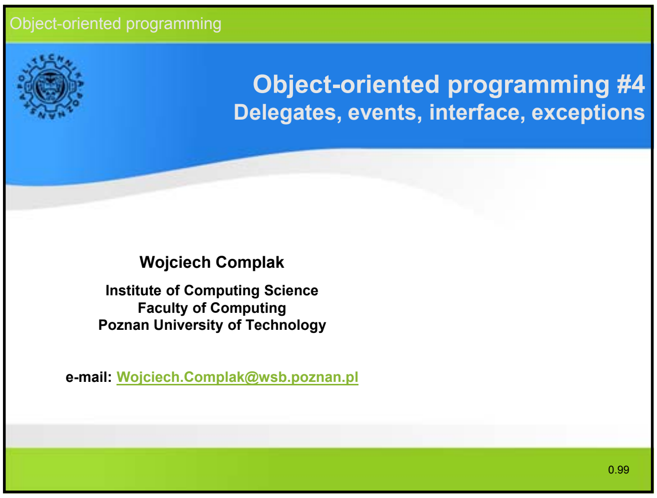


Object-oriented programming

Object-oriented programming #4
Delegates, events, interface, exceptions

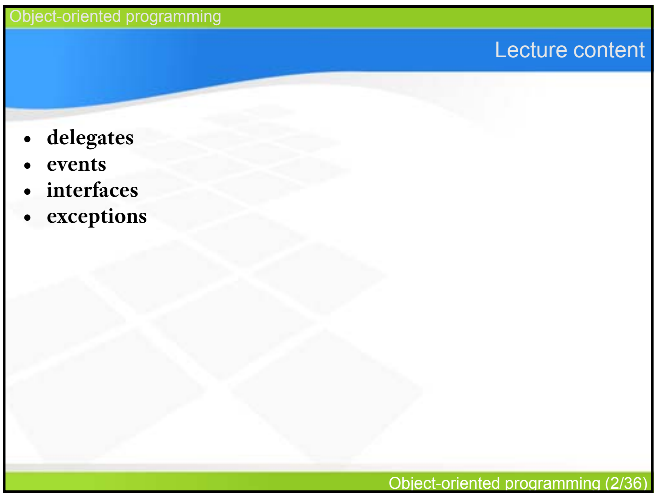
Wojciech Complak

Institute of Computing Science
Faculty of Computing
Poznan University of Technology

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

0.99

1



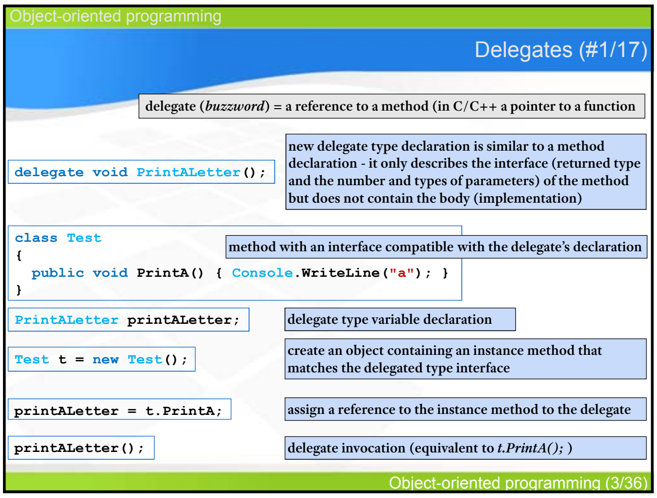
Object-oriented programming

Lecture content

- **delegates**
- **events**
- **interfaces**
- **exceptions**

Object-oriented programming (2/36)

2



Object-oriented programming

Delegates (#1/17)

delegate (*buzzword*) = a reference to a method (in C/C++ a pointer to a function

```
delegate void PrintALetter();
```

new delegate type declaration is similar to a method declaration - it only describes the interface (returned type and the number and types of parameters) of the method but does not contain the body (implementation)

```
class Test
{
  public void PrintA() { Console.WriteLine("a"); }
}
```

method with an interface compatible with the delegate's declaration

```
PrintALetter printALetter;
```

delegate type variable declaration

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

create an object containing an instance method that matches the delegated type interface

```
printALetter = t.PrintA;
```

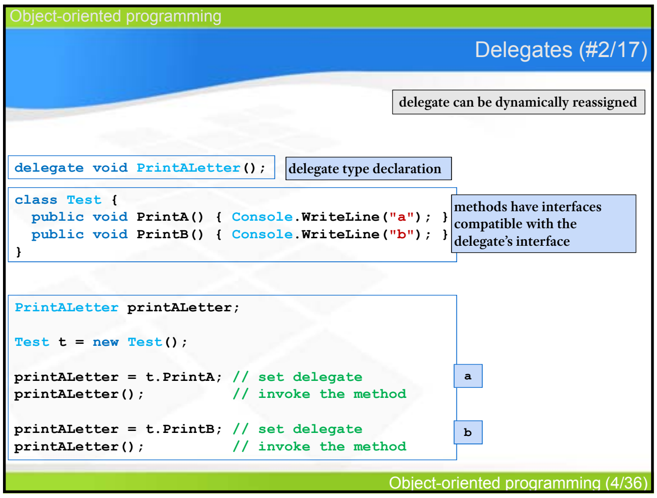
assign a reference to the instance method to the delegate

```
printALetter();
```

delegate invocation (equivalent to *t.PrintA(); *)

Object-oriented programming (3/36)

3



Object-oriented programming

Delegates (#2/17)

delegate can be dynamically reassigned

```
delegate void PrintALetter();
```

delegate type declaration

```
class Test {
  public void PrintA() { Console.WriteLine("a"); }
  public void PrintB() { Console.WriteLine("b"); }
}
```

methods have interfaces compatible with the delegate's interface

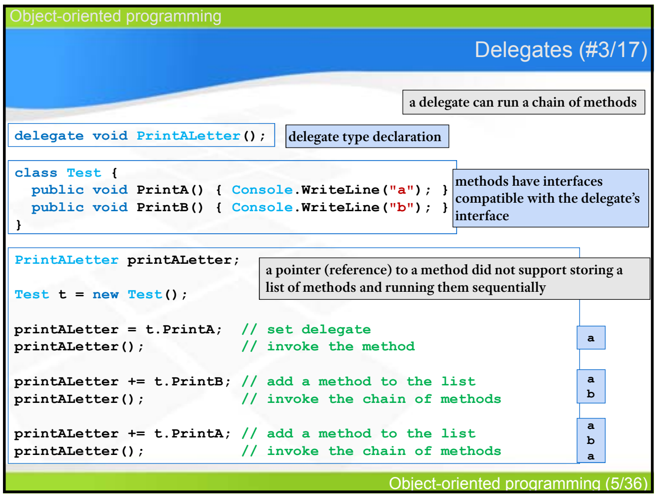
```
PrintALetter printALetter;

Test t = new Test();

printALetter = t.PrintA; // set delegate
printALetter();          // invoke the method

printALetter = t.PrintB; // set delegate
printALetter();          // invoke the method
```

a

b

Object-oriented programming (4/36)

4



Object-oriented programming

Delegates (#3/17)

a delegate can run a chain of methods

```
delegate void PrintALetter();
```

delegate type declaration

```
class Test {
  public void PrintA() { Console.WriteLine("a"); }
  public void PrintB() { Console.WriteLine("b"); }
}
```

methods have interfaces compatible with the delegate's interface

```
PrintALetter printALetter;

Test t = new Test();

printALetter = t.PrintA;  // set delegate
printALetter();           // invoke the method

printALetter += t.PrintB; // add a method to the list
printALetter();           // invoke the chain of methods

printALetter += t.PrintA; // add a method to the list
printALetter();           // invoke the chain of methods
```

a pointer (reference) to a method did not support storing a list of methods and running them sequentially

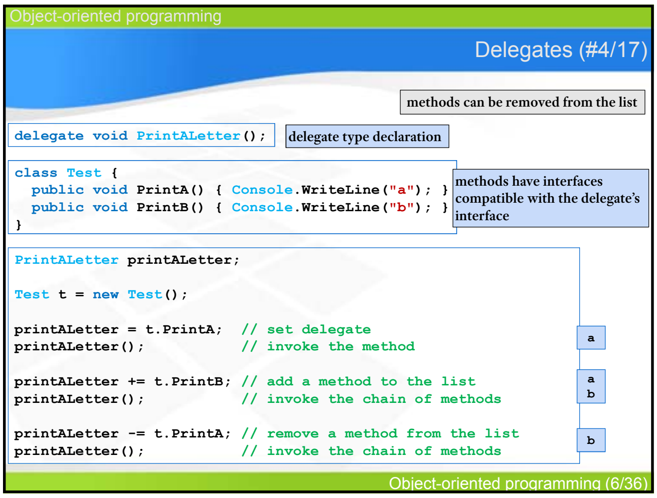
a

a
b

a
b
a

Object-oriented programming (5/36)

5



Object-oriented programming

Delegates (#4/17)

methods can be removed from the list

```
delegate void PrintALetter();
```

delegate type declaration

```
class Test {
  public void PrintA() { Console.WriteLine("a"); }
  public void PrintB() { Console.WriteLine("b"); }
}
```

methods have interfaces compatible with the delegate's interface

```
PrintALetter printALetter;

Test t = new Test();

printALetter = t.PrintA;  // set delegate
printALetter();           // invoke the method

printALetter += t.PrintB; // add a method to the list
printALetter();           // invoke the chain of methods

printALetter -= t.PrintA; // remove a method from the list
printALetter();           // invoke the chain of methods
```

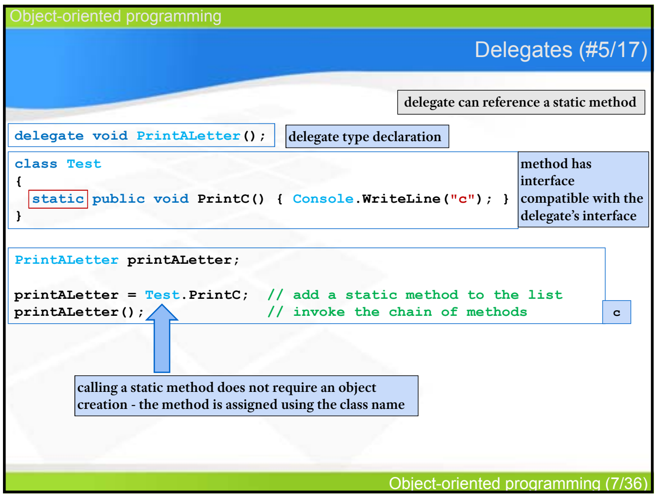
a

a
b

b

Object-oriented programming (6/36)

6



Object-oriented programming

Delegates (#5/17)

delegate can reference a static method

```
delegate void PrintALetter();
```

delegate type declaration

```
class Test
{
  static public void PrintC() { Console.WriteLine("c"); }
}
```

method has interface compatible with the delegate's interface

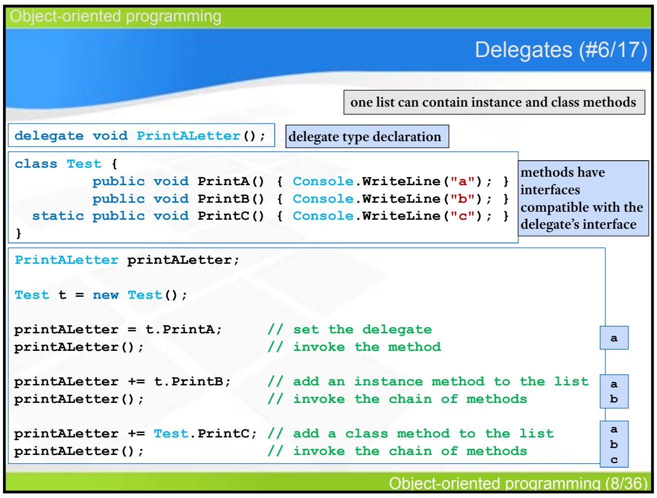
```
PrintALetter printALetter;

printALetter = Test.PrintC;  // add a static method to the list
printALetter();              // invoke the chain of methods
```

```
c
```

calling a static method does not require an object creation - the method is assigned using the class name

Object-oriented programming (7/36)

7



Object-oriented programming

Delegates (#6/17)

one list can contain instance and class methods

```
delegate void PrintALetter();
```

delegate type declaration

```
class Test {
         public void PrintA() { Console.WriteLine("a"); }
         public void PrintB() { Console.WriteLine("b"); }
  static public void PrintC() { Console.WriteLine("c"); }
}
```

methods have interfaces compatible with the delegate's interface

```
PrintALetter printALetter;

Test t = new Test();

printALetter = t.PrintA;     // set the delegate
printALetter();              // invoke the method

printALetter += t.PrintB;    // add an instance method to the list
printALetter();              // invoke the chain of methods

printALetter += Test.PrintC; // add a class method to the list
printALetter();              // invoke the chain of methods
```

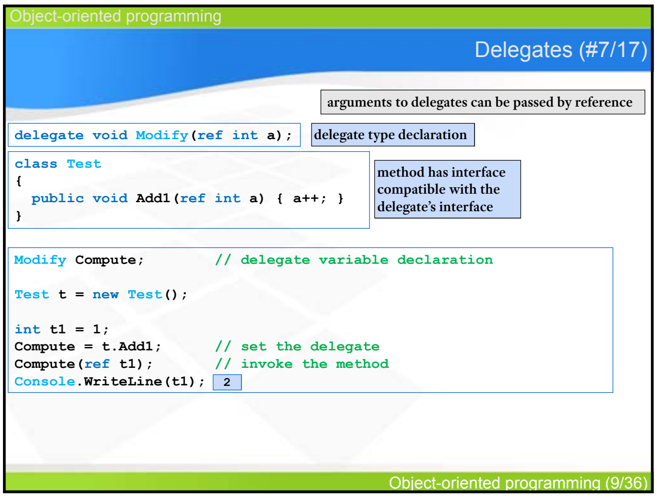
```
a
```

```
a
b
```

```
a
b
c
```

Object-oriented programming (8/36)

8



Object-oriented programming

Delegates (#7/17)

arguments to delegates can be passed by reference

```
delegate void Modify(ref int a);
```

delegate type declaration

```
class Test
{
  public void Add1(ref int a) { a++; }
}
```

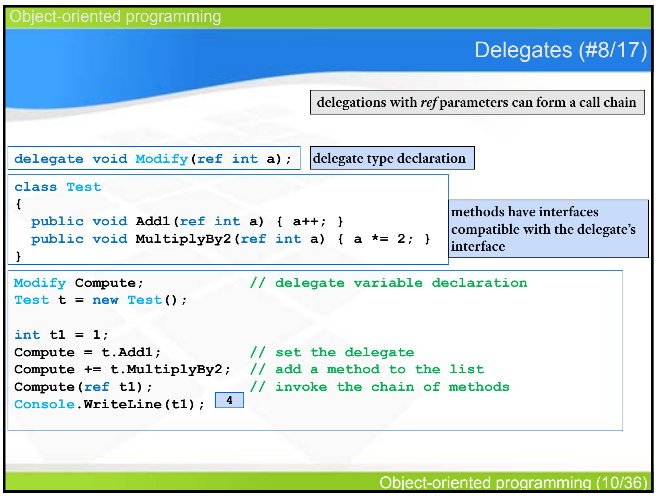
method has interface compatible with the delegate's interface

```
Modify Compute;         // delegate variable declaration

Test t = new Test();

int t1 = 1;
Compute = t.Add1;       // set the delegate
Compute(ref t1);        // invoke the method
Console.WriteLine(t1);
```

```
2
```

Object-oriented programming (9/36)

9



Object-oriented programming

Delegates (#8/17)

delegations with *ref* parameters can form a call chain

```
delegate void Modify(ref int a);
```

delegate type declaration

```
class Test
{
  public void Add1(ref int a) { a++; }
  public void MultiplyBy2(ref int a) { a *= 2; }
}
```

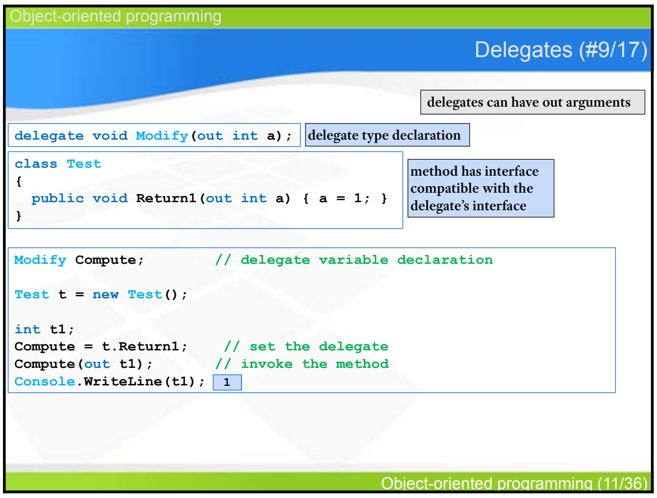
methods have interfaces compatible with the delegate's interface

```
Modify Compute;                // delegate variable declaration
Test t = new Test();

int t1 = 1;
Compute = t.Add1;              // set the delegate
Compute += t.MultiplyBy2;      // add a method to the list
Compute(ref t1);               // invoke the chain of methods
Console.WriteLine(t1);
```

```
4
```

Object-oriented programming (10/36)

10



Object-oriented programming

Delegates (#9/17)

delegates can have out arguments

```
delegate void Modify(out int a);
```

delegate type declaration

```
class Test
{
  public void Return1(out int a) { a = 1; }
}
```

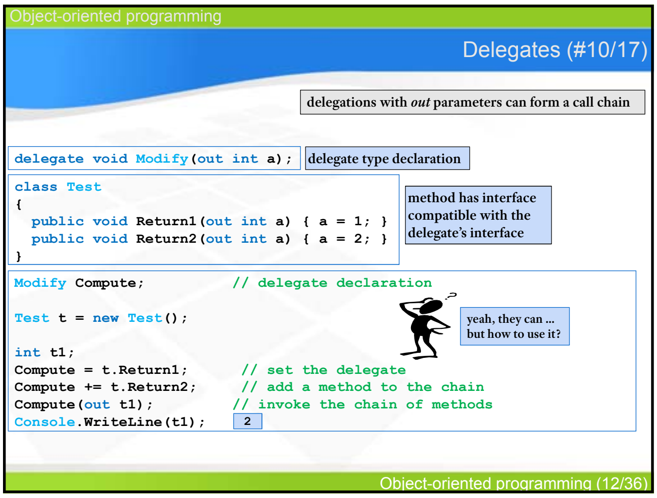
method has interface compatible with the delegate's interface

```
Modify Compute;         // delegate variable declaration

Test t = new Test();

int t1;
Compute = t.Return1;     // set the delegate
Compute(out t1);        // invoke the method
Console.WriteLine(t1);
```

```
1
```

Object-oriented programming (11/36)

11



Object-oriented programming

Delegates (#10/17)

delegations with *out* parameters can form a call chain

```
delegate void Modify(out int a);
```

delegate type declaration

```
class Test
{
  public void Return1(out int a) { a = 1; }
  public void Return2(out int a) { a = 2; }
}
```

method has interface compatible with the delegate's interface

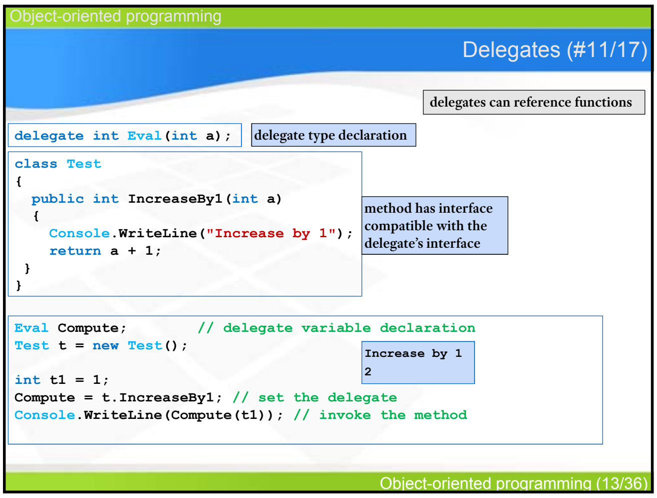
```
Modify Compute;           // delegate declaration

Test t = new Test();

int t1;
Compute = t.Return1;       // set the delegate
Compute += t.Return2;      // add a method to the chain
Compute(out t1);          // invoke the chain of methods
Console.WriteLine(t1);
```

```
2
```

yeah, they can ...
but how to use it?

Object-oriented programming (12/36)

12



Object-oriented programming

Delegates (#11/17)

delegates can reference functions

```
delegate int Eval(int a);
```

delegate type declaration

```
class Test
{
  public int IncreaseBy1(int a)
  {
    Console.WriteLine("Increase by 1");
    return a + 1;
 }
}
```

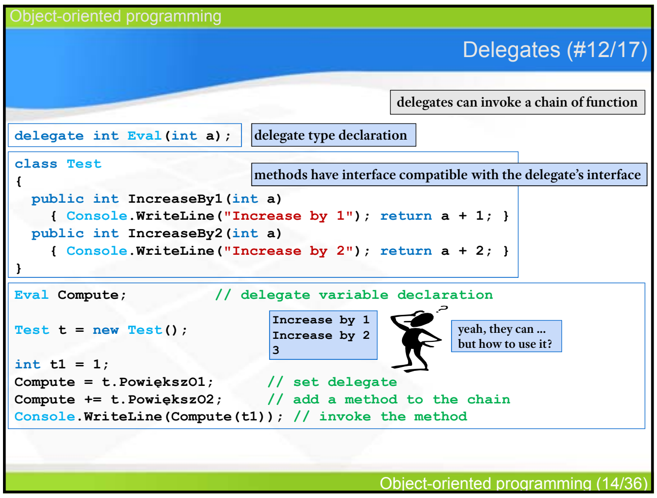
method has interface compatible with the delegate's interface

```
Eval Compute;          // delegate variable declaration
Test t = new Test();

int t1 = 1;
Compute = t.IncreaseBy1; // set the delegate
Console.WriteLine(Compute(t1)); // invoke the method
```

```
Increase by 1
2
```

Object-oriented programming (13/36)

13



Object-oriented programming

Delegates (#12/17)

delegates can invoke a chain of function

```
delegate int Eval(int a);
```

delegate type declaration

```
class Test
{
  public int IncreaseBy1(int a)
    { Console.WriteLine("Increase by 1"); return a + 1; }
  public int IncreaseBy2(int a)
    { Console.WriteLine("Increase by 2"); return a + 2; }
}
```

methods have interface compatible with the delegate's interface

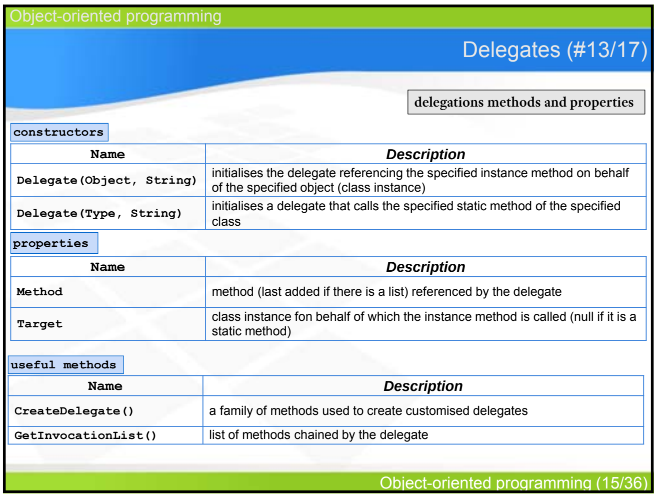
```
Eval Compute;             // delegate variable declaration

Test t = new Test();

int t1 = 1;
Compute = t.PowiększO1;         // set delegate
Compute += t.PowiększO2;        // add a method to the chain
Console.WriteLine(Compute(t1)); // invoke the method
```

```
Increase by 1
Increase by 2
3
```

yeah, they can ...
but how to use it?

Object-oriented programming (14/36)

14



Object-oriented programming

Delegates (#13/17)

delegations methods and properties

constructors

Name	*Description*
`Delegate(Object, String)`	initialises the delegate referencing the specified instance method on behalf of the specified object (class instance)
`Delegate(Type, String)`	initialises a delegate that calls the specified static method of the specified class

properties

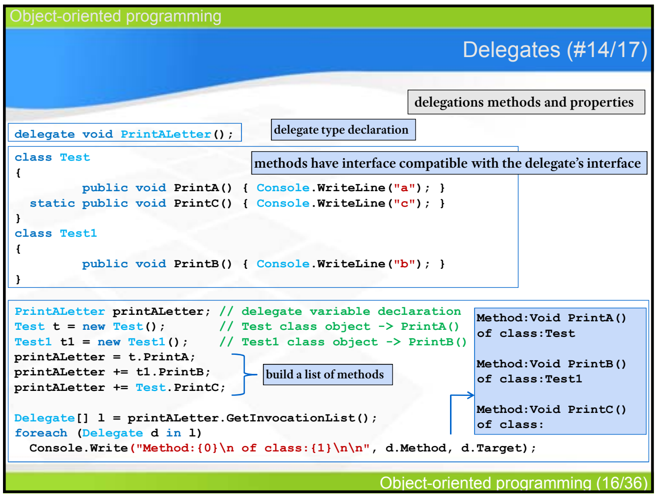
Name	*Description*
`Method`	method (last added if there is a list) referenced by the delegate
`Target`	class instance fon behalf of which the instance method is called (null if it is a static method)

useful methods

Name	*Description*
`CreateDelegate()`	a family of methods used to create customised delegates
`GetInvocationList()`	list of methods chained by the delegate

Object-oriented programming (15/36)

15



Object-oriented programming

Delegates (#14/17)

delegations methods and properties

```
delegate void PrintALetter();
```

delegate type declaration

```
class Test
{
        public void PrintA() { Console.WriteLine("a"); }
  static public void PrintC() { Console.WriteLine("c"); }
}
class Test1
{
        public void PrintB() { Console.WriteLine("b"); }
}
```

methods have interface compatible with the delegate's interface

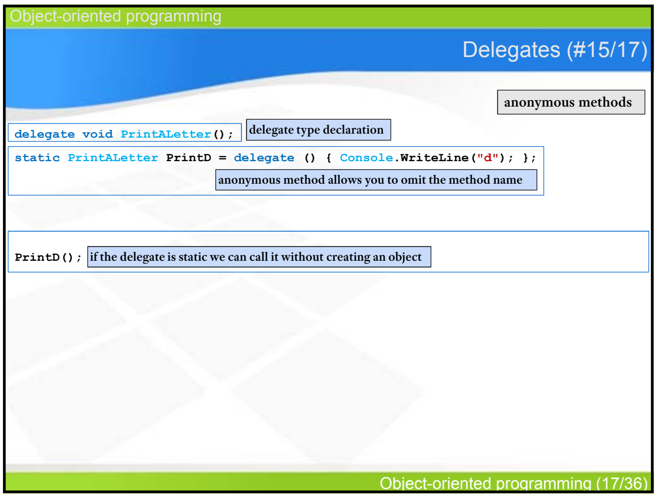
```
PrintALetter printALetter; // delegate variable declaration
Test t = new Test();       // Test class object -> PrintA()
Test1 t1 = new Test1();    // Test1 class object -> PrintB()
printALetter = t.PrintA;
printALetter += t1.PrintB;
printALetter += Test.PrintC;

Delegate[] l = printALetter.GetInvocationList();
foreach (Delegate d in l)
  Console.Write("Method:{0}\n of class:{1}\n\n", d.Method, d.Target);
```

build a list of methods

```
Method:Void PrintA()
of class:Test

Method:Void PrintB()
of class:Test1

Method:Void PrintC()
of class:
```

Object-oriented programming (16/36)

16



Object-oriented programming

Delegates (#15/17)

anonymous methods

```
delegate void PrintALetter();
```

delegate type declaration

```
static PrintALetter PrintD = delegate () { Console.WriteLine("d"); };
```

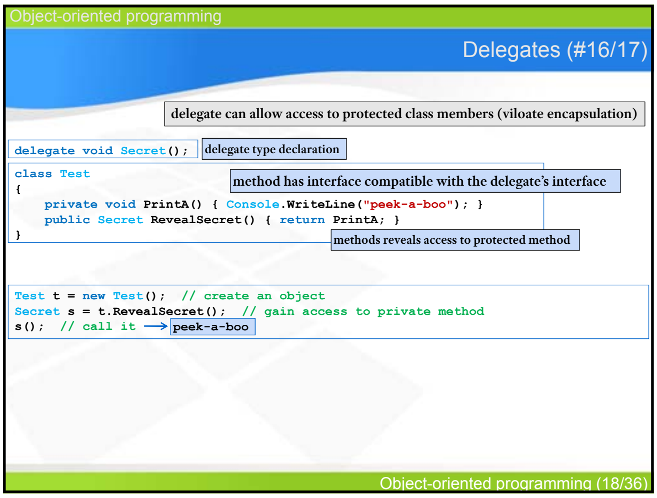
anonymous method allows you to omit the method name

```
PrintD();
```

if the delegate is static we can call it without creating an object

Object-oriented programming (17/36)

17



Object-oriented programming

Delegates (#16/17)

delegate can allow access to protected class members (viloate encapsulation)

```
delegate void Secret();
```

delegate type declaration

```
class Test
{
    private void PrintA() { Console.WriteLine("peek-a-boo"); }
    public Secret RevealSecret() { return PrintA; }
}
```

method has interface compatible with the delegate's interface

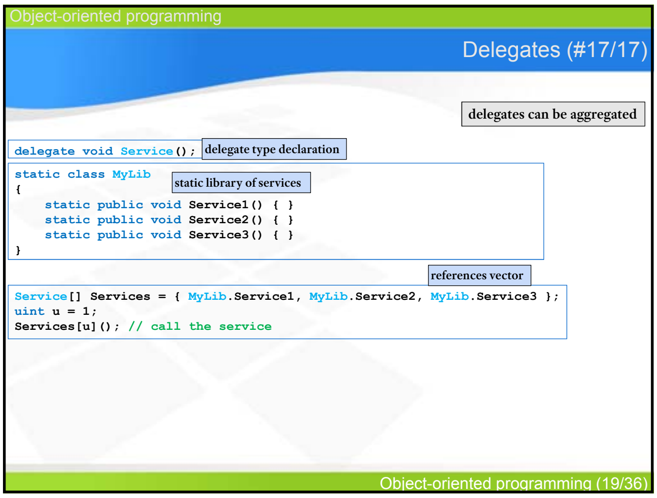
methods reveals access to protected method

```
Test t = new Test();  // create an object
Secret s = t.RevealSecret();  // gain access to private method
s();  // call it
```

```
peek-a-boo
```

Object-oriented programming (18/36)

18



Object-oriented programming

Delegates (#17/17)

delegates can be aggregated

```
delegate void Service();
```

delegate type declaration

```
static class MyLib
{
    static public void Service1() { }
    static public void Service2() { }
    static public void Service3() { }
}
```

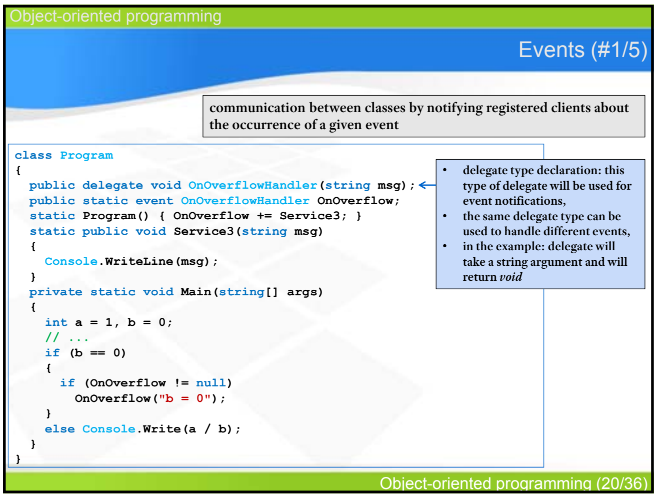
static library of services

references vector

```
Service[] Services = { MyLib.Service1, MyLib.Service2, MyLib.Service3 };
uint u = 1;
Services[u](); // call the service
```

Object-oriented programming (19/36)

19



Object-oriented programming

Events (#1/5)

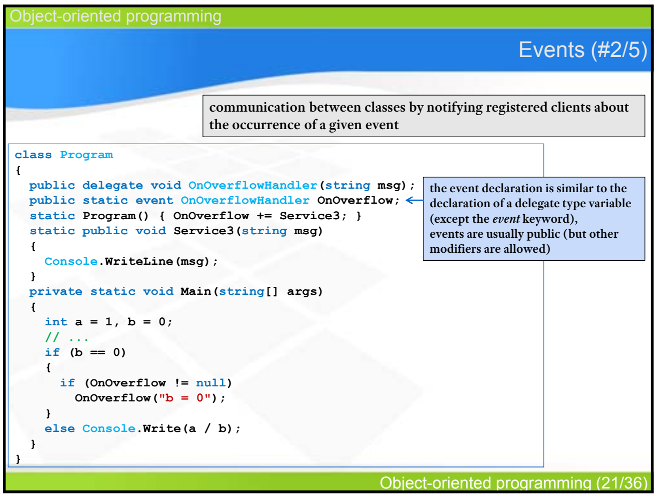
communication between classes by notifying registered clients about the occurrence of a given event

```
class Program
{
  public delegate void OnOverflowHandler(string msg);
  public static event OnOverflowHandler OnOverflow;
  static Program() { OnOverflow += Service3; }
  static public void Service3(string msg)
  {
    Console.WriteLine(msg);
  }
  private static void Main(string[] args)
  {
    int a = 1, b = 0;
    // ...
    if (b == 0)
    {
      if (OnOverflow != null)
        OnOverflow("b = 0");
    }
    else Console.Write(a / b);
  }
}
```

- delegate type declaration: this type of delegate will be used for event notifications,
- the same delegate type can be used to handle different events,
- in the example: delegate will take a string argument and will return *void*

Object-oriented programming (20/36)

20



Object-oriented programming

Events (#2/5)

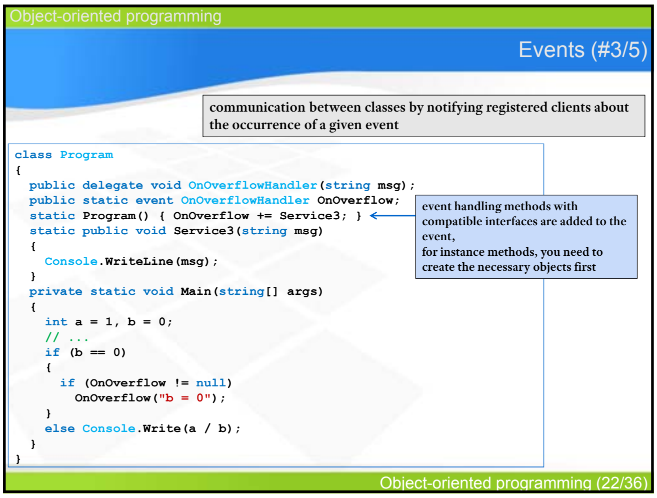
communication between classes by notifying registered clients about the occurrence of a given event

```
class Program
{
  public delegate void OnOverflowHandler(string msg);
  public static event OnOverflowHandler OnOverflow;
  static Program() { OnOverflow += Service3; }
  static public void Service3(string msg)
  {
    Console.WriteLine(msg);
  }
  private static void Main(string[] args)
  {
    int a = 1, b = 0;
    // ...
    if (b == 0)
    {
      if (OnOverflow != null)
        OnOverflow("b = 0");
    }
    else Console.Write(a / b);
  }
}
```

the event declaration is similar to the declaration of a delegate type variable (except the *event* keyword), events are usually public (but other modifiers are allowed)

Object-oriented programming (21/36)

21



Object-oriented programming

Events (#3/5)

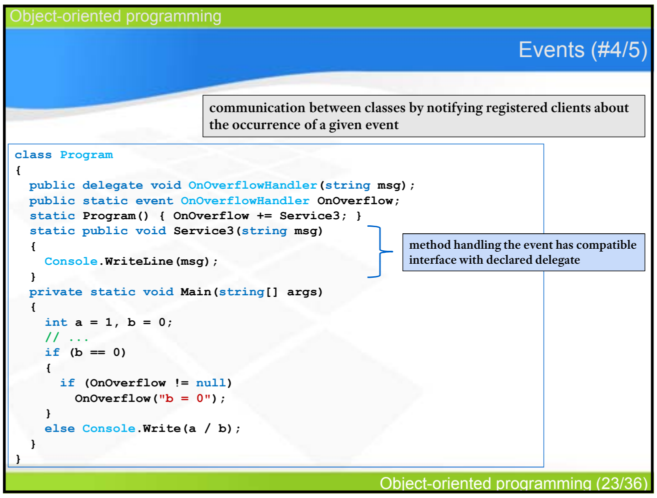
communication between classes by notifying registered clients about the occurrence of a given event

```
class Program
{
  public delegate void OnOverflowHandler(string msg);
  public static event OnOverflowHandler OnOverflow;
  static Program() { OnOverflow += Service3; }
  static public void Service3(string msg)
  {
    Console.WriteLine(msg);
  }
  private static void Main(string[] args)
  {
    int a = 1, b = 0;
    // ...
    if (b == 0)
    {
      if (OnOverflow != null)
        OnOverflow("b = 0");
    }
    else Console.Write(a / b);
  }
}
```

event handling methods with compatible interfaces are added to the event,
for instance methods, you need to create the necessary objects first

Object-oriented programming (22/36)

22



Object-oriented programming

Events (#4/5)

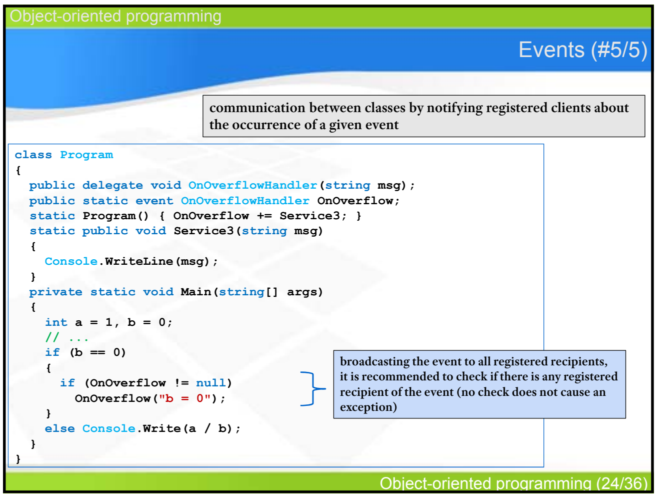
communication between classes by notifying registered clients about the occurrence of a given event

```
class Program
{
  public delegate void OnOverflowHandler(string msg);
  public static event OnOverflowHandler OnOverflow;
  static Program() { OnOverflow += Service3; }
  static public void Service3(string msg)
  {
    Console.WriteLine(msg);
  }
  private static void Main(string[] args)
  {
    int a = 1, b = 0;
    // ...
    if (b == 0)
    {
      if (OnOverflow != null)
        OnOverflow("b = 0");
    }
    else Console.Write(a / b);
  }
}
```

method handling the event has compatible interface with declared delegate

Object-oriented programming (23/36)

23



Object-oriented programming

Events (#5/5)

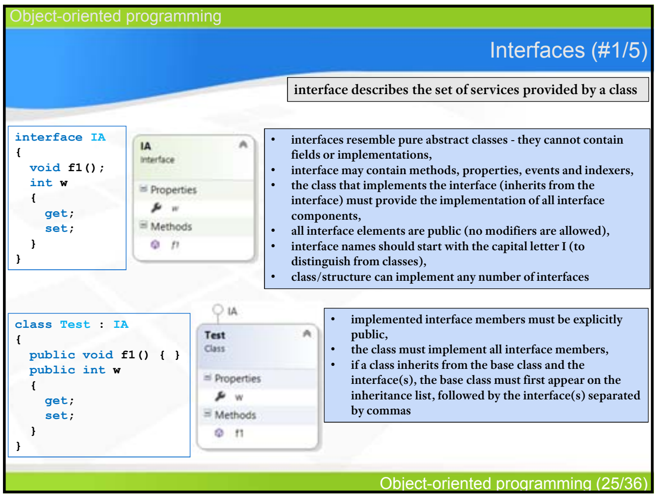
communication between classes by notifying registered clients about the occurrence of a given event

```
class Program
{
  public delegate void OnOverflowHandler(string msg);
  public static event OnOverflowHandler OnOverflow;
  static Program() { OnOverflow += Service3; }
  static public void Service3(string msg)
  {
    Console.WriteLine(msg);
  }
  private static void Main(string[] args)
  {
    int a = 1, b = 0;
    // ...
    if (b == 0)
    {
      if (OnOverflow != null)
        OnOverflow("b = 0");
    }
    else Console.Write(a / b);
  }
}
```

broadcasting the event to all registered recipients, it is recommended to check if there is any registered recipient of the event (no check does not cause an exception)

Object-oriented programming (24/36)

24



Object-oriented programming

Interfaces (#1/5)

interface describes the set of services provided by a class

```
interface IA
{
  void f1();
  int w
  {
    get;
    set;
  }
}
```

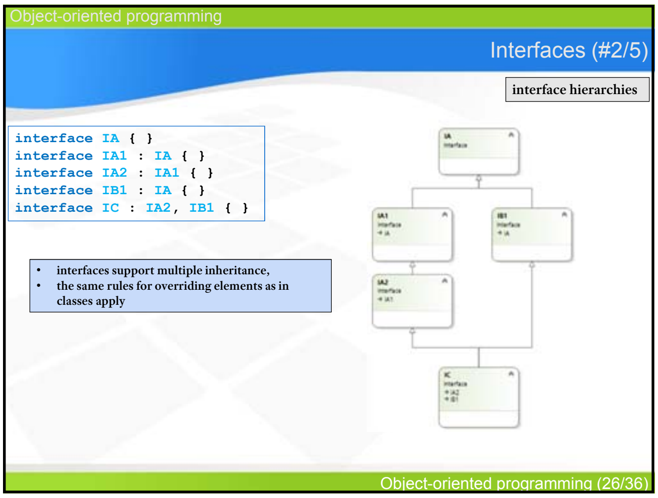
- interfaces resemble pure abstract classes - they cannot contain fields or implementations,
- interface may contain methods, properties, events and indexers,
- the class that implements the interface (inherits from the interface) must provide the implementation of all interface components,
- all interface elements are public (no modifiers are allowed),
- interface names should start with the capital letter I (to distinguish from classes),
- class/structure can implement any number of interfaces

```
class Test : IA
{
  public void f1() { }
  public int w
  {
    get;
    set;
  }
}
```

- implemented interface members must be explicitly public,
- the class must implement all interface members,
- if a class inherits from the base class and the interface(s), the base class must first appear on the inheritance list, followed by the interface(s) separated by commas

Object-oriented programming (25/36)

25



Object-oriented programming

Interfaces (#2/5)

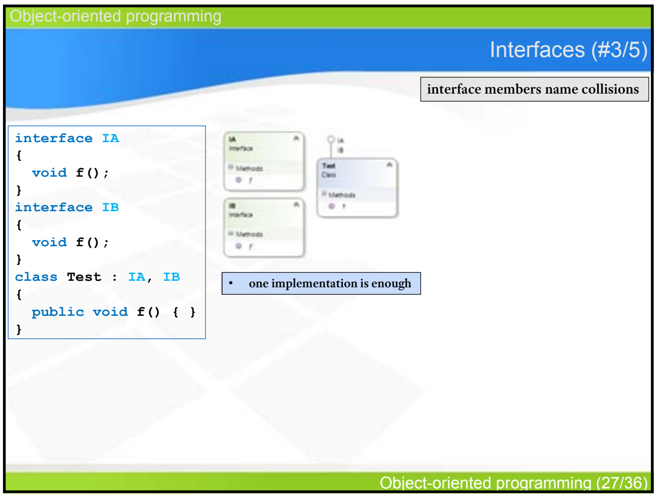
interface hierarchies

```
interface IA { }
interface IA1 : IA { }
interface IA2 : IA1 { }
interface IB1 : IA { }
interface IC : IA2, IB1 { }
```

- interfaces support multiple inheritance,
- the same rules for overriding elements as in classes apply

Object-oriented programming (26/36)

26



Object-oriented programming

Interfaces (#3/5)

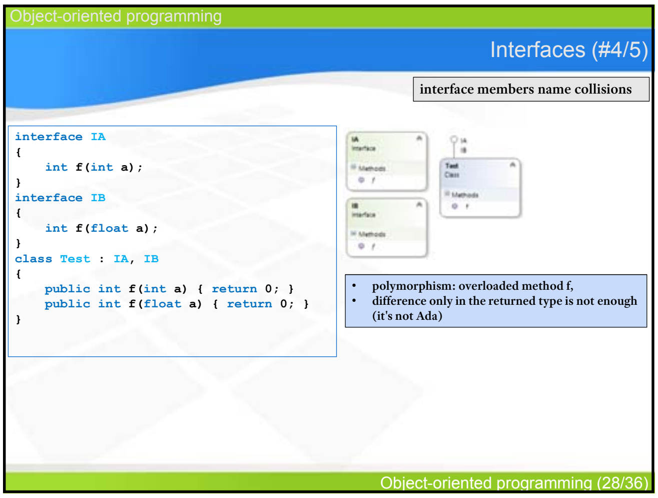
interface members name collisions

```
interface IA
{
  void f();
}
interface IB
{
  void f();
}
class Test : IA, IB
{
  public void f() { }
}
```

- one implementation is enough

Object-oriented programming (27/36)

27



Object-oriented programming

Interfaces (#4/5)

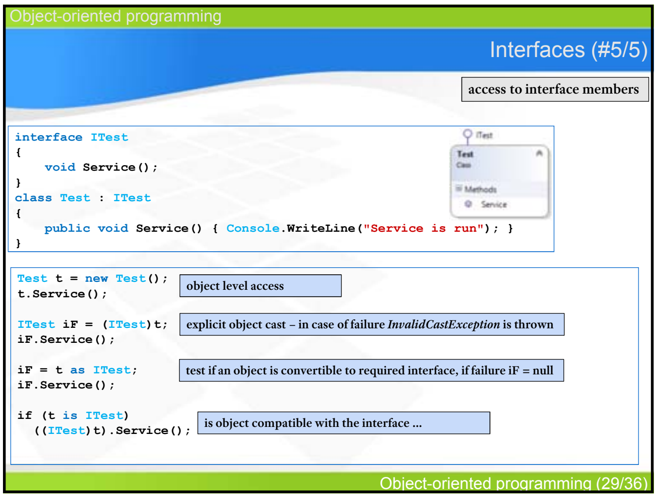
interface members name collisions

```
interface IA
{
    int f(int a);
}
interface IB
{
    int f(float a);
}
class Test : IA, IB
{
    public int f(int a) { return 0; }
    public int f(float a) { return 0; }
}
```

- polymorphism: overloaded method f,
- difference only in the returned type is not enough (it's not Ada)

Object-oriented programming (28/36)

28



Object-oriented programming

Interfaces (#5/5)

access to interface members

```
interface ITest
{
    void Service();
}
class Test : ITest
{
    public void Service() { Console.WriteLine("Service is run"); }
}
```

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

object level access

```
ITest iF = (ITest)t;
iF.Service();
```

explicit object cast – in case of failure *InvalidCastException* is thrown

```
iF = t as ITest;
iF.Service();
```

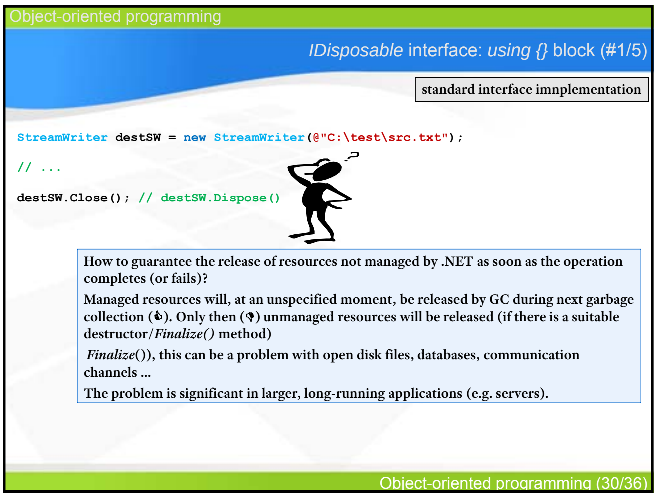
test if an object is convertible to required interface, if failure iF = null

```
if (t is ITest)
  ((ITest)t).Service();
```

is object compatible with the interface ...

Object-oriented programming (29/36)

29



Object-oriented programming

IDisposable interface: *using {}* block (#1/5)

standard interface imnplementation

```
StreamWriter destSW = new StreamWriter(@"C:\test\src.txt");

// ...

destSW.Close(); // destSW.Dispose()
```

How to guarantee the release of resources not managed by .NET as soon as the operation completes (or fails)?

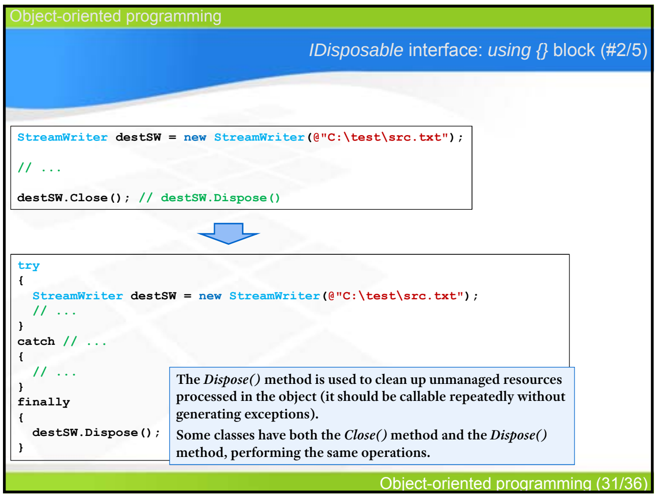
Managed resources will, at an unspecified moment, be released by GC during next garbage collection (👍). Only then (👎) unmanaged resources will be released (if there is a suitable destructor/*Finalize()* method)

Finalize()), this can be a problem with open disk files, databases, communication channels ...

The problem is significant in larger, long-running applications (e.g. servers).

Object-oriented programming (30/36)

30



Object-oriented programming

IDisposable interface: *using {}* block (#2/5)

```
StreamWriter destSW = new StreamWriter(@"C:\test\src.txt");

// ...

destSW.Close(); // destSW.Dispose()
```

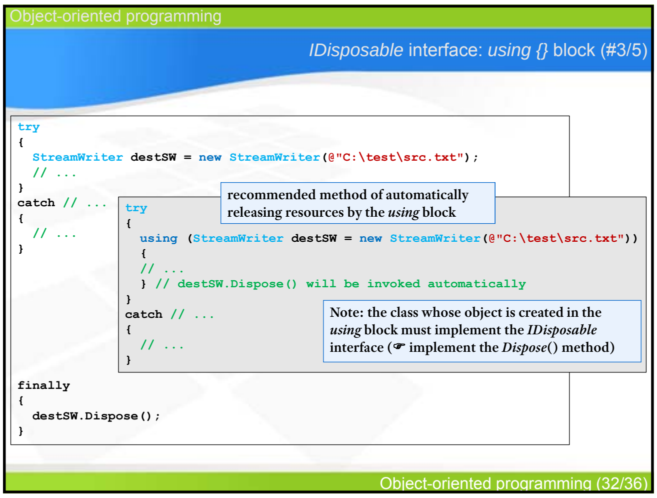
```
try
{
  StreamWriter destSW = new StreamWriter(@"C:\test\src.txt");
  // ...
}
catch // ...
{
  // ...
}
finally
{
  destSW.Dispose();
}
```

The *Dispose()* method is used to clean up unmanaged resources processed in the object (it should be callable repeatedly without generating exceptions).

Some classes have both the *Close()* method and the *Dispose()* method, performing the same operations.

Object-oriented programming (31/36)

31



Object-oriented programming

IDisposable interface: *using {}* block (#3/5)

```
try
{
  StreamWriter destSW = new StreamWriter(@"C:\test\src.txt");
  // ...
}
catch // ...
{
  // ...
}
finally
{
  destSW.Dispose();
}
```

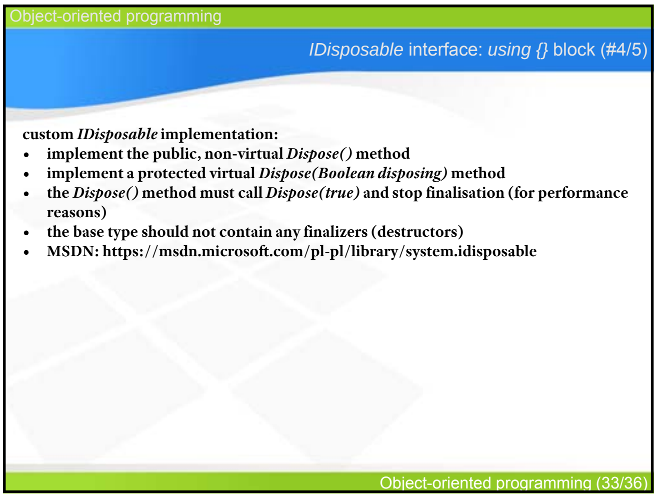
recommended method of automatically releasing resources by the *using* block

```
try
{
  using (StreamWriter destSW = new StreamWriter(@"C:\test\src.txt"))
  {
  // ...
  } // destSW.Dispose() will be invoked automatically
}
catch // ...
{
  // ...
}
```

Note: the class whose object is created in the *using* block must implement the *IDisposable* interface (☞ implement the *Dispose*() method)

Object-oriented programming (32/36)

32



Object-oriented programming

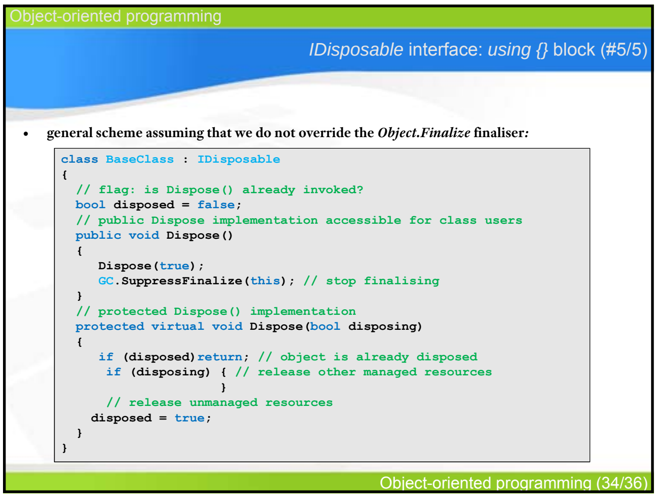
IDisposable interface: *using {}* block (#4/5)

custom *IDisposable* implementation:

- implement the public, non-virtual *Dispose()* method
- implement a protected virtual *Dispose(Boolean disposing)* method
- the *Dispose()* method must call *Dispose(true)* and stop finalisation (for performance reasons)
- the base type should not contain any finalizers (destructors)
- MSDN: https://msdn.microsoft.com/pl-pl/library/system.idisposable

Object-oriented programming (33/36)

33



Object-oriented programming

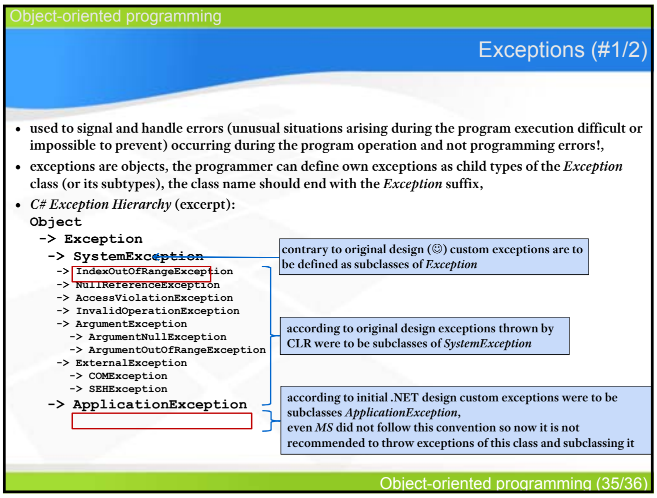
IDisposable interface: *using {}* block (#5/5)

- general scheme assuming that we do not override the *Object.Finalize* finaliser*:*

```
class BaseClass : IDisposable
{
  // flag: is Dispose() already invoked?
  bool disposed = false;
  // public Dispose implementation accessible for class users
  public void Dispose()
  {
     Dispose(true);
     GC.SuppressFinalize(this); // stop finalising
  }
  // protected Dispose() implementation
  protected virtual void Dispose(bool disposing)
  {
     if (disposed)return; // object is already disposed
      if (disposing) { // release other managed resources
                     }
      // release unmanaged resources
    disposed = true;
  }
}
```

Object-oriented programming (34/36)

34



Object-oriented programming

Exceptions (#1/2)

- used to signal and handle errors (unusual situations arising during the program execution difficult or impossible to prevent) occurring during the program operation and not programming errors!,
- exceptions are objects, the programmer can define own exceptions as child types of the *Exception* class (or its subtypes), the class name should end with the *Exception* suffix,
- *C# Exception Hierarchy* (excerpt):

```
Object
 -> Exception
  -> SystemException
   -> IndexOutOfRangeException
   -> NullReferenceException
   -> AccessViolationException
   -> InvalidOperationException
   -> ArgumentException
     -> ArgumentNullException
     -> ArgumentOutOfRangeException
   -> ExternalException
     -> COMException
     -> SEHException
  -> ApplicationException
```

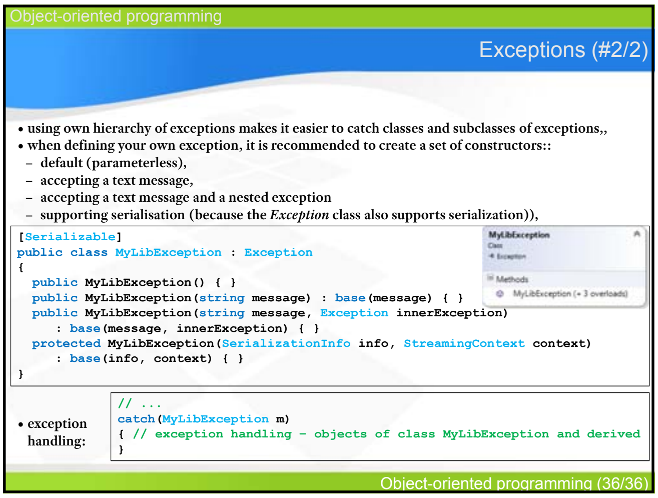
contrary to original design (☺) custom exceptions are to be defined as subclasses of *Exception*

according to original design exceptions thrown by CLR were to be subclasses of *SystemException*

according to initial .NET design custom exceptions were to be subclasses *ApplicationException*, even *MS* did not follow this convention so now it is not recommended to throw exceptions of this class and subclassing it

Object-oriented programming (35/36)

35



Object-oriented programming

Exceptions (#2/2)

- using own hierarchy of exceptions makes it easier to catch classes and subclasses of exceptions,,
- when defining your own exception, it is recommended to create a set of constructors::
  - default (parameterless),
  - accepting a text message,
  - accepting a text message and a nested exception
  - supporting serialisation (because the *Exception* class also supports serialization)),

```
[Serializable]
public class MyLibException : Exception
{
  public MyLibException() { }
  public MyLibException(string message) : base(message) { }
  public MyLibException(string message, Exception innerException)
     : base(message, innerException) { }
  protected MyLibException(SerializationInfo info, StreamingContext context)
     : base(info, context) { }
}
```

- exception handling:

```
// ...
catch(MyLibException m)
{ // exception handling – objects of class MyLibException and derived
}
```

Object-oriented programming (36/36)

36