) { /* ignore */ }
```


The IDictionary<K, V> Interface III

```
29 Console.WriteLine(map["giovanni"]); // 1
30 Console.WriteLine(map["mirko"]); // 2
31 Console.WriteLine(map["andrea"]); // 0
32 Console.WriteLine(map["matteo"]); // 3
33 Console.WriteLine(map["danilo"]); // 4
34
35 foreach (string key in map.Keys)
36     Console.WriteLine(key); // giovanni, mirko, andrea, matteo, danilo (in some order)
37 foreach (ExampleItem val in map.Values)
38     Console.WriteLine(val); // 1, 2, 0, 3, 4 (in some order)
39
40 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

```
1 // A type for IMMUTABLE triplets of items where the 1st one is of type T1,
2 // the 2nd one is of type T2, and the 3rd one is of type T3
3 public class Tuple<T1, T2, T3>
4 {
5     private readonly object[] _items;
6
7     public Tuple(T1 x, T2 y, T3 z) { _items = new object[] {x, y, z}; }
8
9     public T1 Item1 => (T1) _items[0]; // Gets the first item
10    public T2 Item2 => (T2) _items[1]; // Gets the second item
11    public T3 Item3 => (T3) _items[2]; // Gets the third item
12
13    public object this[int index] => _items[index];
14
15    int Length => _items.Length;
16
17    // method Equals compares triplets by value
18    // method GetHashCode is coherent w.r.t. equals
19
20    // method ToString prints the items
21 }
```

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

Creating/Usage example for the Dictionary class

```
1 Tuple<string, int> pair = Tuple.Create("giovanni", 29);
2 Tuple<string, int, DateTime> triplet = Tuple.Create("mirko", 45, DateTime.Now);
3
4 var number = 30;
5 Tuple<string, int> anotherPair = Tuple.Create("giovanni", number - 1);
6 Tuple<string, int, DateTime> anotherTriplet = Tuple.Create("mirko", 45, DateTime.Now);
7
8 Console.WriteLine(pair.ToString()); // (giovanni, 29)
9 Console.WriteLine(anotherPair.ToString()); // (giovanni, 29)
10 Console.WriteLine(triplet.ToString()); // (mirko, 45, 19/03/2021 16:33:30)
11 Console.WriteLine(anotherTriplet.ToString()); // (mirko, 45, 19/03/2021 16:33:30)
12
13 Console.WriteLine(pair.Equals(anotherPair)); // true
14 Console.WriteLine(triplet.Equals(anotherTriplet)); // false ..... WHY? :)
15
16 Console.WriteLine(triplet.Item1 == anotherTriplet.Item1); // true
17 Console.WriteLine(triplet.Item2 == anotherTriplet.Item2); // true
18 Console.WriteLine(triplet.Item3 == anotherTriplet.Item3); // false ..... WHY? :)
```

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 dependng 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

```
1 public interface IFeeCalculator
2 {
3     int Fee(int amount);
4 }
5
6 public interface IWithdrawAction
7 {
8     void Handler(int balance, int amount);
9 }
10
11 public class StandardFee : IFeeCalculator
12 {
13     public int Fee(int amount) => 1;
14 }
15
16 public class BusinessFee : IFeeCalculator
17 {
18     public int Fee(int amount) => amount > 100 ? 1 : 0;
19 }
20
21 public class OnConsoleAction : IWithdrawAction
22 {
23     public void Handler(int balance, int amount) =>
24         Console.WriteLine("Could not withdraw "+amount);
25 }
26
27 public class ErrorAction : IWithdrawAction
28 {
29     public void Handler(int balance, int amount) =>
30         Console.Error.WriteLine($"ERROR on Withdrawal: {balance}:{amount}");
31 }
```

FlexibleBankAccount

```
1 public class FlexibleBankAccount : IBankAccount
2 {
3     public int Balance { get; private set; }
4     public string Name { get; }
5
6     private IFeeCalculator _feeCalculator;
7
8     private ISet<IWithdrawAction> _actions;
9
10    public FlexibleBankAccount(string name, IFeeCalculator feeCalculator, ISet<
11    IWithdrawAction> actions)
12    {
13        _feeCalculator = feeCalculator;
14        _actions = actions;
15        Name = name;
16    }
17
18    public void Deposit(int amount) => Balance += amount;
19
20    public void Withdraw(int amount)
21    {
22        var fee = _feeCalculator.Fee(amount);
23        if (Balance < amount + fee)
24        {
25            foreach (var wa in _actions) wa.Handler(Balance, amount);
26        }
27        else Balance -= (amount + fee);
28    }
29
30    public override string ToString() => $"Name: {Name}, Balance: {Balance}";
31 }
```

UseFlexibleBankAccount

```
1 class UseFlexibleBankAccounts
2 {
3     static void Main(string[] args)
4     {
5         var actions1 = new HashSet<IWithdrawAction>(new []{new OnConsoleAction()});
6         var ba1 = new FlexibleBankAccount("a", new BusinessFee(), actions1);
7         ba1.Deposit(1000); // 1000
8         ba1.Withdraw(50); // 950
9         ba1.Withdraw(150); // 699
10        ba1.Withdraw(1000); // 699, + console output
11        Console.WriteLine(ba1.Balance); // 699
12
13        var actions2 = new HashSet<IWithdrawAction>(
14            new IWithdrawAction[]{new OnConsoleAction(), new ErrorAction()});
15        var ba2 = new FlexibleBankAccount("a", new StandardFee(), actions2);
16        ba2.Deposit(1000); // 1000
17        ba2.Withdraw(50); // 949
18        ba2.Withdraw(150); // 798
19        ba2.Withdraw(1000); // 798, + console/error outputs
20        Console.WriteLine(ba2.Balance); // 700
21    }
22 }
```


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

```
1 public delegate int ComputeFee(int amount);
2 public delegate void WithdrawAction(int balance, int amount);
3
4
5 public class FlexibleBankAccount : IBankAccount
6 {
7     public int Balance { get; private set; }
8     public string Name { get; }
9
10    private ComputeFee _computeFee;
11
12    private WithdrawAction _action;
13
14    public FlexibleBankAccount(string name, ComputeFee computeFee, WithdrawAction action)
15    {
16        _computeFee = computeFee;
17        _action = action;
18        Name = name;
19    }
20
21    public void Deposit(int amount) => Balance += amount;
22
23    public void Withdraw(int amount)
24    {
25        var fee = _computeFee(amount);
26        if (Balance < amount + fee) _action(Balance, amount);
27        else Balance -= (amount + fee);
28    }
29
30    public override string ToString() => $"Name: {Name}, Balance: {Balance}";
31 }
```

Instantiation of delegates

```
1 class UseFlexibleBankAccounts
2 {
3     private static int StandardFee(int amount) => 1;
4     private static int BusinessFee(int amount) => amount > 100 ? 1 : 0;
5     private static void OnConsoleAction(int balance, int amount) =>
6         Console.WriteLine("Could not withdraw "+amount);
7     private static void ErrorAction(int balance, int amount) =>
8         Console.Error.WriteLine($"ERROR on Withdrawal: {balance}:{amount}");
9
10    static void Main(string[] args)
11    {
12        var fee1 = new ComputeFee(BusinessFee);
13        var act1 = new WithdrawAction(OnConsoleAction);
14        var ba1 = new FlexibleBankAccount("a", fee1, act1);
15        // var ba1 = new FlexibleBankAccount("a", BusinessFee, OnConsoleAction);
16        ba1.Deposit(1000); // 1000
17        ba1.Withdraw(50); // 950
18        ba1.Withdraw(150); // 699
19        ba1.Withdraw(1000); // 699, + console output
20        Console.WriteLine(ba1.Balance); // 699
21
22        var fee2 = new ComputeFee(StandardFee);
23        var act2 = new WithdrawAction(OnConsoleAction) + new WithdrawAction(ErrorAction);
24        var ba2 = new FlexibleBankAccount("a", fee2, act2);
25        ba2.Deposit(1000); // 1000
26        ba2.Withdraw(50); // 949
27        ba2.Withdraw(150); // 798
28        ba2.Withdraw(1000); // 798, + console/error outputs
29        Console.WriteLine(ba2.Balance); // 700
30    }
31 }
```

Anonymous functions

Pros and cons of delegates

- surely reduce boilerplate 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)...

Anonymous 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

```
1 static void Main(string[] args)
2 {
3     // use of anonymous function to fill a variable
4     WithdrawAction act1 = delegate(int balance, int amount)
5     {
6         Console.WriteLine("Could not withdraw " + amount);
7     };
8     // use of anonymous function in-line where needed
9     var ba1 = new FlexibleBankAccount("a", delegate(int amount) { return 1; }, act1);
10    ba1.Deposit(1000); // 1000
11    ba1.Withdraw(50); // 950
12    ba1.Withdraw(150); // 699
13    ba1.Withdraw(1000); // 699, + console output
14    Console.WriteLine(ba1.Balance); // 699
15
16    ComputeFee fee2 = delegate(int amount) { return amount > 100 ? 1 : 0; };
17    // cannot use them to chain delegates
18    var act2 = new WithdrawAction(OnConsoleAction) + new WithdrawAction(ErrorAction);
19    var ba2 = new FlexibleBankAccount("a", fee2, act2);
20    ba2.Deposit(1000); // 1000
21    ba2.Withdraw(50); // 949
22    ba2.Withdraw(150); // 798
23    ba2.Withdraw(1000); // 798, + console/error outputs
24    Console.WriteLine(ba2.Balance); // 700
25
26    // Use in libraries...
27    List<int> list = new List<int>(new int[]{10,30,20,40,5});
28    list.Sort(delegate(int i, int i1) { return i - i1; });
29    foreach(var i in list) Console.WriteLine(i); // 5,10,20,30,...
30    list.Sort(delegate(int i, int i1) { return i1 - i; });
31    foreach(var i in list) Console.WriteLine(i); // 40,30,20,...
32 }
```

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 heavily 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: $(T_1\ x_1, \dots, T_n\ x_n) \Rightarrow \{<body>\}$
- With inference: $(x_1, \dots, x_n) \Rightarrow \{<body>\}$
- With single-expression body: $(T_1\ x_1, \dots, T_n\ x_n) \Rightarrow <exp>$
- With single argument and inference: $x \Rightarrow \{<body>\}$
- With unused inputs: $(_, \dots, _) \Rightarrow \{<body>\}$
- ...and combinations

⇒ should generally use the shortest version possible

Using lambdas

```
1 static void Main(string[] args)
2 {
3     // use of anonymous function in-line where needed
4     var ba1 = new FlexibleBankAccount(
5         "a",
6         _ => 1, // lambda of the form x=><exp>
7         (_, a) => Console.WriteLine("Could not withdraw " + a));
8     ba1.Deposit(1000); // 1000
9     ba1.Withdraw(50); // 950
10    ba1.Withdraw(150); // 699
11    ba1.Withdraw(1000); // 699, + console output
12    Console.WriteLine(ba1.Balance); // 699
13
14    WithdrawAction act2 = (_, a) => Console.WriteLine("Could not withdraw " + a);
15    act2 += (b, a) => Console.Error.WriteLine($"ERROR on Withdrawal: {b}:{a}");
16    var ba2 = new FlexibleBankAccount("a", amount => amount > 100 ? 1 : 0 , act2);
17    ba2.Deposit(1000); // 1000
18    ba2.Withdraw(50); // 949
19    ba2.Withdraw(150); // 798
20    ba2.Withdraw(1000); // 798, + console/error outputs
21    Console.WriteLine(ba2.Balance); // 700
22
23    // Use in libraries...
24    List<int> list = new List<int>(new int[]{11,33,22,49,5});
25    list.Sort((i,j) => i%10 - j%10);
26    foreach(var i in list) Console.WriteLine(i); // 5,10,20,30,...
27    list.Sort((i,j) => j-i);
28    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 Predicate<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

```
1 public class FlexibleBankAccount : IBankAccount
2 {
3     public int Balance { get; private set; }
4     public string Name { get; }
5
6     private Func<int,int> _computeFee;
7
8     private Action<int,int> _action;
9
10    public FlexibleBankAccount(string name, Func<int,int> computeFee, Action<int,int>
11    action)
12    {
13        _computeFee = computeFee;
14        _action = action;
15        Name = name;
16    }
17
18    public void Deposit(int amount) => Balance += amount;
19
20    public void Withdraw(int amount)
21    {
22        var fee = _computeFee(amount);
23        if (Balance < amount + fee) _action(Balance, amount);
24        else Balance -= (amount + fee);
25    }
26
27    public override string ToString() => $"Name: {Name}, Balance: {Balance}";
28 }
```

Use BankAccount with Func and Action

```
1 class UseFlexibleBankAccounts
2 {
3     static void Main(string[] args)
4     {
5         var ba1 = new FlexibleBankAccount(
6             "a",
7             _ => 1,
8             (_, a) => Console.WriteLine("Could not withdraw " + a));
9         // var ba1 = new FlexibleBankAccount("a", BusinessFee, OnConsoleAction);
10        ba1.Deposit(1000); // 1000
11        ba1.Withdraw(50); // 950
12        ba1.Withdraw(150); // 699
13        ba1.Withdraw(1000); // 699, + console output
14        Console.WriteLine(ba1.Balance); // 699
15
16        Action<int, int> act2 = (_, a) => Console.WriteLine("Could not withdraw " + a);
17        act2 += (b, a) => Console.Error.WriteLine($"ERROR on Withdrawal: {b}:{a}");
18        var ba2 = new FlexibleBankAccount("a", a => a > 100 ? 1 : 0, act2);
19        ba2.Deposit(1000); // 1000
20        ba2.Withdraw(50); // 949
21        ba2.Withdraw(150); // 798
22        ba2.Withdraw(1000); // 798, + console/error outputs
23        Console.WriteLine(ba2.Balance); // 700
24    }
25 }
```

Exercise with Func, Predicate and Action

```
1 class Helpers {
2
3     public static void ShowAll(IEnumerable<int> elems)
4     {
5         foreach (var e in elems) Console.WriteLine(e);
6     }
7
8     public static IEnumerable<int> GetAllPositive(IEnumerable<int> elems)
9     {
10         foreach (var e in elems) if (e>0) yield return e;
11     }
12
13     public static IEnumerable<int> IncrementAll(IEnumerable<int> elems)
14     {
15         foreach (var e in elems) yield return e+1;
16     }
17
18     public static int SumAll(IEnumerable<int> elems)
19     {
20         var sum = 0;
21         foreach (var e in elems) sum += e;
22         return sum;
23     }
24
25     public static IEnumerable<int> Iterate(int size)
26     {
27         for (var i=0; i<size; i++) yield return i;
28     }
29 }
```

Solution for the first three

```
1 class GeneralizedHelpers
2 {
3     // Generalizing ShowAll
4     public static void ForEach<T>(IEnumerable<T> elems, Action<T> action)
5     {
6         foreach (var e in elems) action(e);
7     }
8
9     // Generalizing GetAllPositive
10    public static IEnumerable<T> Filter<T>(IEnumerable<T> elems, Predicate<T> pred)
11    {
12        foreach (var e in elems)
13            if (pred(e)) yield return e;
14    }
15
16    // Generalizing Increment
17    public static IEnumerable<TResult> Map<TResult, T>(IEnumerable<T> elems, Func<T,
18    TResult> map)
19    {
20        foreach (var e in elems) yield return map(e);
21    }
22
23    // Generalize the others by yourself!
```

Expectations

```
1 class UseGeneralizedHelpers
2 {
3     static void Main(string[] args)
4     {
5         var list = new List<string>(new[] { "a", "bb", "ccc", "dddd" });
6         foreach (s in list) Console.Write(s + " "); // a; bb; ccc; dddd;
7         Console.WriteLine();
8
9         var list2 = Filter(list, s => s.Length < 4);
10        foreach (s in list2) Console.Write(s + " "); // a; bb; ccc;
11        Console.WriteLine();
12
13        var list3 = Map(list, s => s.Length);
14        foreach (s in list3) Console.Write(s + " "); // 1; 2; 3;
15        Console.WriteLine();
16
17        // Generalize SumAll to extract from the above list: "abbcccdddd"
18        // Use same generalization above to extract from the above list the shortest string
19        // Generalize Increment to produce enumeration "a", "aa", "aaa",...
20    }
21 }
```

A preview of standard LINQ library

```
1 class Program
2 {
3     static void Main(string[] args)
4     {
5         var archive = new List<Person>(new Person[]
6         {
7             new Person( "Mario", "Rossi", new DateTime(1990, 1, 18), false),
8             new Person( "Gino", "Bianchi", new DateTime(1980, 2, 20), false),
9             new Person( "Carla", "Neri", new DateTime(1992, 12, 2), true),
10            new Person( "Rosa", "Rosa", new DateTime(1970, 3, 1), false),
11            new Person( "Italo", "Casadei", new DateTime(1990, 12, 25), true)
12        });
13
14        var marriedPeople = archive.FindAll(p => p.Married).Select(p => p.ToString());
15        var toShow = string.Join(" /// ", marriedPeople.Select(p => p.ToString()));
16        Console.WriteLine(toShow);
17
18        // count people born later than 1/1/1990
19        Predicate<Person> young = p => p.Birth.CompareTo(new DateTime(1990, 1, 1)) > 0;
20        var toShow2 = archive.FindAll(p => young(p)).Count;
21
22        Console.WriteLine(toShow2);
23
24        var marriedPeople2 =
25            from person in archive
26            where person.Married select person.ToString();
27        Console.WriteLine(string.Join(" /// ", marriedPeople2.Select(p => p.ToString())));
28
29    }
```

The used Person class

```
1 class Person
2 {
3     public string Name { get; }
4     public string Surname { get; }
5     public DateTime Birth { get; }
6     public Boolean Married { get; }
7
8     public Person(string name, string surname, DateTime birth, bool married)
9     {
10         Name = name;
11         Surname = surname;
12         Birth = birth;
13         Married = married;
14     }
15
16     public override string ToString()
17     {
18         return $"Name: {Name}, Surname: {Surname}, Birth: {Birth}, Married: {Married}";
19     }
20 }
```