Object-oriented programming #2
Classes

Wojciech Complak

Institute of Computing Science
Faculty of Computing
Poznan University of Technology

e-mail: Wojciech.Complak@wsb.poznan.pl

0.9

1

Lecture content

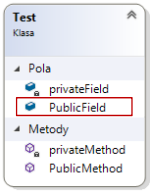
- encapsulation in C#
- static members
- static class
- constructor(s)
- destructor
- constants
- read-only fields
- implementation of encapsulation
- properties
- auto-implemented properties
- restricting access to property/accessor/mutator
- static properties
- recommended order of members in the class
- design patterns, *Singleton* pattern

2

Encapsulation in C# (#1/6)

```
class Test
{
  public int PublicField = 3;
  private int privateField = 5;
  public void PublicMethod()
  {
    this.PublicField++;
    this.privateField++;
    this.privateMethod();
  }
  private void privateMethod()
  {
    this.PublicField++;
    this.privateField++;
    this.privateMethod();
  }
}
```

public field:
- accessible for public and private methods of the class,
- accessible from outside the class



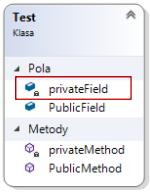
members are sorted alphabetically regardless of their visibility

3

Encapsulation in C# (#2/6)

```
class Test
{
  public int PublicField = 3;
  private int privateField = 5;
  public void PublicMethod()
  {
    this.PublicField++;
    this.privateField++;
    this.privateMethod();
  }
  private void privateMethod()
  {
    this.PublicField++;
    this.privateField++;
    this.privateMethod();
  }
}
```

private field:
- accessible for public and private methods of the class,
- not accessible from outside the class

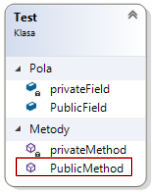


4

Encapsulation in C# (#3/6)

```
class Test
{
  public int PublicField = 3;
  private int privateField = 5;
  public void PublicMethod()
  {
    this.PublicField++;
    this.privateField++;
    this.privateMethod();
  }
  private void privateMethod()
  {
    this.PublicField++;
    this.privateField++;
    this.privateMethod();
  }
}
```

public method:
- accessible for public and private methods of the class,
- accessible from outside the class

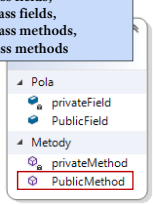


5

Encapsulation in C# (#4/6)

```
class Test
{
  public int PublicField = 3;
  private int privateField = 5;
  public void PublicMethod()
  {
    this.PublicField++;
    this.privateField++;
    this.privateMethod();
  }
  private void privateMethod()
  {
    this.PublicField++;
    this.privateField++;
    this.privateMethod();
  }
}
```

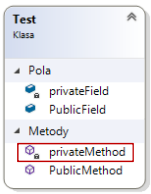
public method can access:
- public class fields,
- private class fields,
- private class methods,
- public class methods



6

Encapsulation in C# (#5/6)

```
class Test
{
  public int PublicField = 3;
  private int privateField = 5;
  public void PublicMethod()
  {
    this.PublicField++;
    this.privateField++;
    this.privateMethod();
  }
  private void privateMethod()
  {
    this.PublicField++;
    this.privateField++;
    this.privateMethod();
  }
}
```



private method:
- accessible for public and private methods of the class,
- not accessible from outside the class

7

Encapsulation in C# (#6/6)

```
class Test
{
  public int PublicField = 3;
  private int privateField = 5;
  public void PublicMethod()
  {
    this.PublicField++;
    this.privateField++;
    this.privateMethod();
  }
  private void privateMethod()
  {
    this.PublicField++;
    this.privateField++;
    this.privateMethod();
  }
}
```

private method can access:
- public class fields
- private class fields
- private class methods
- public class methods

8

Static members (#1/3)

static modifier associates a member with the class, rather than with an object/instance

```
class Test
{
  static private int field1 = 5;
  static public int Field2 = 6;
  private int field3 = 4;
  static private void Method1()
  {
    field1 += Field2;
    // field3++;
    // Method2();
  }
  private void Method2()
  {
    field1 += Field2;
    this.field3++;
    Method1();
  }
}
```

static fields private and public

object/instance field

static method

instance (non-static) method

- static members can be used even when no instance of the class exists,
- static field declaration creates one instance of a variable shared by all class objects,
- static fields are created when the program is started

9

Static members (#2/3)

static modifier associates a member with the class, rather than with an object/instance

```
class Test
{
  static private int field1 = 5;
  static public int Field2 = 6;
  private int field3 = 4;
  static private void Method1()
  {
    field1 += Field2;
    // field3++;
    // Method2();
  }
  private void Method2()
  {
    field1 += Field2;
    this.field3++;
    Method1();
  }
}
```

static fields private and public

object/instance field

static method has access only to static members, it cannot access instance members (because no class object may exist), therefore *this* cannot be used

instance (non-static) method has access to static (they always exist) and non-static members

10

Static members (#3/3)

access to members

```
class Test
{
    static private int field1 = 5;
    static public int Field2 = 6;
    private int field3 = 4;
    static public void Method1() { /* */ }
    public void Method2() { /* */ }
}
// ...
Test.Method1();

Test t = new Test();
t.Method2();
```

use the class to access static members - regardless of whether any class object exists

use the reference to an object of the class to access non-static members

11

Static class

if a class contains only static members (☞ there is no need to keep the object state) it should be a static class (use *static* modifier)

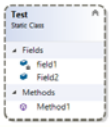
- ☞ an object of a static class cannot be created, a static class cannot be a base class or a derived class,
- ☞ C# requires that in a static class all members must explicitly static – *static* modifier (by default members are non-static),
- ☞ static classes (utility classes) are e. g. *Math* and *Console*

```
static class Test
{
  static private int field1 = 5;
  static public int Field2 = 6;
  static public void Method1()
  {
    field1 += Field2;
  }
}
// ...
Test.Method1();

Test t = new Test();
```

calling a static method from a static class

an object of a static class cannot be created



12

Creating objects: constructor

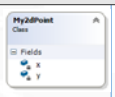
- constructor = method with the same name as the class (C++/Java/C#, in Delphi the method marked with the *Constructor* keyword),
- the constructor is automatically started when creating the object (instance of the class) to initialize the object (what the constructor exactly does depends on the language and implementation),
- the task of the constructor is to make the object usable,
- a class can have any number of constructors,
- the constructor can have any set of parameters (parameterless = default),
- the constructor cannot return a result,
- the constructor signals errors by throwing exceptions

13

Default constructor: automatically generated (#1/3)

in C# (and Java), a "default constructor" refers to a nullary public constructor automatically generated by the compiler if no constructors have been defined for the class. The default constructor implicitly calls the superclass's nullary constructor and initialises fields to default values.

```
class My2dPoint
{
  int x, y;
}
// ...
My2dPoint t = new My2dPoint();
```



calling the default constructor

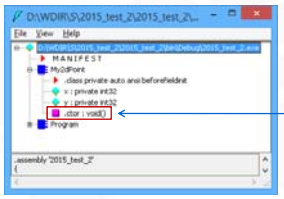
☞ in the absence of a public default constructor, it wouldn't be possible to directly create class objects

14

Default constructor: automatically generated (#2/3)

```
class My2dPoint
{
  int x, y;
}
```

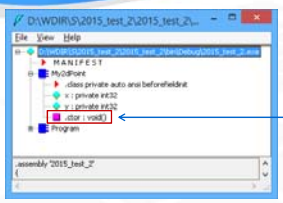
the class diagram does NOT show the automatically generated default constructor



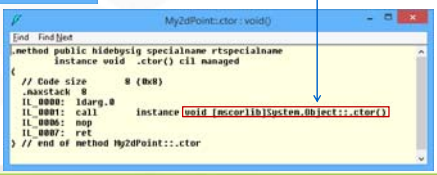
but in the generated CIL you can see it (use ILDasm or ILSpy)

15

Default constructor: automatically generated (#3/3)



every instance constructor calls the base class (nullary by default) constructor (in this case *Object* class)



```
My2dPoint::.ctor : void()
.method public hidebysig specialname rtspecialname
        instance void .ctor() cil managed
{
  // Code size       8 (0x8)
  .maxstack  8
  IL_0000:  ldarg.0
  IL_0001:  call       instance void [mscorlib]System.Object::.ctor()
  IL_0006:  nop
  IL_0007:  ret
} // end of method My2dPoint::.ctor
```


16

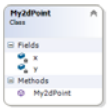
Default constructor: explicit (#1/2)

non-default constructor

```
class My2dPoint
{
  int x, y;
  public My2dPoint()
  {
    x = y = 0;
  }
}
// ...
My2dPoint t = new My2dPoint();
```

default contructor implementation

invocation of the default constructor

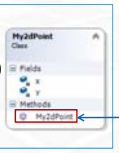


☞ if the default constructor is the only implemented, it must be *public*, otherwise (*private*) you will not be able to create objects of this class,
☞ defining a *private* default constructor (it is recommended to explicitly use the *private* attribute) is useful in classes that have only static fields and objects of this class are NOT to be created

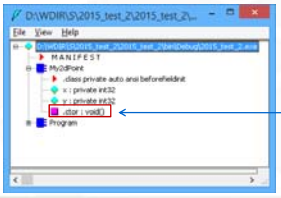
17

Default constructor: explicit (#2/2)

```
class My2dPoint
{
  int x, y;
  public My2dPoint()
  {
    x = y = 0;
  }
}
```



the class diagram DOES show the explicitly implemented default constructor



and so it is in CIL

18

Constructors (#1/7)

non-default contructor

```
class My2dPoint
{
  int x, y;
  public My2dPoint(int newX, int newY)
  {
    x = newX;
    y = newY;
  }
}
// ...
My2dPoint t1 = new My2dPoint();
My2dPoint t2 = new My2dPoint(1,2);
```

constructor implementation

it's not possible any more – if the programmer defines any custom constructor then C# does not automatically generate a default constructor

calling non-default constructor

19

Constructors (#2/7)

you can define as many constructors as you like - so that you can conveniently create class objects

```
class My2dPoint
{
  int x, y;
  public My2dPoint()
  {
    x = y = 0;
  }
  public My2dPoint(int newX, int newY)
  {
    x = newX;
    y = newY;
  }
}
// ...
My2dPoint t1 = new My2dPoint();
My2dPoint t2 = new My2dPoint(1,2);
```

2 constructors, the default one is public (if the default constructor is not public, a security padlock will be shown)

so now you can...

and also

20

Constructors (#3/7)

an alternative to duplicating constructors is to use the default and named arguments (the constructor is also a method!)

```
class My2dPoint
{
  int x, y;
  public My2dPoint( int newX = 0 , int newY = 0 )
  {
    x = newX;
    y = newY;
  }
}
// ...
My2dPoint p1 = new My2dPoint();                  → x:0,y:0  both default
My2dPoint p2 = new My2dPoint(5);                 → x:5,y:0  x: explicit, y:default
My2dPoint p3 = new My2dPoint(4,5);               → x:4,y:5  both explicit
My2dPoint p4 = new My2dPoint(newY: 5);           → x:0,y:5  y: named, x: default
My2dPoint p5 = new My2dPoint(newX: 4, newY: 5);  → x:4,y:5  both named
My2dPoint p6 = new My2dPoint(newY: 4, newX: 5);  → x:5,y:4  both named
My2dPoint p7 = new My2dPoint(4, newY: 5);        → x:4,y:5  x by position, y: named
```

named argument must appear after all passed by position arguments

21

Constructors (#4/7)

what if the default constructor is also implemented?

```
class My2dPoint
{
  int x, y;
  public My2dPoint()
  {
    x = y = 0;
  }
  public My2dPoint( int newX = 0 , int newY = 0 )
  {
    x = newX;
    y = newY;
  }
}
// ...
My2dPoint p1 = new My2dPoint();
```

which ctor will be launched?

default – in case of ambiguity, this with the same number of explicit parameters are preferred

22

Constructors (#5/7)

one constructor can be called from another – to reuse code

```
class My2dPoint
{
  int x, y;
  public My2dPoint() : this(0,0) { }
  public My2dPoint(int newX, int newY)
  {
    x = newX;
    y = newY;
  }
}
// ...
My2dPoint t1 = new My2dPoint();
My2dPoint t2 = new My2dPoint(1,2);
```

order of ctor invocation

C# does not allow calling other ctor from the body of another ctor

still valid ...

this way too

23

Constructors (#6/7)

a ctor can be *private* - it can still be invoked by other constructors but it cannot be used directly to create an instance of the class

```
class My2dPoint
{
  int x, y;
  public My2dPoint() : this(0,0) { }
  private My2dPoint(int newX, int newY)
  {
    x = newX;
    y = newY;
  }
}
// ...
My2dPoint t1 = new My2dPoint();
My2dPoint t2 = new My2dPoint(1,2);
```

so this is valid ...

and this is not ...

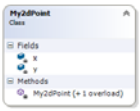
24

Constructors (#7/7)

chaining ctors calls can be used to simplify the code and validate data (but you have to remember about the fixed order of execution)

```
class My2dPoint
{
  int x, y;
  public My2dPoint() : this(0,0)
  {
    Console.WriteLine("Public constructor");
  }
  private My2dPoint(int newX, int newY)
  {
    Console.WriteLine("Private constructor");
    x = newX;
    y = newY;
  }
}
// ...
My2dPoint t1 = new My2dPoint();
```

Private constructor
Public constructor



this was not meant to be - the public ctor was to check the parameters and then call the private ctor