Object-oriented programming in C#

Mirko Viroli, Giovanni Ciatto

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

a.a. 2022/2023

Outline

Brief introduction to C#

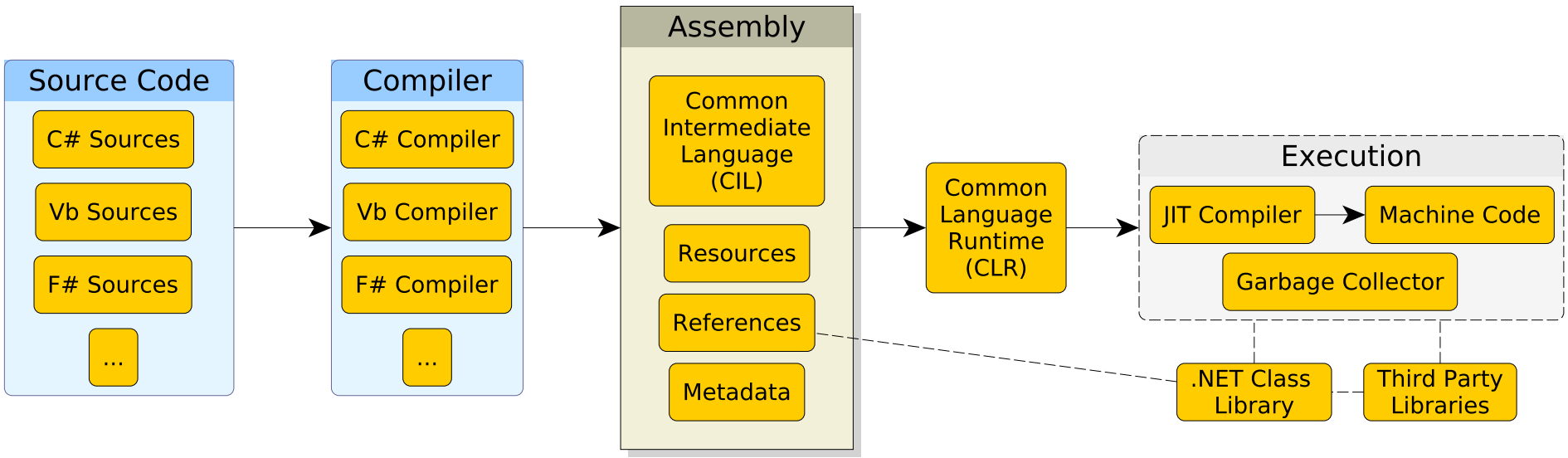
C# and .NET

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

.NET

Main elements

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



.NET Platform – Present vs. Past

Past to Present

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

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

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

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

- Since .NET 5, implementations are aligned

In these slides

Stick to .NET 6

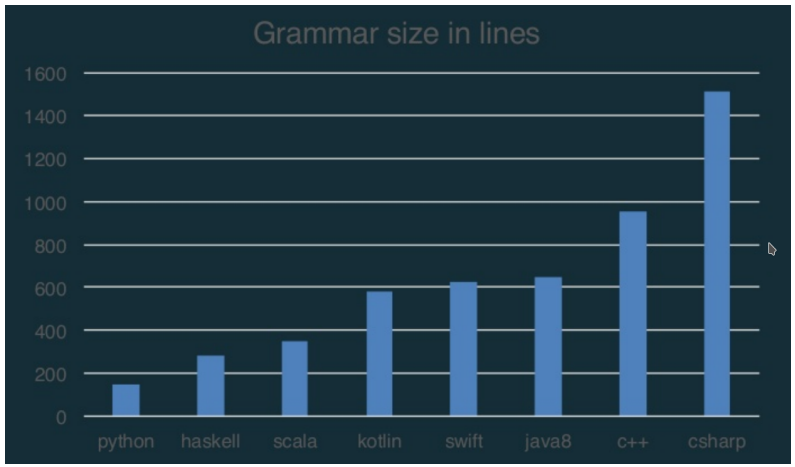
Features of C#

Ingredients

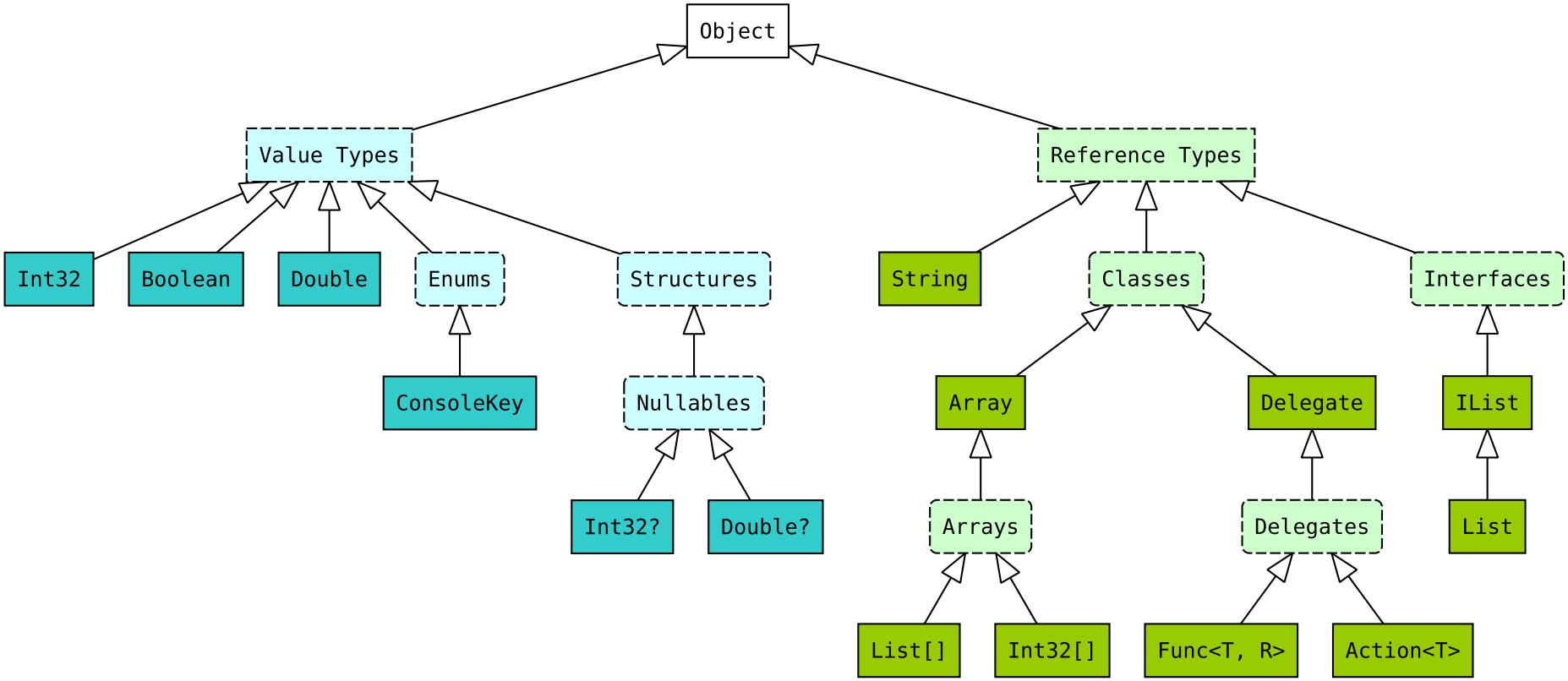
- C-like language: the imperative and structured parts are very similar
- Java-like language: essentially very similar to Java, specially at the beginning
- Static and strong typing: types are checked at run-time, preventing ill-typed operations
- Object-orientation: object by references, automatic garbage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

Philosophy

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



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



Variables: initialisation and inference

On variables – essentially as in Java

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

```
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 $0 \ldots (2^8 - 1)$
`SByte`	`sbyte`	*val*	1	integers in $-2^7 \ldots (2^7 - 1)$
`Int16`	`short`	*val*	2	integers in $-2^{15} \ldots (2^{15} - 1)$
`UInt16`	`ushort`	*val*	2	integers in $0 \ldots (2^{16} - 1)$
`Int32`	`int`	*val*	4	integers in $-2^{31} \ldots (2^{31} - 1)$
`UInt32`	`uint`	*val*	4	integers in $0 \ldots (2^{32} - 1)$
`Int64`	`long`	*val*	8	integers in $-2^{63} \ldots (2^{63} - 1)$
`UInt64`	`ulong`	*val*	8	integers in $0 \ldots (2^{64} - 1)$
`Float`	`float`	*val*	4	abs in $1.5 \times 10^{-45} \ldots 3.4 \times 10^{38}$
`Double`	`double`	*val*	8	abs in $5.0 \times 10^{-324} \ldots 1.7 \times 10^{308}$
`Decimal`	`decimal`	*val*	16	abs in $1.0 \times 10^{-28} \ldots 7.9228 \times 10^{28}$
`Object`	`object`	*ref*	O(1)	anything
`String`	`string`	*ref*	O(n)	sequences of n UTF-16 characters

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

Outline

C# classes

The core of OOP is essentially as in Java

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

Formatting

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

Point3D

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

Solution

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

Working with command line

Creating a new project into your solution

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

Run the project

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

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 getter is a method `SetXYZ` taking the property `XYZ`'s value and returning nothing
- properties that one only wants to read have no setter, and vice-versa for getters

Person with Getters and Setters

```
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 }
```

Immutability

Design for immutability

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

Readonly fields

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

Properties

Improving over Get/Set accessors

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

Case 1: General notation

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

Case 2: Auto-implemented properties

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

Case 3: Expression-bodied getter

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

Person with Properties

```
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

.NET Arrays

Array types

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

.NET Arrays

Arrays features

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

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

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

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

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

Array Types Instantiation I

Constructors for N-dimensional Arrays of T

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

- 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

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

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

Array Types Instantiation II

```
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

.NET Nullables

Nullable types definition

- let T by a value type of any sort, then $T?$ denotes the nullable T type
- A nullable type $T?$ can be defined as $T \cup \{ \texttt{null} \}$.
- A variable of type $T?$ can be assigned with any admissible value of T, or with `null`

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

Nullables features

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

  `HasValue` returning null if the object is null

  `Value` returning the non-null value, if present

- When non-null, nullable-type variables behave like they non-nullable counterparts

Nullable Types Operators

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

```
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"
```