C# Fundamentals

Dr. Patrik Reali (Elca Informatik)

FHA, Sommer ‘04

C# Fundamentals

- Application
  - Main method
- Language
  - Builtin Types
    - primitive
    - arrays
  - Iterations
  - Control Flow
  - Parameters
- Framework
  - System.Object
  - System.String
  - System.Array
  - System.Console

C# Class Declaration (partial!)

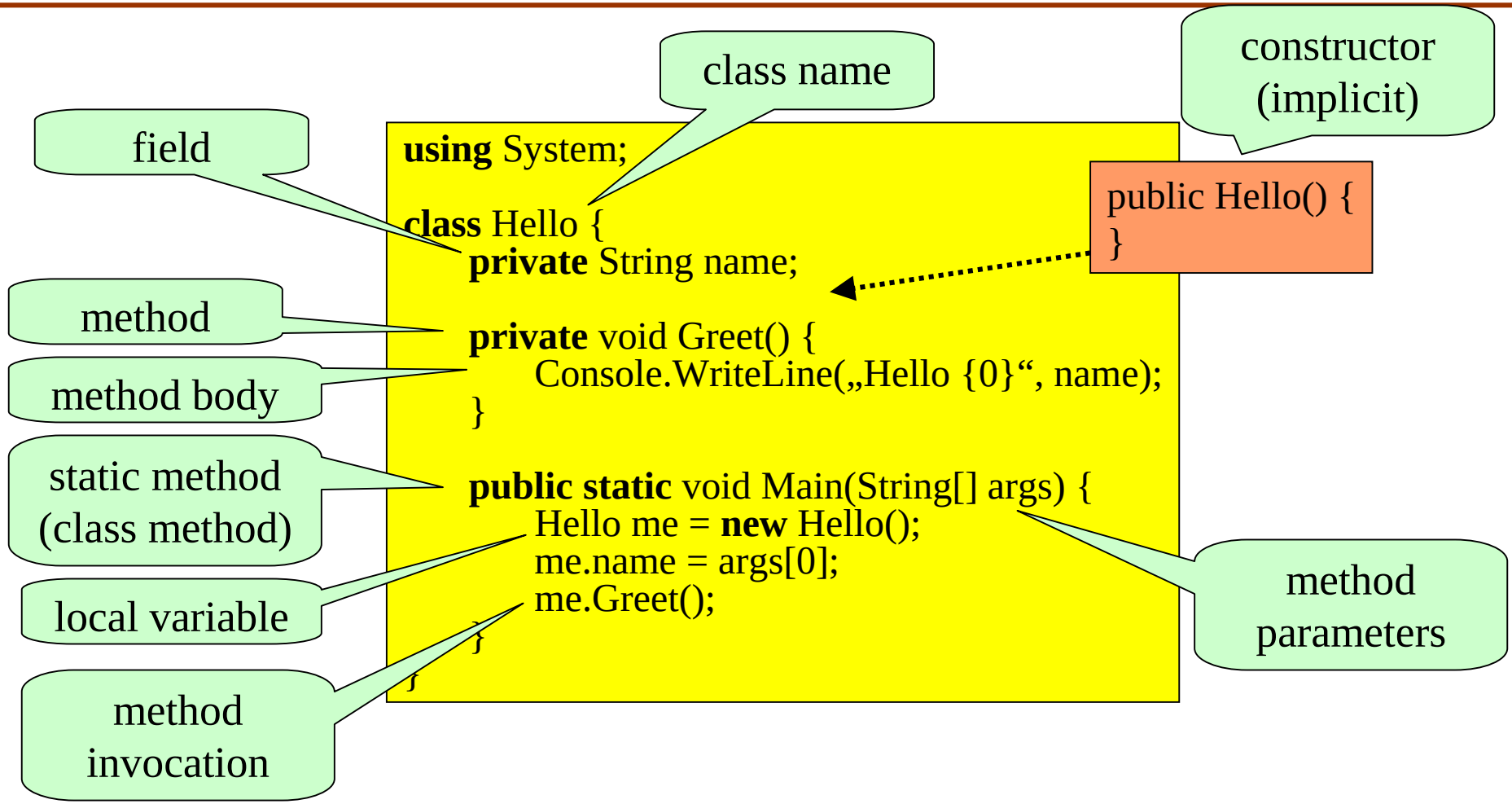
- A class is defined with

```
[modifiers] class ClassName {
    // fields
    [modifiers] type FieldName [= initial value];

    // constructors
    [modifiers] ClassName([arguments]) [: this/super([arguments])]{
        //code
    }
    // methods
    [modifiers] type MethodName([arguments]) {
        // code
    }
}
```

- Modifiers: *(...complete list later...)*
  - visibility (e.g. **public**, **private**)
  - binding (e.g. **static**, **virtual**, **override**)
  - restrictions (e.g. **abstract**, **readonly**)

C# Class Declaration



C#: Main() Method

Main must

- be named Main
- be static

Main can

- return void or int
- have (void) or (string[]) parameters

```
public static void Main() {
        // Main
}
```

```
public static int Main(string[] args) {
        // Main
        return result_code;
}
```

```
public static void Main(string[] args) {
        // Main
}
```

C# Command Line Parameters

```
using System;
class Test {
      public static void Main(String[] args) {
            for(int x = 0; x < args.Length; x++) {
                  Console.WriteLine(„Arg[{0}] = {1}“, x, args[x]);
            }
      }
}
```

- Test.exe abc 5 blah 3.2
- output:

```
abc
5
blah
3.2
```

Namespaces

- Classes are organized into hierarchical namespaces, e.g. System.Security.Principal
- Declare them using **namespace** keyword

```
namespace A {
    namespace B {
        public class C {
            // fully qualified name is A.B.C
            ....
        }
    }
}
```

```
namespace A.B {
    public class D {
        ...
    }
}
```

- **using** allows local unqualified access to classes and structures in a namespace

```
// make declarations in A.B
// locally visible
using A.B;

C.Method();          // instead of A.B.C.Method()
```

System.Console.Write

- Two static methods
  - System.Console.Write
  - System.Console.WriteLine
- Embed parameters with „{param}“
- Multiple parameters possible
- Output format is configurable
- Simple

```
Console.WriteLine(„Simple Text“);
```

- With parameters

```
Console.WriteLine(„Hi {0}, your ID is {1}“, name, id);
```

- With output configuration

```
Console.WriteLine(„Hi {0}, your ID is {1:X6}“, name, id);
```

C	Currency
D	Decimal
E	Exponential
F	Fixed Point
G	General
N	Numerical
X	Hex

```
Simple Text
```

```
Hi Patrik, your ID is 1234
```

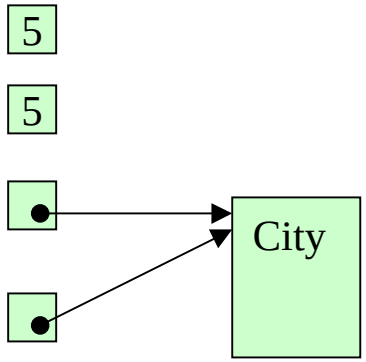
```
Hi Patrik, your ID is 0004D2
```

Value Types vs. Reference Types

- Different copy-semantic:

```
a = 5;
b = a;
a += 1;
      // b == 5
```

```
a = new Address();
a.City = „Lugano“;
b = a;
a.City = „Locarno“;
      // b.City == „Locarno“
```



- Reference-Types:
  - assignment creates new reference to instance
  - class instances are reference-types
- Value-Types:
  - assignment creates new copy of the value
  - primitive types and structures are value-types

C# Structures

- Structures are value-types

```
struct Date {
        int Day;
        int Month;
        int Year;
}
```

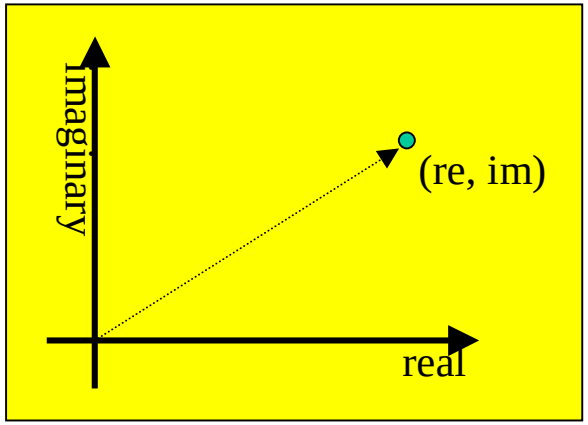
```
Date a;
Date b = a;

a.Day++;
        // a.Day != b.Day
```

- Structures are
  - stack-allocated
  - subclasses of System.ValueType
  - final (no further subclassing)
- Constructors:
  - The default constructor is always present
  - Custom constructors are allowed

Execise

- Create a structure (not a class, why?) to implement Complex numbers



- Construction:
  - cartesian coordinates (re, im)
  - polar coordinates (radius, angle)
- Query coordinates
  - GetRe(), GetIm(), GetRadius(), GetAngle()
- Useful to know
  - System.Math library
  - im = radius * cos(angle)
  - re = radius * sin(angle)
  - angle = atan(re/im)

System.Object

- *„The mother of all types“*
  - public **virtual** Boolean Equals(Object obj);
  - public **virtual** Int32 GetHashCode();
  - public Type GetType();
  - public **virtual** String ToString();
  - protected **virtual** void Finalize();
  - protected Object MemberwiseClone();
  - public static bool Equals(Object objA, Object objB);
  - public static bool ReferenceEquals(Object objA, Object objB);
- Virtual methods can be redefined in your class!
- Equals / GetHashCode
  - assume reference types
  - must be changed (both!) for value types or value semantic

C# Built-in Types

C# Alias	System Type	C# Alias	System.Type
sbyte	System.SByte	**long**	**System.Int64**
byte	**System.Byte**	ulong	System.UInt64
short	**System.Int16**	**char**	**System.Char**
ushort	System.UInt16	**float**	**System.Single**
int	**System.Int32**	**double**	**System.Double**
uint	System.UInt32	**bool**	**System.Boolean**
		decimal	**System.Decimal**
string	**System.String**	**object**	**System.Object**

type CLS compliant type
type non-compliant type: should not be used in public members

Constant Values

- Notations for number constants
  - –integers
    - 1234 (signed integer)
    - 1234U (unsigned integer)
    - 1234UL (long unsigned integer)
    - 1234L (long signed integer)
    - 0x1234 (hexadecimal)
  - –chars
    - 'a'
    - \a escape sequence
    - \x0D hexadecimal escape sequence
    - \u12AB unicode escape sequence
  - –reals
    - 1.0
    - .1
    - 1.0E23
    - 4E3
    - 1.2E+3
    - .4E-12
    - 1.2F (float real)
    - 1.3D (double real)
    - 6E4D
  - –decimals
    - 1.3M
    - 2.6E-2M
  - –strings
    - "abcd"
    - "abc\nad" \n as escape sequence
    - @"c:\temp\" \ are not escaped

Boxing and Unboxing

- Boxing: automatic conversion of a value-type in a reference-type

```
Object o = 25;
```

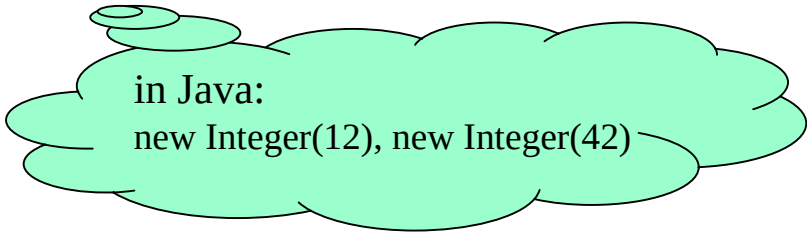
- Unboxing: conversion of a reference-type into a value-type

```
int i = (int)someObject;
```

- Why?
  - all types are really compatible to System.Object
  - simplifies life in generic methods like in

```
System.Console.Write(„Values {0} {1} {2}“, 12, 42, 33);

IList myList = new ArrayList();
myList.add(2);
```



Constants

- C# constants

```
static const int MAX_COUNT = 1234;
```

  - the value must be a constant, no method invocation allowed (exception: new)

- C# readonly fields

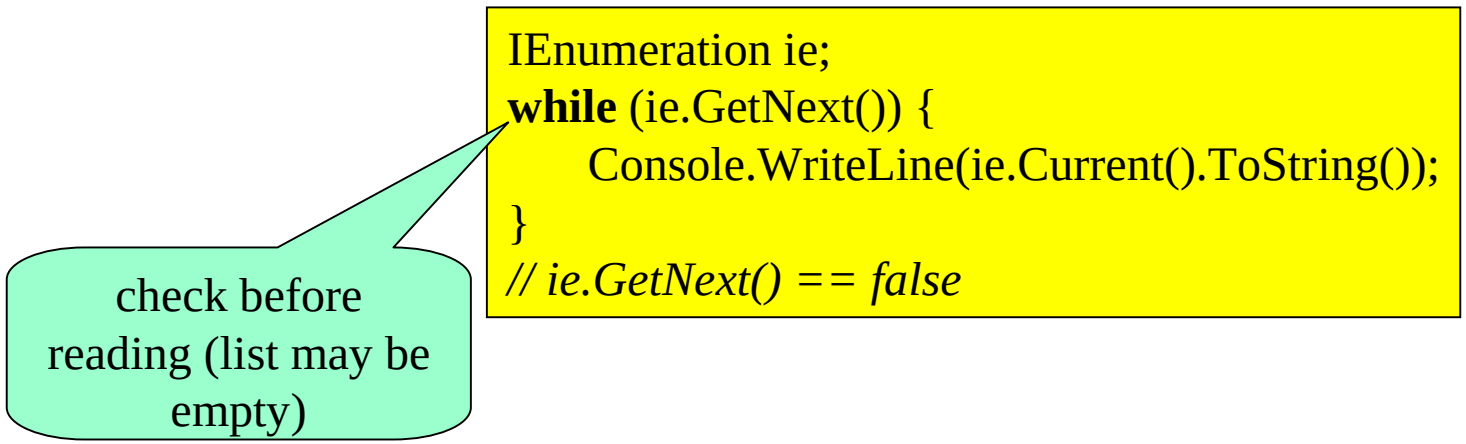
```
static readonly int MAX_COUNT = ReadValue();
```

C# Iteration Constructs

- Loop over data
  - for-loop
    - for(init; condition; increment)
  - foreach
    - Special form for enumerations (IEnumeration)
  - while-loop
    - while (condition) { do something }
  - do while loop
    - do { …something… } while (cond);

C# Iteration Constructs: while

- Iteration with test at the begin of the loop



C# Iteration Constructs: do/while

- Iteration with test after the loop (first pass always executed)

```
do {
      ...do something...
} while (condition);
// !condition is established
```

C# Iteration Constructs: for

- Form:

```
for (init; condition; step) {
        ...       // invariant: condition
}
// !condition
```

- Equivalent to

```
init;
while (!condition) { ...; step; }
//!condition
```

- Special case

```
for (int i= Int32.MinValue; i <= Int32.MaxValue; i++) {
        ... just do it...
}
// i = Int32.MaxValue + 1
```

C# Iteration Constructs: foreach

- Simplify access to Lists, Arrays, ...

```
String[] list = new String[]{„A“, „B“, „C“};
foreach (String s in list) {
    Console.WriteLine(s);
}
```

```
void PrintList(IList list) {
    foreach (String s in list) {
        Console.WriteLine(s);
    }
}
```

will throw InvalidCastException if some element is not a String

dictionaries are enumerated as DictionaryEntry

```
void PrintList(IDictionary dic) {
    foreach (DictionaryEntry de in list) {
        Console.WriteLine(„dict[{0}] = {1}“, de.Key, de.Value);
    }
}
```

C# Control Flow

- Statements to control the execution flow
  - if
  - switch
  - return
  - break
  - continue
  - goto

C# Control Flow: if

- optional execution (only if condition is true)

```
if (condition) {
    ...
}
```

```
if (condition) {
    ...
} else {
    ...
}
```

```
if (condition1) {
    ...
} else if (condition2){
    ...
} else if (condition3) {
    ...
}
```

- else case is optional
- special case for assignments

```
a = (condition) ? true-case : false-case;
```

holds compact code; difficult to read

C# Control Flow: switch

- Switch: choose among many values

numeric and string types are allowed

```
switch (variable) {
    case a:
        // code for (variable == a)
        break;
    case b:
        // code for (variable == b)
        goto case a;
    default:
        ... otherwise ...
}
```

each section **must** end with a jump statement:

- •break leave switch
- •continue switch again
- •default jump to default
- •goto go to
- •return leave method

default is optional; when missing, no code is executed

C# Operators

- Unary
  - `+` positive
  - `-` negative
  - `!` bit not
  - `++x x++` increment
  - `--x x--` decrement
- Multiplicative
  - `*` multiply
  - `/` divide
  - `%` modulo
- Additive
  - `+` addition
  - `-` subtraction
- Shift
  - `<<` shift right
  - `>>` shift left
- Relational
  - `< <= > >=` value tests
  - `is as` type tests
  - `== !=` equality, disequality
- Bit operations
  - `&` logical AND
  - `|` logical OR
  - `^` logical XOR
  - `&&` conditional AND
  - `||` conditional OR
- Assignment
  - `?:` conditional assignment
  - `=` assignment
  - `*= /= %=` assignment
  - `-= +=`
  - `<<= >>=`
  - `&= ^= |=`

C# Access Modifiers

- Access modifiers change the visibility of an entity
  - **public** everybody
  - **private** for this class only
  - **protected** for this class and subclasses only
  - **internal** for this assembly only
  - **protected internal** class, subclasses, and assembly

C# Static Members

- Static members are bound to the class instead of the instance.
  - no self-reference needed
  - state is shared among all instances
- Useful for
  - class-wide constraints
    - Singleton objects
  - class-wide state

```
class T {
    static int calls;       // count invocations

    void M() {
        calls+;             // synchronization?
        ...
    }
}
```

```
class Singleton {
    static Singleton instance;

    private Singleton() {
    }

    public static Singleton Create() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
}
```

C# Parameter Modifiers

- Parameter modifier change the way parameters are passed
  - (none) by value (default)
  - **out** value from method assigned to parameter
  - **ref** by reference
  - **params** open parameter list

```
void M1(int a) ;
```

```
M1(a) ;          // a is not changed, only value passed
```

```
void M2(out int b);
```

```
M2(out a) ;      // a is set
```

```
void M3(ref int c);
```

```
M3(ref a) ;      // a is modified
```

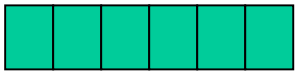
```
void M4(params int[] list);
```

```
M4(list) ;       // open list; values of list can be modified
M4(1, 2, 3, 4) ;
```

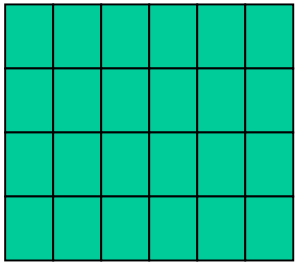
Passing an instance by value, doesn‘t protect its fields against modification!!!

C# Arrays

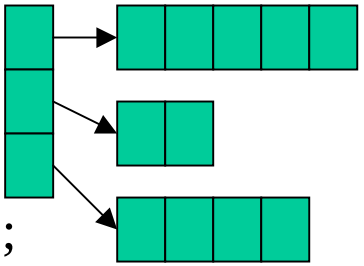
- One-Dimensional Arrays
  - **int[]** intList = **new int[5]**;
  - **String[]** sList = **new String[]{„AG“, „ZH“, „BE“}**;



- Multi-Dimensional Arrays
  - **String[,]** sMat = **new String[9,9]**;
  - **int[,]** matrix = **new int[,] = new int[,]{ {1,2}, {2,1}}**;



- Jagged Arrays
  - **int[][]** jag1 = **new int[3][]**;
    for(int i=0; i < jag1.Length; i++)
      jag1[i] = new int[10-i];
  - **int[][]** jag2 = **new int[][] {new int[]{1}, new int[]{1,2}}**;



System.Array

Clear()	Set values to default
CopyTo()	Copy to another array
GetEnumerator()	Return IEnumerator to traverse all elements
GetLength() **Length**	Return array length
Reverse()	Reverse items
Sort()	Sort one-dimensional array

```
string[] names = new String[]{„Alain“, „Pedro“, „Hannelore“, „Juliet“, „Aki“};
Array.Sort(names);
Array.Reverse(names);
Array.Clear(names, 1, 3);
```

```
Aki, Alain, Hannelore, Juliet, Pedro
Pedro, Juliet, Hannelore, Alain, Aki
Pedro, , , , Aki
```

System.String

Length	String length
Concat()	[static] Concatenate strings
Copy()	[static] Copy a string
Format()	[static] Create a string using a format (just like Console.Write)
PadLeft() **PadRight()**	Insert some characters in a string
Insert() **Remove()** **Replace()**	Modificate a String
ToUpper() **ToLower()**	Uppercase / Lowercase conversion
Join()	[static] Join a String[] in a String
Split()	Split a String in a String[]

System.Text.StringBuilder

- Strings are immutable
  - "abc" + "def" + s + "xyz"
    causes allocation of multiple strings
  - all string operations return a new, different string
- StringBuilder: string buffer for efficient string operations

```
String[] names = new String[]{"Alain", "Pedro", .....};
StringBuilder sb = new StringBuilder(names[0]);
for (int i = 1; i < names.Length; i++) {
      sb.Append(", ");
      sb.Append(names[i]);
}

String result = sb.ToString();
```

C# Enumerations

- Custom numeric type

```
enum Color { red, blue, green }
```

  - compiler enforces type compatibility
  - values can be customized

```
enum Color {
      red = 0x0000FF,
      blue = 0x00FF00,
      green = 0xFF0000
}
```

  - operations allowed (result is not required to be in enumeration)

```
enum Color {
      red = 0x0000FF, blue = 0x00FF00, green = 0xFF0000
      violet = red | blue;
}
```

System.Enum

Format()	Convert to string replesentation
GetName() GetNames()	Return the name of an enumeration constant
GetUnderlyingType()	Underlying type of an enumeration
GetValues()	Enumeration‘s constant values
IsDefined	Check whether value defined in enumeration
Parse	Convert string representation to enum constant

Exercise:

.NET defines two collection types: lists and dictionaries.

All implement either IList or IDictionary

- IList: Array, ArrayList
- IDictionary: SortedList, HashTable, ListDictionary, HybridDictionary

How long does it take to insert, retrieve, and delete items in a collection containing 5, 50, 500 items?