Object-oriented programming #3
Inheritance

Wojciech Complak

Institute of Computing Science
Faculty of Computing
Poznan University of Technology

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

0.92

1

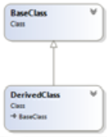
Lecture content

- inheritance "is-a"
- access to members
- inheritance and constructors
- overriding fields and methods
- abstract classes
- *Object* class
- polymorphism
- composition/aggregation "has"
- SOLID principles

2

Inheritance "is-a" (#1/6)

creating class hierarchy

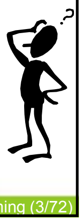


```
class BaseClass
{
}
```
base class

```
class DerivedClass : BaseClass
{
}
```
derived class

based upon the definition of the base class, we create a derived (child) class that:
- has (inherits) all members of the base class (but does not necessarily has access to them)
- adds new members
- redefines (overrides) the inherited members
- the derived class has (usually) more functionality than the base class

this example is correct but ... hardly useful



terminology (depending on the author, language, translation ...):
- *base class = superclass = parent class*
- *derived class = subclass = child class*

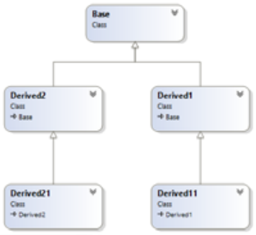
the superclass and subclass are an analogy to the concepts of set theory - the subclass is a subset of the superset

3

Inheritance "is-a" (#2/6)

existing derived class can become the base class for the next one...

```
class Base
{
}
class Derived1 : Base
{
}
class Derived11 : Derived1
{
}
class Derived2 : Base
{
}
class Derived21 : Derived2
{
}
```



- class *Base* is the root of the tree (it's only the base class),
- classes *Derived1* and *Derived2* are intermediate classes because they are both base classes (for *Derived11* i *Derived21*) and derived from class *Base*,
- classes *Derived11* and *Derived21* are leaves in the tree – only derived classes

4

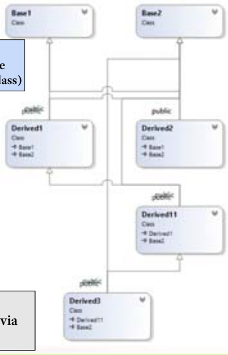
Inheritance "is-a" (#3/6)

other object-oriented languages support more complex structures

C++

```
class Base1
{
};
class Base2
{
};
class Derived1 : public Base1, public Base2
{
};
class Derived2 : public Base1, public Base2
{
};
class Derived11 : public Derived1, public Base2
{
};
class Derived3 : public Derived11, public Base2
{
};
```

important feature of inheritance model of C++ (not supported by C# and Java): the mode of inheriting from the base class (one can voluntarily restrict access to the base class)



- *Derived1* class – multiple inheritance from *Base1 and Base2*
- *Derived3* inherits (among others) twice from *Base2* : indirectly via *Derived11* and directly

5

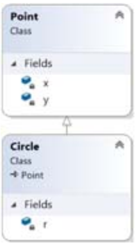
Inheritance "is-a" (#4/6)

inhertinace: adding new fields

```
class Point
{
  private double x, y;
}
class Circle : Point
{
  private double r;
}
```

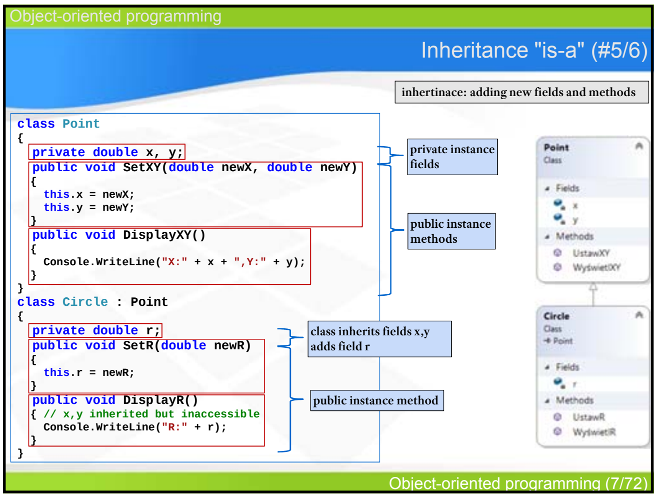
base class contains two fields: *x*, *y*

derived class:
- inherits two fields (*x*, *y*) from its base class
- adds new field (*r*)



„is-a" = Liskov *substitution principle*: objects of a superclass shall be replaceable with objects of its subclasses without breaking the application

6



Object-oriented programming

Inheritance "is-a" (#5/6)

inhertinace: adding new fields and methods

```
class Point
{
  private double x, y;
  public void SetXY(double newX, double newY)
  {
    this.x = newX;
    this.y = newY;
  }
  public void DisplayXY()
  {
    Console.WriteLine("X:" + x + ",Y:" + y);
  }
}
class Circle : Point
{
  private double r;
  public void SetR(double newR)
  {
    this.r = newR;
  }
  public void DisplayR()
  { // x,y inherited but inaccessible
    Console.WriteLine("R:" + r);
  }
}
```

private instance fields

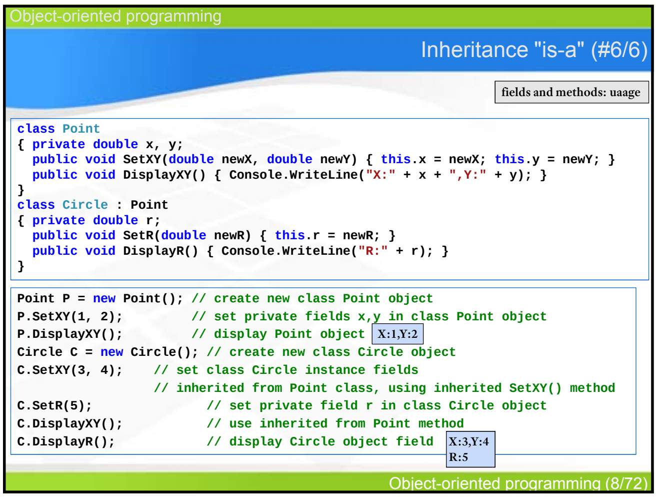
public instance methods

class inherits fields x,y adds field r

public instance method

Object-oriented programming (7/72)

7



Object-oriented programming

Inheritance "is-a" (#6/6)

fields and methods: uaage

```
class Point
{ private double x, y;
  public void SetXY(double newX, double newY) { this.x = newX; this.y = newY; }
  public void DisplayXY() { Console.WriteLine("X:" + x + ",Y:" + y); }
}
class Circle : Point
{ private double r;
  public void SetR(double newR) { this.r = newR; }
  public void DisplayR() { Console.WriteLine("R:" + r); }
}
```

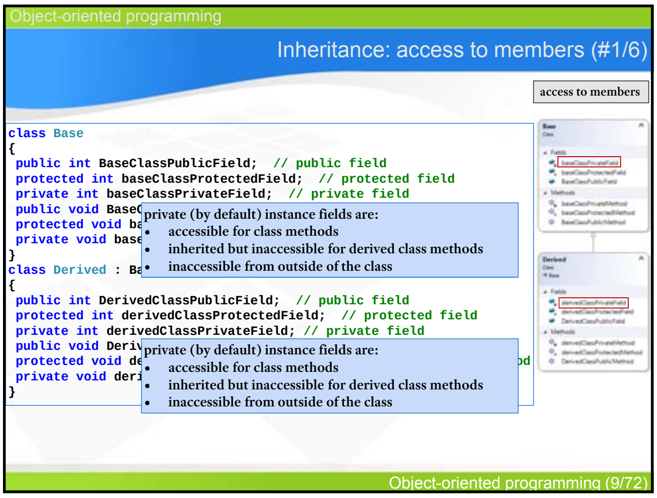
```
Point P = new Point(); // create new class Point object
P.SetXY(1, 2);         // set private fields x,y in class Point object
P.DisplayXY();         // display Point object
Circle C = new Circle(); // create new class Circle object
C.SetXY(3, 4);    // set class Circle instance fields
                  // inherited from Point class, using inherited SetXY() method
C.SetR(5);               // set private field r in class Circle object
C.DisplayXY();           // use inherited from Point method
C.DisplayR();            // display Circle object field
```

X:1,Y:2

X:3,Y:4
R:5

Object-oriented programming (8/72)

8



Object-oriented programming

Inheritance: access to members (#1/6)

access to members

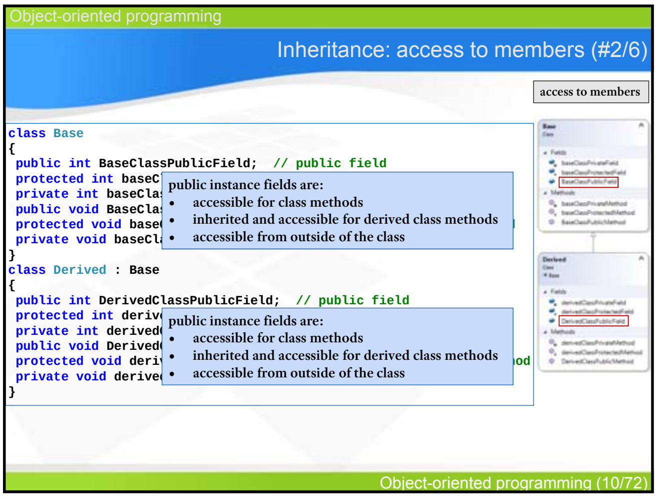
```
class Base
{
 public int BaseClassPublicField;  // public field
 protected int baseClassProtectedField;  // protected field
 private int baseClassPrivateField;  // private field
 public void BaseC
 protected void ba
 private void base
}
class Derived : Ba
{
 public int DerivedClassPublicField;  // public field
 protected int derivedClassProtectedField;  // protected field
 private int derivedClassPrivateField; // private field
 public void Deriv
 protected void de                                            od
 private void deri
}
```

private (by default) instance fields are:
- accessible for class methods
- inherited but inaccessible for derived class methods
- inaccessible from outside of the class

private (by default) instance fields are:
- accessible for class methods
- inherited but inaccessible for derived class methods
- inaccessible from outside of the class

Object-oriented programming (9/72)

9



Object-oriented programming

Inheritance: access to members (#2/6)

access to members

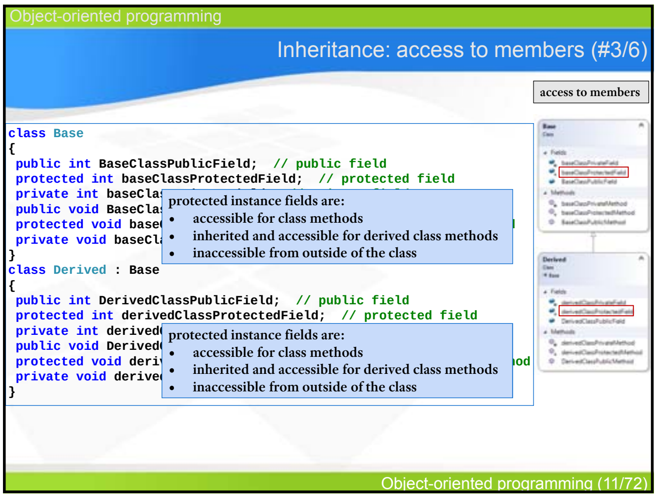
```
class Base
{
 public int BaseClassPublicField;  // public field
 protected int baseC
 private int baseCla
 public void BaseCla
 protected void base
 private void baseCl
}
class Derived : Base
{
 public int DerivedClassPublicField;  // public field
 protected int deriv
 private int derived
 public void Derived
 protected void deri                                         od
 private void derive
}
```

public instance fields are:
- accessible for class methods
- inherited and accessible for derived class methods
- accessible from outside of the class

public instance fields are:
- accessible for class methods
- inherited and accessible for derived class methods
- accessible from outside of the class

Object-oriented programming (10/72)

10



Object-oriented programming

Inheritance: access to members (#3/6)

access to members

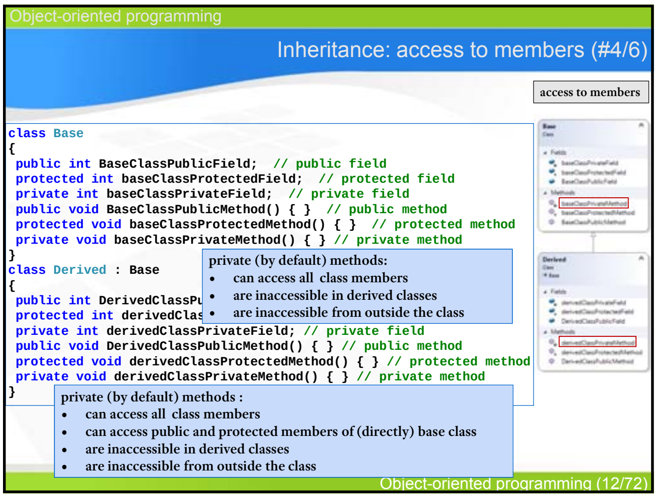
```
class Base
{
 public int BaseClassPublicField;  // public field
 protected int baseClassProtectedField;  // protected field
 private int baseCla
 public void BaseCla
 protected void base
 private void baseCl
}
class Derived : Base
{
 public int DerivedClassPublicField;  // public field
 protected int derivedClassProtectedField;  // protected field
 private int derived
 public void Derived
 protected void deri                                          od
 private void derive
}
```

protected instance fields are:
- accessible for class methods
- inherited and accessible for derived class methods
- inaccessible from outside of the class

protected instance fields are:
- accessible for class methods
- inherited and accessible for derived class methods
- inaccessible from outside of the class

Object-oriented programming (11/72)

11



Object-oriented programming

Inheritance: access to members (#4/6)

access to members

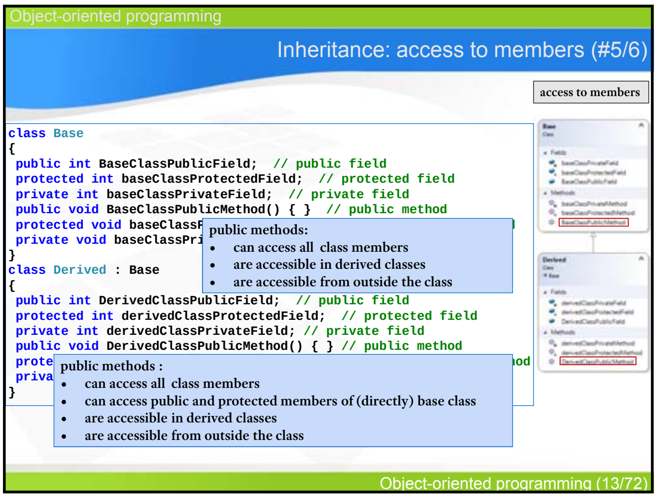
```
class Base
{
 public int BaseClassPublicField;  // public field
 protected int baseClassProtectedField;  // protected field
 private int baseClassPrivateField;  // private field
 public void BaseClassPublicMethod() { }  // public method
 protected void baseClassProtectedMethod() { }  // protected method
 private void baseClassPrivateMethod() { } // private method
}
class Derived : Base
{
 public int DerivedClassPu
 protected int derivedClas
 private int derivedClassPrivateField; // private field
 public void DerivedClassPublicMethod() { } // public method
 protected void derivedClassProtectedMethod() { } // protected method
 private void derivedClassPrivateMethod() { } // private method
}
```

private (by default) methods:
- can access all class members
- are inaccessible in derived classes
- are inaccessible from outside the class

private (by default) methods :
- can access all class members
- can access public and protected members of (directly) base class
- are inaccessible in derived classes
- are inaccessible from outside the class

Object-oriented programming (12/72)

12



Object-oriented programming

Inheritance: access to members (#5/6)

access to members

```
class Base
{
 public int BaseClassPublicField;  // public field
 protected int baseClassProtectedField;  // protected field
 private int baseClassPrivateField;  // private field
 public void BaseClassPublicMethod() { }  // public method
 protected void baseClassP
 private void baseClassPri
}
class Derived : Base
{
 public int DerivedClassPublicField;  // public field
 protected int derivedClassProtectedField;  // protected field
 private int derivedClassPrivateField; // private field
 public void DerivedClassPublicMethod() { } // public method
 prote
 priva
}
```

public methods:

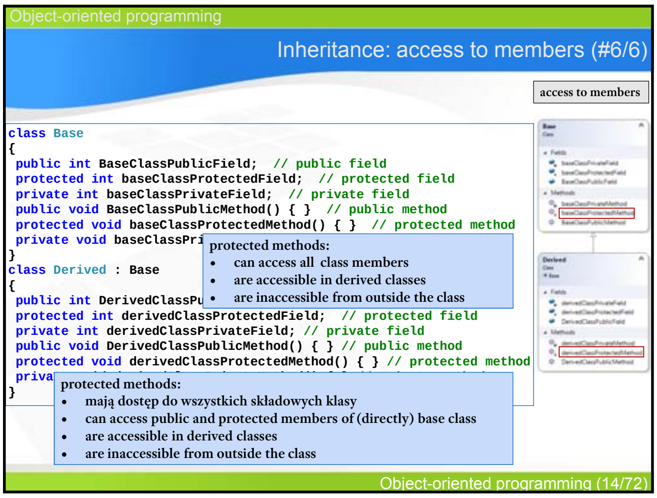
- can access all class members
- are accessible in derived classes
- are accessible from outside the class

public methods :

- can access all class members
- can access public and protected members of (directly) base class
- are accessible in derived classes
- are accessible from outside the class

Object-oriented programming (13/72)

13



Object-oriented programming

Inheritance: access to members (#6/6)

access to members

```
class Base
{
 public int BaseClassPublicField;  // public field
 protected int baseClassProtectedField;  // protected field
 private int baseClassPrivateField;  // private field
 public void BaseClassPublicMethod() { }  // public method
 protected void baseClassProtectedMethod() { }  // protected method
 private void baseClassPri
}
class Derived : Base
{
 public int DerivedClassPu
 protected int derivedClassProtectedField;  // protected field
 private int derivedClassPrivateField; // private field
 public void DerivedClassPublicMethod() { } // public method
 protected void derivedClassProtectedMethod() { } // protected method
 priva
}
```

protected methods:

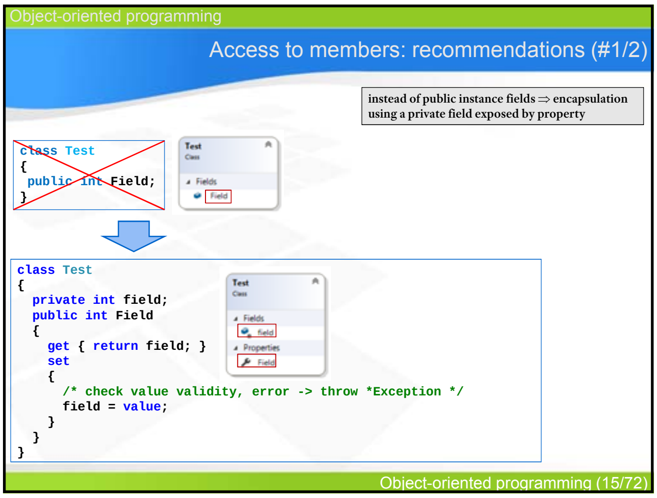
- can access all class members
- are accessible in derived classes
- are inaccessible from outside the class

protected methods:

- mają dostęp do wszystkich składowych klasy
- can access public and protected members of (directly) base class
- are accessible in derived classes
- are inaccessible from outside the class

Object-oriented programming (14/72)

14



Object-oriented programming

Access to members: recommendations (#1/2)

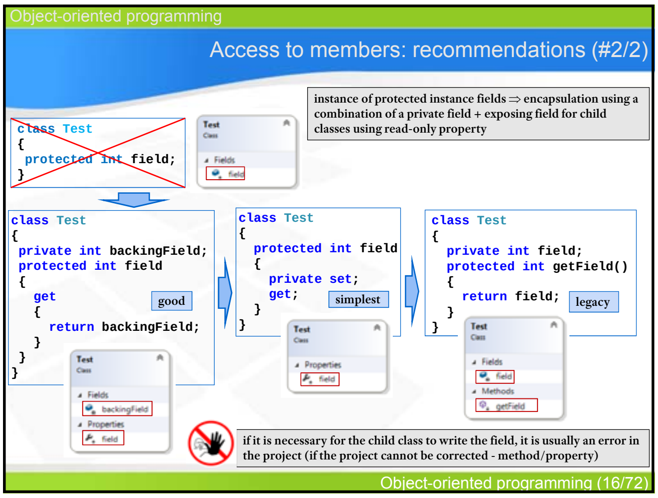
instead of public instance fields ⇒ encapsulation using a private field exposed by property

```
class Test
{
 public int Field;
}
```

```
class Test
{
  private int field;
  public int Field
  {
    get { return field; }
    set
    {
      /* check value validity, error -> throw *Exception */
      field = value;
    }
  }
}
```

Object-oriented programming (15/72)

15



Object-oriented programming

Access to members: recommendations (#2/2)

instance of protected instance fields ⇒ encapsulation using a combination of a private field + exposing field for child classes using read-only property

```
class Test
{
 protected int field;
}
```

```
class Test
{
 private int backingField;
 protected int field
 {
   get
   {
     return backingField;
   }
 }
}
```

good

```
class Test
{
  protected int field
  {
    private set;
    get;
  }
}
```

simplest

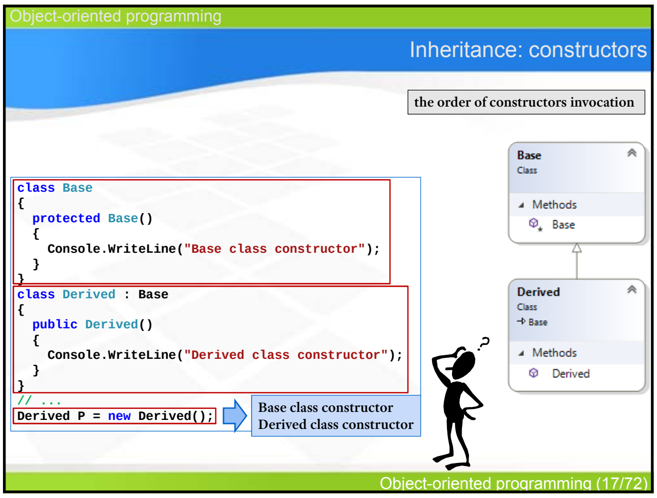
```
class Test
{
  private int field;
  protected int getField()
  {
    return field;
  }
}
```

legacy

if it is necessary for the child class to write the field, it is usually an error in the project (if the project cannot be corrected - method/property)

Object-oriented programming (16/72)

16



Object-oriented programming

Inheritance: constructors

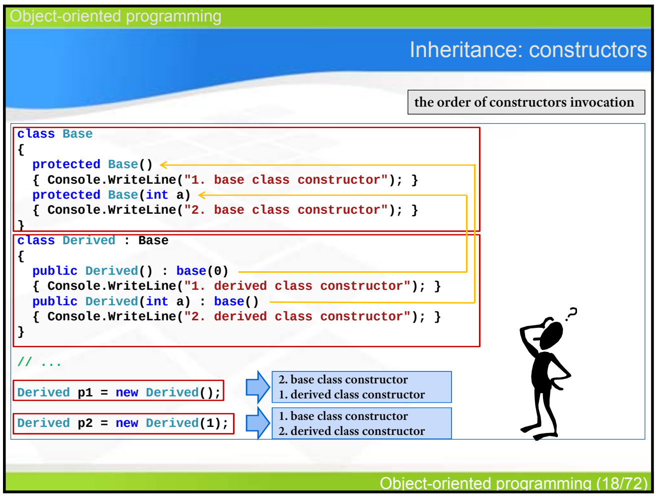
the order of constructors invocation

```
class Base
{
  protected Base()
  {
    Console.WriteLine("Base class constructor");
  }
}
class Derived : Base
{
  public Derived()
  {
    Console.WriteLine("Derived class constructor");
  }
}
// ...
Derived P = new Derived();
```

Base class constructor
Derived class constructor

Object-oriented programming (17/72)

17



Object-oriented programming

Inheritance: constructors

the order of constructors invocation

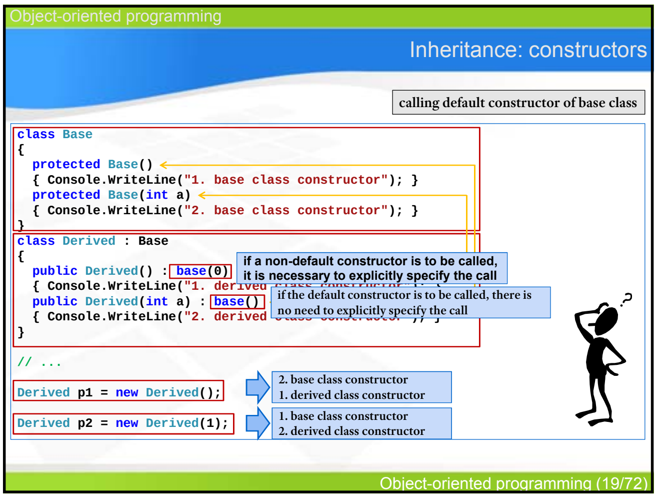
```
class Base
{
  protected Base()
  { Console.WriteLine("1. base class constructor"); }
  protected Base(int a)
  { Console.WriteLine("2. base class constructor"); }
}
class Derived : Base
{
  public Derived() : base(0)
  { Console.WriteLine("1. derived class constructor"); }
  public Derived(int a) : base()
  { Console.WriteLine("2. derived class constructor"); }
}
// ...
Derived p1 = new Derived();
Derived p2 = new Derived(1);
```

2. base class constructor
1. derived class constructor

1. base class constructor
2. derived class constructor

Object-oriented programming (18/72)

18


Object-oriented programming

Inheritance: constructors

calling default constructor of base class

```
class Base
{
  protected Base()
  { Console.WriteLine("1. base class constructor"); }
  protected Base(int a)
  { Console.WriteLine("2. base class constructor"); }
}
class Derived : Base
{
  public Derived() : base(0)
  { Console.WriteLine("1. derived class constructor"); }
  public Derived(int a) : base()
  { Console.WriteLine("2. derived class constructor"); }
}
// ...
Derived p1 = new Derived();
Derived p2 = new Derived(1);
```

if a non-default constructor is to be called, it is necessary to explicitly specify the call

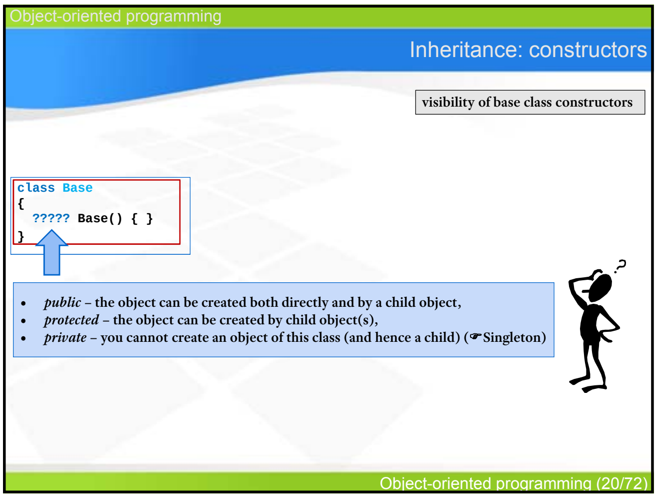
if the default constructor is to be called, there is no need to explicitly specify the call

`Derived p1 = new Derived();` → 2. base class constructor
1. derived class constructor

`Derived p2 = new Derived(1);` → 1. base class constructor
2. derived class constructor

Object-oriented programming (19/72)

19


Object-oriented programming

Inheritance: constructors

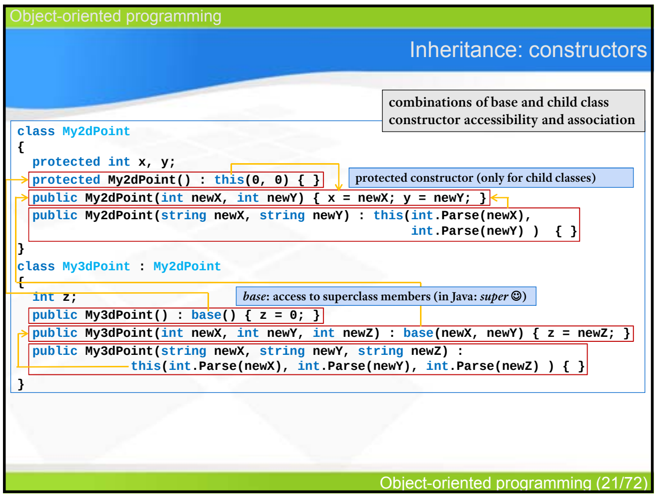
visibility of base class constructors

```
class Base
{
  ????? Base() { }
}
```

- *public* – the object can be created both directly and by a child object,
- *protected* – the object can be created by child object(s),
- *private* – you cannot create an object of this class (and hence a child) (☞Singleton)

Object-oriented programming (20/72)

20


Object-oriented programming

Inheritance: constructors

combinations of base and child class constructor accessibility and association

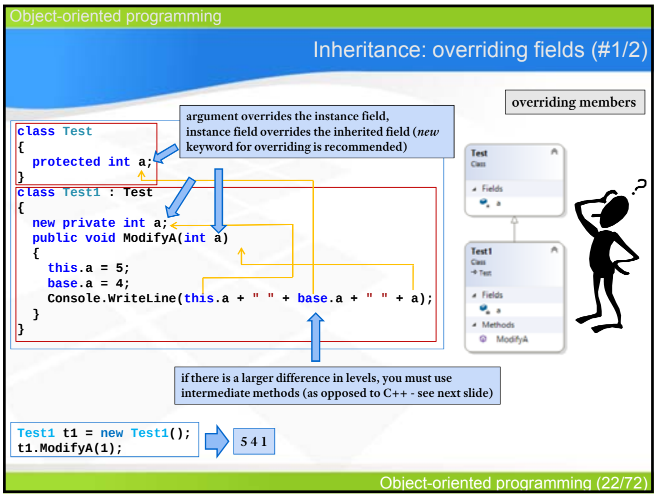
```
class My2dPoint
{
  protected int x, y;
  protected My2dPoint() : this(0, 0) { }
  public My2dPoint(int newX, int newY) { x = newX; y = newY; }
  public My2dPoint(string newX, string newY) : this(int.Parse(newX),
                                                    int.Parse(newY) )  { }
}
class My3dPoint : My2dPoint
{
  int z;
  public My3dPoint() : base() { z = 0; }
  public My3dPoint(int newX, int newY, int newZ) : base(newX, newY) { z = newZ; }
  public My3dPoint(string newX, string newY, string newZ) :
         this(int.Parse(newX), int.Parse(newY), int.Parse(newZ) ) { }
}
```

protected constructor (only for child classes)

base: access to superclass members (in Java: *super* ☺)

Object-oriented programming (21/72)

21


Object-oriented programming

Inheritance: overriding fields (#1/2)

overriding members

```
class Test
{
  protected int a;
}
class Test1 : Test
{
  new private int a;
  public void ModifyA(int a)
  {
    this.a = 5;
    base.a = 4;
    Console.WriteLine(this.a + " " + base.a + " " + a);
  }
}
```

argument overrides the instance field, instance field overrides the inherited field (*new* keyword for overriding is recommended)

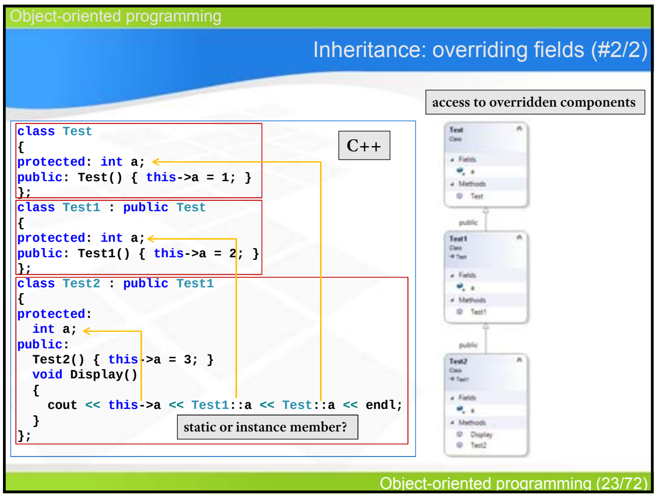
if there is a larger difference in levels, you must use intermediate methods (as opposed to C++ - see next slide)

```
Test1 t1 = new Test1();
t1.ModifyA(1);
```

→ 5 4 1

Object-oriented programming (22/72)

22


Object-oriented programming

Inheritance: overriding fields (#2/2)

access to overridden components

C++

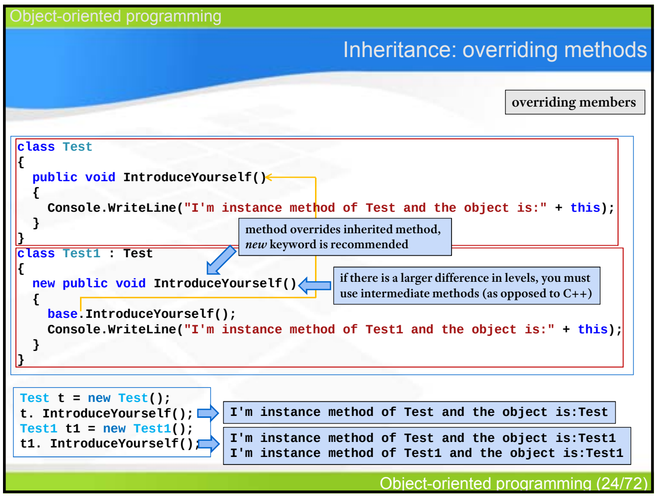
```
class Test
{
protected: int a;
public: Test() { this->a = 1; }
};
class Test1 : public Test
{
protected: int a;
public: Test1() { this->a = 2; }
};
class Test2 : public Test1
{
protected:
  int a;
public:
  Test2() { this->a = 3; }
  void Display()
  {
    cout << this->a << Test1::a << Test::a << endl;
  }
};
```

static or instance member?

Test
Class
Fields
a
Methods
Test
public
Test1
Class
→ Test
Fields
a
Methods
Test1
public
Test2
Class
→ Test1
Fields
a
Methods
Display
Test2

Object-oriented programming (23/72)

23


Object-oriented programming

Inheritance: overriding methods

overriding members

```
class Test
{
  public void IntroduceYourself()
  {
    Console.WriteLine("I'm instance method of Test and the object is:" + this);
  }
}
class Test1 : Test
{
  new public void IntroduceYourself()
  {
    base.IntroduceYourself();
    Console.WriteLine("I'm instance method of Test1 and the object is:" + this);
  }
}
```

method overrides inherited method, *new* keyword is recommended

if there is a larger difference in levels, you must use intermediate methods (as opposed to C++)

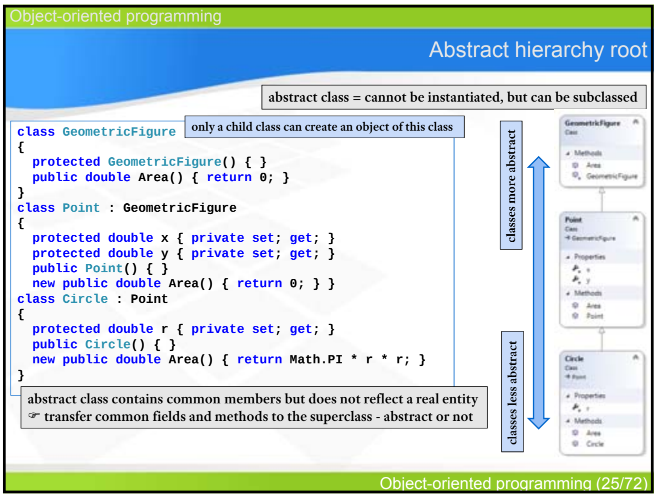
```
Test t = new Test();
t. IntroduceYourself();
Test1 t1 = new Test1();
t1. IntroduceYourself();
```

```
I'm instance method of Test and the object is:Test
```

```
I'm instance method of Test and the object is:Test1
I'm instance method of Test1 and the object is:Test1
```

Object-oriented programming (24/72)

24



Object-oriented programming

Abstract hierarchy root

abstract class = cannot be instantiated, but can be subclassed

only a child class can create an object of this class

```
class GeometricFigure
{
  protected GeometricFigure() { }
  public double Area() { return 0; }
}
class Point : GeometricFigure
{
  protected double x { private set; get; }
  protected double y { private set; get; }
  public Point() { }
  new public double Area() { return 0; } }
class Circle : Point
{
  protected double r { private set; get; }
  public Circle() { }
  new public double Area() { return Math.PI * r * r; }
}
```

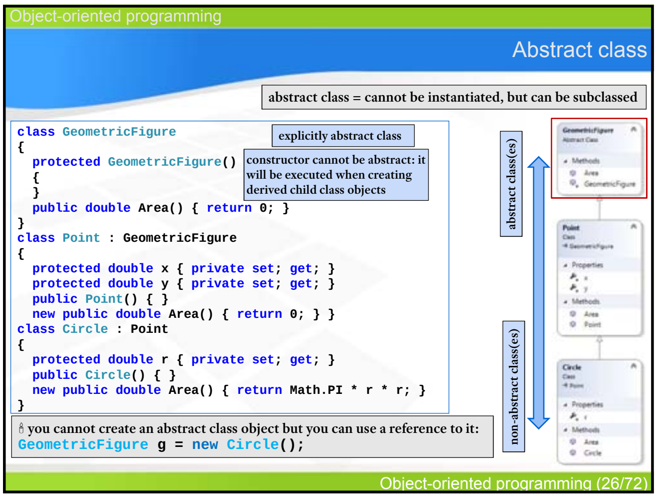
abstract class contains common members but does not reflect a real entity
☞ transfer common fields and methods to the superclass - abstract or not

classes more abstract

classes less abstract

Object-oriented programming (25/72)

25



Object-oriented programming

Abstract class

abstract class = cannot be instantiated, but can be subclassed

explicitly abstract class

constructor cannot be abstract: it will be executed when creating derived child class objects

```
class GeometricFigure
{
  protected GeometricFigure()
  {
  }
  public double Area() { return 0; }
}
class Point : GeometricFigure
{
  protected double x { private set; get; }
  protected double y { private set; get; }
  public Point() { }
  new public double Area() { return 0; } }
class Circle : Point
{
  protected double r { private set; get; }
  public Circle() { }
  new public double Area() { return Math.PI * r * r; }
}
```

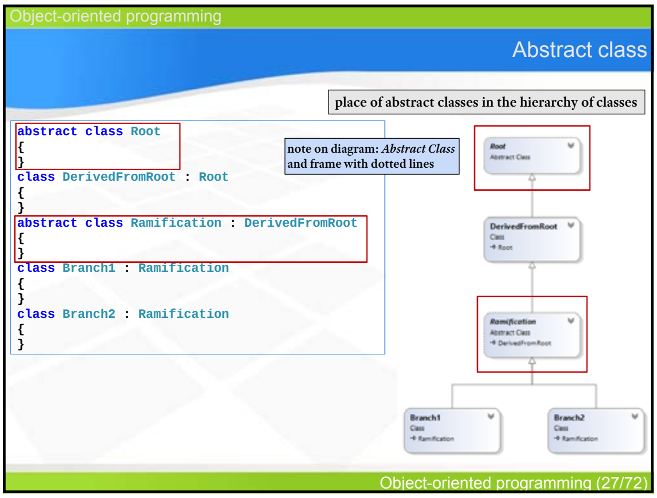
🕯 you cannot create an abstract class object but you can use a reference to it:
`GeometricFigure g = new Circle();`

abstract class(es)

non-abstract class(es)

Object-oriented programming (26/72)

26



Object-oriented programming

Abstract class

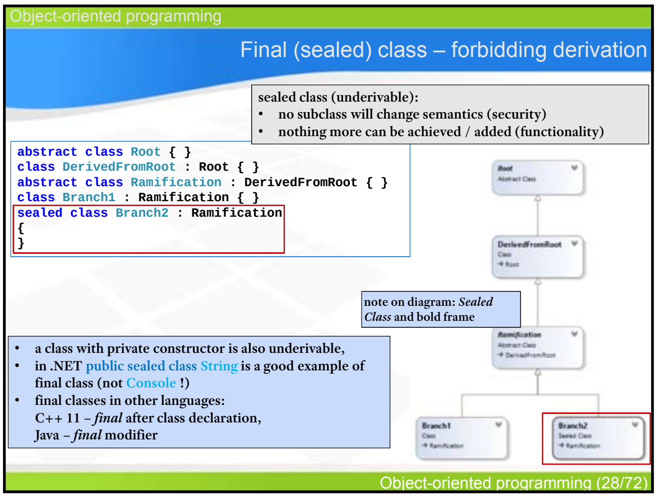
place of abstract classes in the hierarchy of classes

```
abstract class Root
{
}
class DerivedFromRoot : Root
{
}
abstract class Ramification : DerivedFromRoot
{
}
class Branch1 : Ramification
{
}
class Branch2 : Ramification
{
}
```

note on diagram: *Abstract Class* and frame with dotted lines

Object-oriented programming (27/72)

27



Object-oriented programming

Final (sealed) class – forbidding derivation

sealed class (underivable):
- no subclass will change semantics (security)
- nothing more can be achieved / added (functionality)

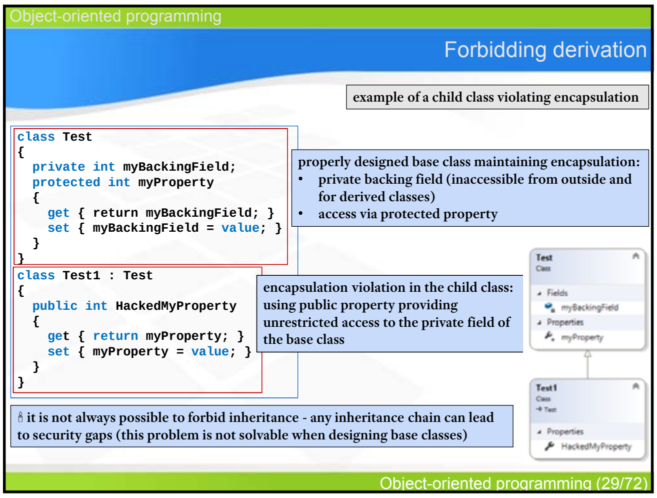
```
abstract class Root { }
class DerivedFromRoot : Root { }
abstract class Ramification : DerivedFromRoot { }
class Branch1 : Ramification { }
sealed class Branch2 : Ramification
{
}
```

note on diagram: *Sealed Class* and bold frame

- a class with private constructor is also underivable,
- in .NET public sealed class String is a good example of final class (not Console !)
- final classes in other languages:
  C++ 11 – *final* after class declaration,
  Java – *final* modifier

Object-oriented programming (28/72)

28



Object-oriented programming

Forbidding derivation

example of a child class violating encapsulation

```
class Test
{
  private int myBackingField;
  protected int myProperty
  {
    get { return myBackingField; }
    set { myBackingField = value; }
  }
}
class Test1 : Test
{
  public int HackedMyProperty
  {
    get { return myProperty; }
    set { myProperty = value; }
  }
}
```

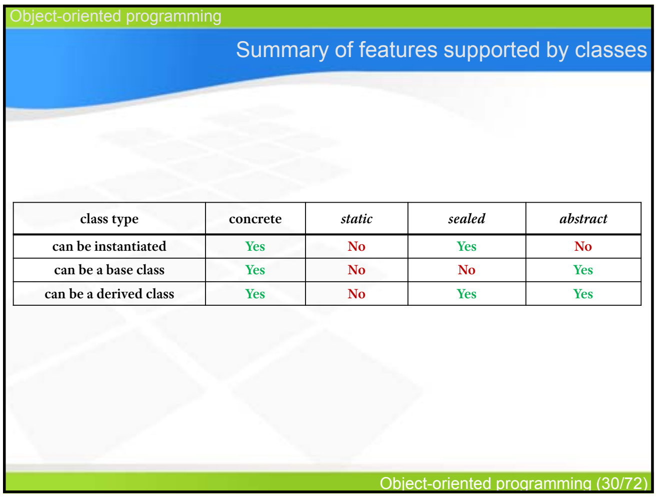
properly designed base class maintaining encapsulation:
- private backing field (inaccessible from outside and for derived classes)
- access via protected property

encapsulation violation in the child class: using public property providing unrestricted access to the private field of the base class

🕯 it is not always possible to forbid inheritance - any inheritance chain can lead to security gaps (this problem is not solvable when designing base classes)

Object-oriented programming (29/72)

29



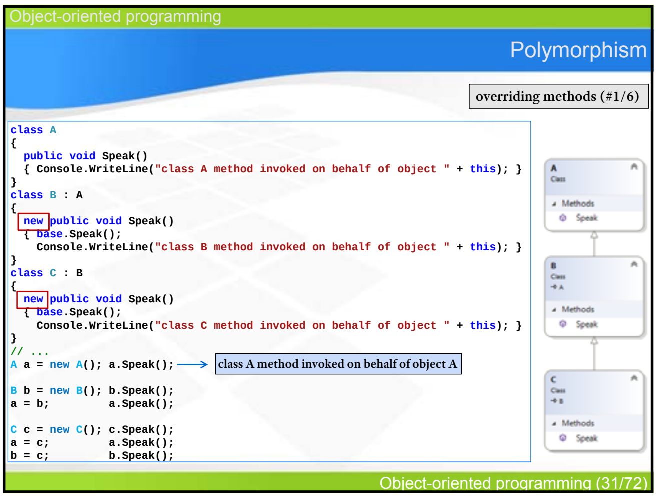
Object-oriented programming

Summary of features supported by classes

class type	concrete	*static*	*sealed*	*abstract*
can be instantiated	Yes	No	Yes	No
can be a base class	Yes	No	No	Yes
can be a derived class	Yes	No	Yes	Yes

Object-oriented programming (30/72)

30



Object-oriented programming

Polymorphism

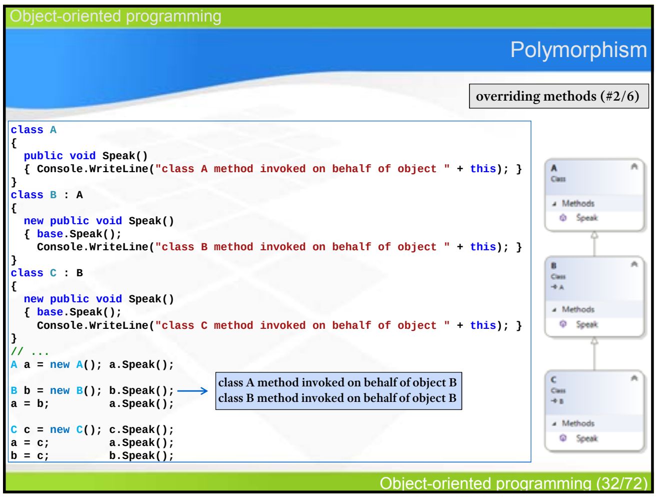
overriding methods (#1/6)

```
class A
{
  public void Speak()
  { Console.WriteLine("class A method invoked on behalf of object " + this); }
}
class B : A
{
  new public void Speak()
  { base.Speak();
    Console.WriteLine("class B method invoked on behalf of object " + this); }
}
class C : B
{
  new public void Speak()
  { base.Speak();
    Console.WriteLine("class C method invoked on behalf of object " + this); }
}
// ...
A a = new A(); a.Speak();

B b = new B(); b.Speak();
a = b;         a.Speak();

C c = new C(); c.Speak();
a = c;         a.Speak();
b = c;         b.Speak();
```

class A method invoked on behalf of object A

Object-oriented programming (31/72)

31



Object-oriented programming

Polymorphism

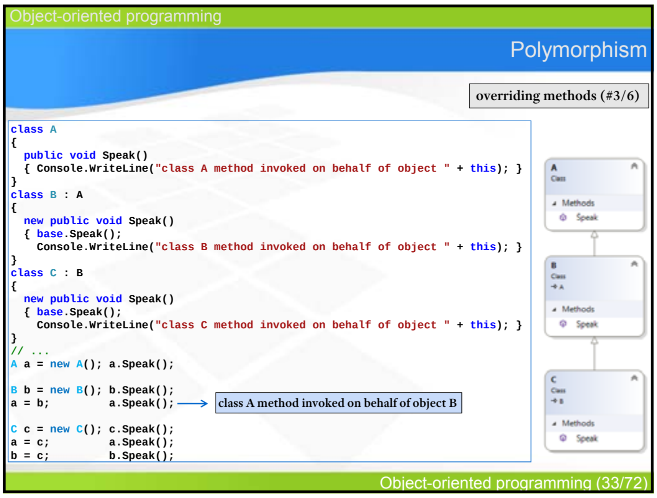
overriding methods (#2/6)

```
class A
{
  public void Speak()
  { Console.WriteLine("class A method invoked on behalf of object " + this); }
}
class B : A
{
  new public void Speak()
  { base.Speak();
    Console.WriteLine("class B method invoked on behalf of object " + this); }
}
class C : B
{
  new public void Speak()
  { base.Speak();
    Console.WriteLine("class C method invoked on behalf of object " + this); }
}
// ...
A a = new A(); a.Speak();

B b = new B(); b.Speak();
a = b;         a.Speak();

C c = new C(); c.Speak();
a = c;         a.Speak();
b = c;         b.Speak();
```

class A method invoked on behalf of object B
class B method invoked on behalf of object B

Object-oriented programming (32/72)

32



Object-oriented programming

Polymorphism

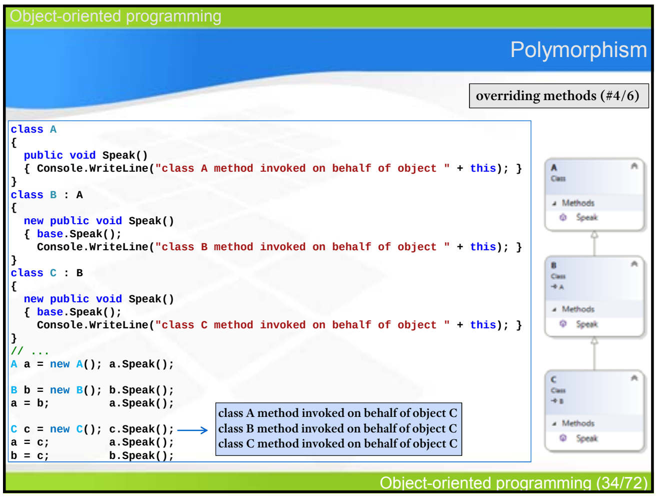
overriding methods (#3/6)

```
class A
{
  public void Speak()
  { Console.WriteLine("class A method invoked on behalf of object " + this); }
}
class B : A
{
  new public void Speak()
  { base.Speak();
    Console.WriteLine("class B method invoked on behalf of object " + this); }
}
class C : B
{
  new public void Speak()
  { base.Speak();
    Console.WriteLine("class C method invoked on behalf of object " + this); }
}
// ...
A a = new A(); a.Speak();

B b = new B(); b.Speak();
a = b;         a.Speak();

C c = new C(); c.Speak();
a = c;         a.Speak();
b = c;         b.Speak();
```

class A method invoked on behalf of object B

Object-oriented programming (33/72)

33



Object-oriented programming

Polymorphism

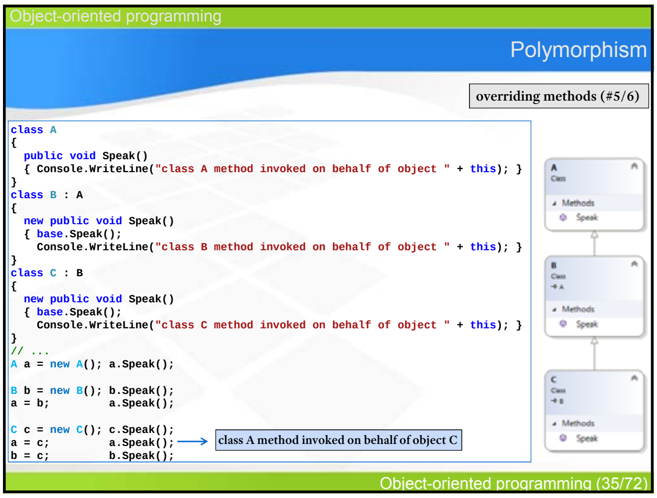
overriding methods (#4/6)

```
class A
{
  public void Speak()
  { Console.WriteLine("class A method invoked on behalf of object " + this); }
}
class B : A
{
  new public void Speak()
  { base.Speak();
    Console.WriteLine("class B method invoked on behalf of object " + this); }
}
class C : B
{
  new public void Speak()
  { base.Speak();
    Console.WriteLine("class C method invoked on behalf of object " + this); }
}
// ...
A a = new A(); a.Speak();

B b = new B(); b.Speak();
a = b;         a.Speak();

C c = new C(); c.Speak();
a = c;         a.Speak();
b = c;         b.Speak();
```

class A method invoked on behalf of object C
class B method invoked on behalf of object C
class C method invoked on behalf of object C

Object-oriented programming (34/72)

34



Object-oriented programming

Polymorphism

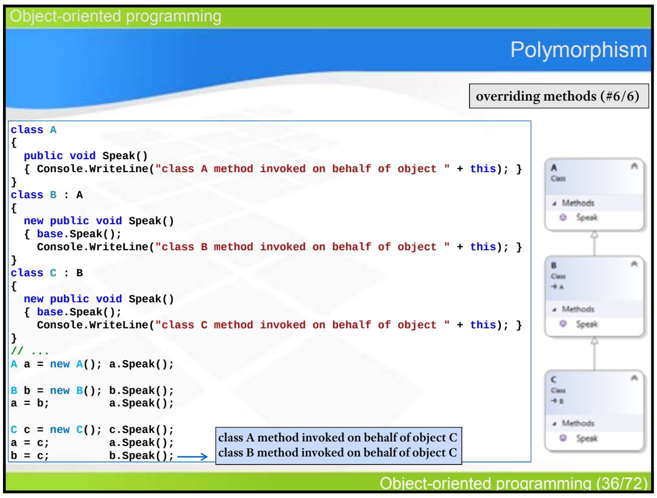
overriding methods (#5/6)

```
class A
{
  public void Speak()
  { Console.WriteLine("class A method invoked on behalf of object " + this); }
}
class B : A
{
  new public void Speak()
  { base.Speak();
    Console.WriteLine("class B method invoked on behalf of object " + this); }
}
class C : B
{
  new public void Speak()
  { base.Speak();
    Console.WriteLine("class C method invoked on behalf of object " + this); }
}
// ...
A a = new A(); a.Speak();

B b = new B(); b.Speak();
a = b;         a.Speak();

C c = new C(); c.Speak();
a = c;         a.Speak();
b = c;         b.Speak();
```

class A method invoked on behalf of object C

Object-oriented programming (35/72)

35



Object-oriented programming

Polymorphism

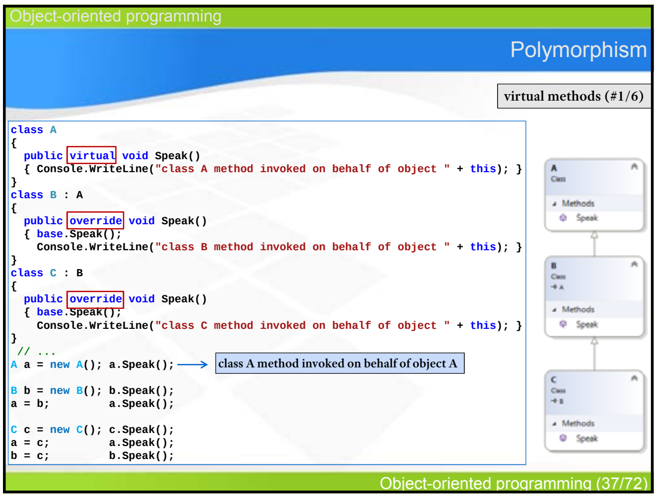
overriding methods (#6/6)

```
class A
{
  public void Speak()
  { Console.WriteLine("class A method invoked on behalf of object " + this); }
}
class B : A
{
  new public void Speak()
  { base.Speak();
    Console.WriteLine("class B method invoked on behalf of object " + this); }
}
class C : B
{
  new public void Speak()
  { base.Speak();
    Console.WriteLine("class C method invoked on behalf of object " + this); }
}
// ...
A a = new A(); a.Speak();

B b = new B(); b.Speak();
a = b;         a.Speak();

C c = new C(); c.Speak();
a = c;         a.Speak();
b = c;         b.Speak();
```

class A method invoked on behalf of object C
class B method invoked on behalf of object C

Object-oriented programming (36/72)

36



Object-oriented programming

Polymorphism

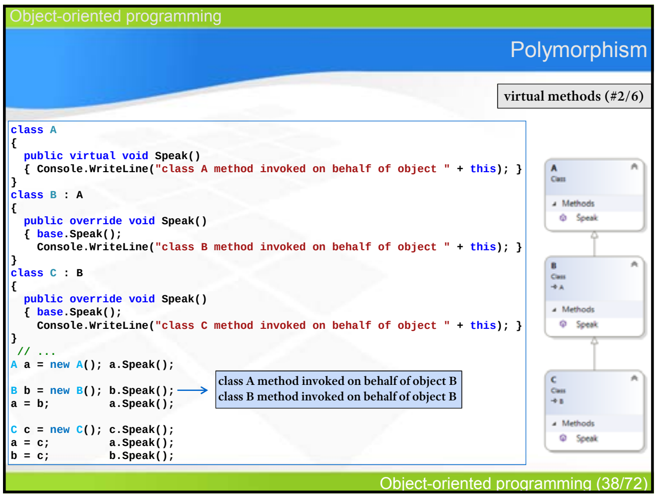
virtual methods (#1/6)

```
class A
{
  public virtual void Speak()
  { Console.WriteLine("class A method invoked on behalf of object " + this); }
}
class B : A
{
  public override void Speak()
  { base.Speak();
    Console.WriteLine("class B method invoked on behalf of object " + this); }
}
class C : B
{
  public override void Speak()
  { base.Speak();
    Console.WriteLine("class C method invoked on behalf of object " + this); }
}
 // ...
A a = new A(); a.Speak();

B b = new B(); b.Speak();
a = b;         a.Speak();

C c = new C(); c.Speak();
a = c;         a.Speak();
b = c;         b.Speak();
```

```
class A method invoked on behalf of object A
```

Object-oriented programming (37/72)

37



Object-oriented programming

Polymorphism

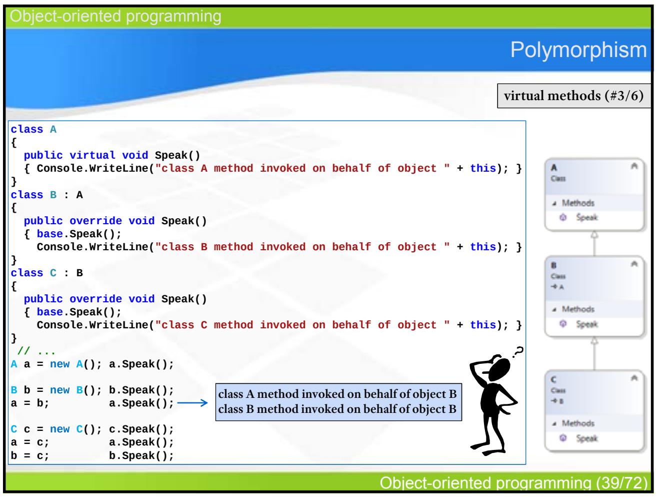
virtual methods (#2/6)

```
class A
{
  public virtual void Speak()
  { Console.WriteLine("class A method invoked on behalf of object " + this); }
}
class B : A
{
  public override void Speak()
  { base.Speak();
    Console.WriteLine("class B method invoked on behalf of object " + this); }
}
class C : B
{
  public override void Speak()
  { base.Speak();
    Console.WriteLine("class C method invoked on behalf of object " + this); }
}
 // ...
A a = new A(); a.Speak();

B b = new B(); b.Speak();
a = b;         a.Speak();

C c = new C(); c.Speak();
a = c;         a.Speak();
b = c;         b.Speak();
```

```
class A method invoked on behalf of object B
class B method invoked on behalf of object B
```

Object-oriented programming (38/72)

38



Object-oriented programming

Polymorphism

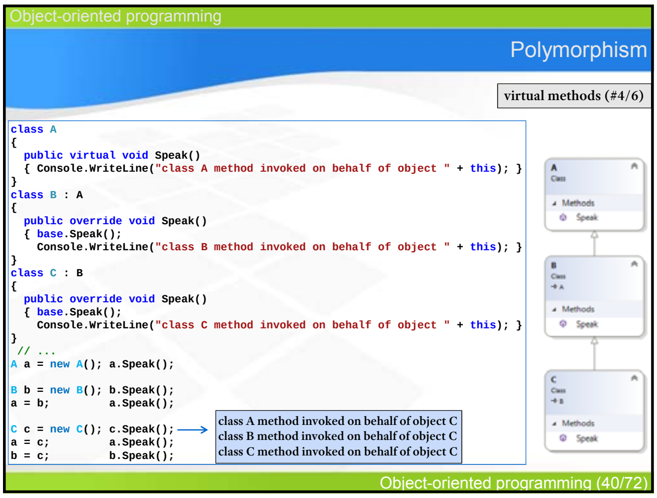
virtual methods (#3/6)

```
class A
{
  public virtual void Speak()
  { Console.WriteLine("class A method invoked on behalf of object " + this); }
}
class B : A
{
  public override void Speak()
  { base.Speak();
    Console.WriteLine("class B method invoked on behalf of object " + this); }
}
class C : B
{
  public override void Speak()
  { base.Speak();
    Console.WriteLine("class C method invoked on behalf of object " + this); }
}
 // ...
A a = new A(); a.Speak();

B b = new B(); b.Speak();
a = b;         a.Speak();

C c = new C(); c.Speak();
a = c;         a.Speak();
b = c;         b.Speak();
```

```
class A method invoked on behalf of object B
class B method invoked on behalf of object B
```

Object-oriented programming (39/72)

39



Object-oriented programming

Polymorphism

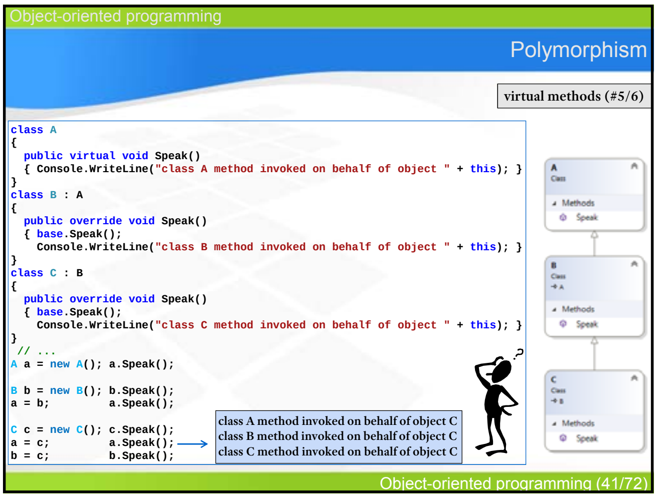
virtual methods (#4/6)

```
class A
{
  public virtual void Speak()
  { Console.WriteLine("class A method invoked on behalf of object " + this); }
}
class B : A
{
  public override void Speak()
  { base.Speak();
    Console.WriteLine("class B method invoked on behalf of object " + this); }
}
class C : B
{
  public override void Speak()
  { base.Speak();
    Console.WriteLine("class C method invoked on behalf of object " + this); }
}
 // ...
A a = new A(); a.Speak();

B b = new B(); b.Speak();
a = b;         a.Speak();

C c = new C(); c.Speak();
a = c;         a.Speak();
b = c;         b.Speak();
```

```
class A method invoked on behalf of object C
class B method invoked on behalf of object C
class C method invoked on behalf of object C
```

Object-oriented programming (40/72)

40



Object-oriented programming

Polymorphism

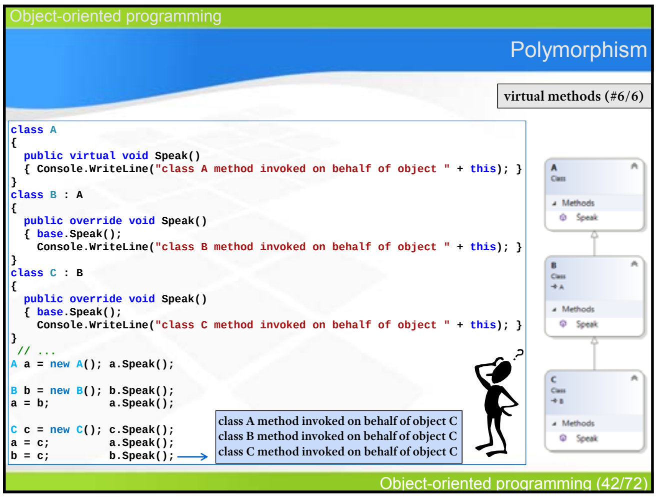
virtual methods (#5/6)

```
class A
{
  public virtual void Speak()
  { Console.WriteLine("class A method invoked on behalf of object " + this); }
}
class B : A
{
  public override void Speak()
  { base.Speak();
    Console.WriteLine("class B method invoked on behalf of object " + this); }
}
class C : B
{
  public override void Speak()
  { base.Speak();
    Console.WriteLine("class C method invoked on behalf of object " + this); }
}
 // ...
A a = new A(); a.Speak();

B b = new B(); b.Speak();
a = b;         a.Speak();

C c = new C(); c.Speak();
a = c;         a.Speak();
b = c;         b.Speak();
```

```
class A method invoked on behalf of object C
class B method invoked on behalf of object C
class C method invoked on behalf of object C
```

Object-oriented programming (41/72)

41



Object-oriented programming

Polymorphism

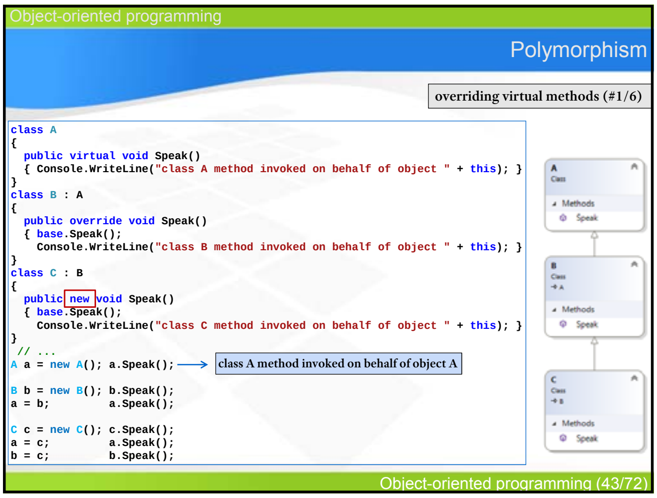
virtual methods (#6/6)

```
class A
{
  public virtual void Speak()
  { Console.WriteLine("class A method invoked on behalf of object " + this); }
}
class B : A
{
  public override void Speak()
  { base.Speak();
    Console.WriteLine("class B method invoked on behalf of object " + this); }
}
class C : B
{
  public override void Speak()
  { base.Speak();
    Console.WriteLine("class C method invoked on behalf of object " + this); }
}
 // ...
A a = new A(); a.Speak();

B b = new B(); b.Speak();
a = b;         a.Speak();

C c = new C(); c.Speak();
a = c;         a.Speak();
b = c;         b.Speak();
```

```
class A method invoked on behalf of object C
class B method invoked on behalf of object C
class C method invoked on behalf of object C
```

Object-oriented programming (42/72)

42



Object-oriented programming

Polymorphism

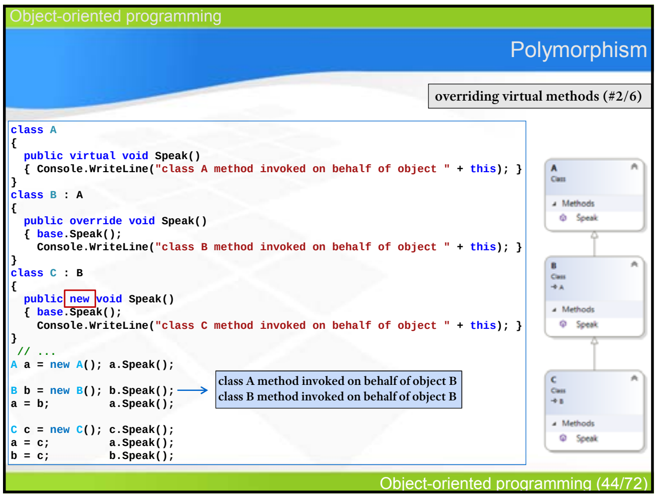
overriding virtual methods (#1/6)

```
class A
{
  public virtual void Speak()
  { Console.WriteLine("class A method invoked on behalf of object " + this); }
}
class B : A
{
  public override void Speak()
  { base.Speak();
    Console.WriteLine("class B method invoked on behalf of object " + this); }
}
class C : B
{
  public new void Speak()
  { base.Speak();
    Console.WriteLine("class C method invoked on behalf of object " + this); }
}
 // ...
A a = new A(); a.Speak();

B b = new B(); b.Speak();
a = b;         a.Speak();

C c = new C(); c.Speak();
a = c;         a.Speak();
b = c;         b.Speak();
```

class A method invoked on behalf of object A

Object-oriented programming (43/72)

43



Object-oriented programming

Polymorphism

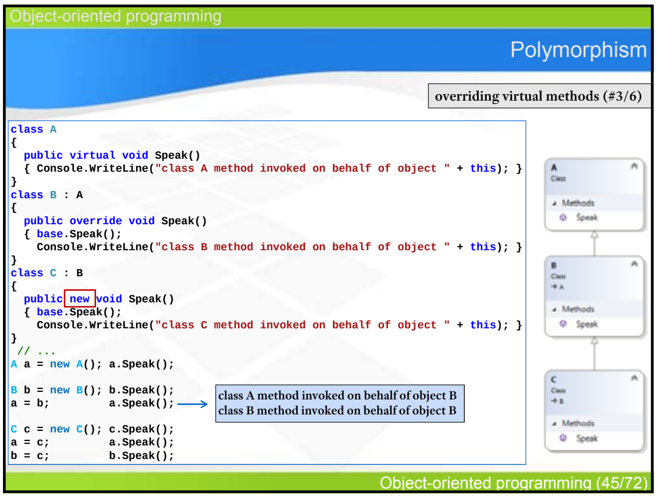
overriding virtual methods (#2/6)

```
class A
{
  public virtual void Speak()
  { Console.WriteLine("class A method invoked on behalf of object " + this); }
}
class B : A
{
  public override void Speak()
  { base.Speak();
    Console.WriteLine("class B method invoked on behalf of object " + this); }
}
class C : B
{
  public new void Speak()
  { base.Speak();
    Console.WriteLine("class C method invoked on behalf of object " + this); }
}
 // ...
A a = new A(); a.Speak();

B b = new B(); b.Speak();
a = b;         a.Speak();

C c = new C(); c.Speak();
a = c;         a.Speak();
b = c;         b.Speak();
```

```
class A method invoked on behalf of object B
class B method invoked on behalf of object B
```

Object-oriented programming (44/72)

44



Object-oriented programming

Polymorphism

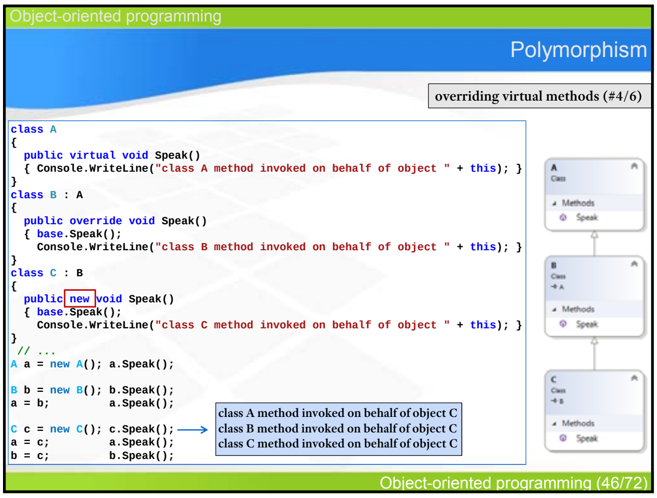
overriding virtual methods (#3/6)

```
class A
{
  public virtual void Speak()
  { Console.WriteLine("class A method invoked on behalf of object " + this); }
}
class B : A
{
  public override void Speak()
  { base.Speak();
    Console.WriteLine("class B method invoked on behalf of object " + this); }
}
class C : B
{
  public new void Speak()
  { base.Speak();
    Console.WriteLine("class C method invoked on behalf of object " + this); }
}
 // ...
A a = new A(); a.Speak();

B b = new B(); b.Speak();
a = b;         a.Speak();

C c = new C(); c.Speak();
a = c;         a.Speak();
b = c;         b.Speak();
```

```
class A method invoked on behalf of object B
class B method invoked on behalf of object B
```

Object-oriented programming (45/72)

45



Object-oriented programming

Polymorphism

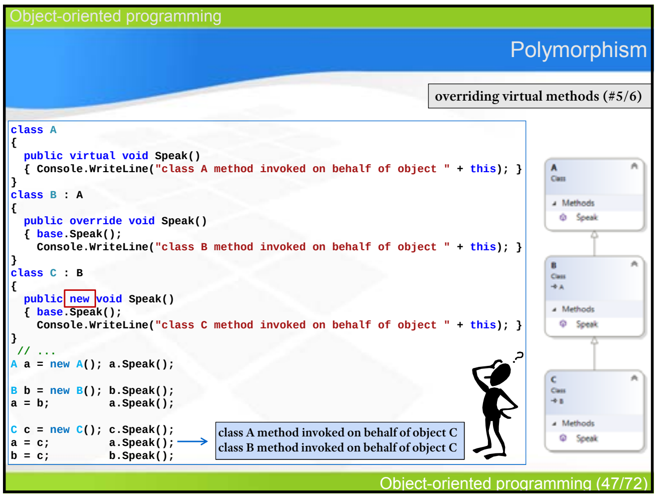
overriding virtual methods (#4/6)

```
class A
{
  public virtual void Speak()
  { Console.WriteLine("class A method invoked on behalf of object " + this); }
}
class B : A
{
  public override void Speak()
  { base.Speak();
    Console.WriteLine("class B method invoked on behalf of object " + this); }
}
class C : B
{
  public new void Speak()
  { base.Speak();
    Console.WriteLine("class C method invoked on behalf of object " + this); }
}
 // ...
A a = new A(); a.Speak();

B b = new B(); b.Speak();
a = b;         a.Speak();

C c = new C(); c.Speak();
a = c;         a.Speak();
b = c;         b.Speak();
```

```
class A method invoked on behalf of object C
class B method invoked on behalf of object C
class C method invoked on behalf of object C
```

Object-oriented programming (46/72)

46



Object-oriented programming

Polymorphism

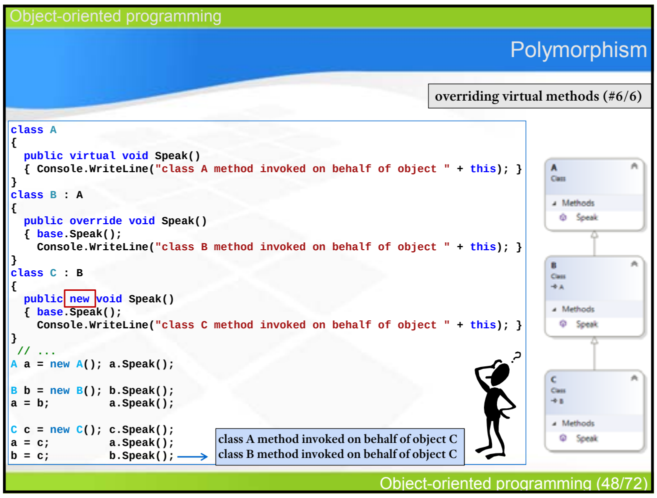
overriding virtual methods (#5/6)

```
class A
{
  public virtual void Speak()
  { Console.WriteLine("class A method invoked on behalf of object " + this); }
}
class B : A
{
  public override void Speak()
  { base.Speak();
    Console.WriteLine("class B method invoked on behalf of object " + this); }
}
class C : B
{
  public new void Speak()
  { base.Speak();
    Console.WriteLine("class C method invoked on behalf of object " + this); }
}
 // ...
A a = new A(); a.Speak();

B b = new B(); b.Speak();
a = b;         a.Speak();

C c = new C(); c.Speak();
a = c;         a.Speak();
b = c;         b.Speak();
```

```
class A method invoked on behalf of object C
class B method invoked on behalf of object C
```

Object-oriented programming (47/72)

47



Object-oriented programming

Polymorphism

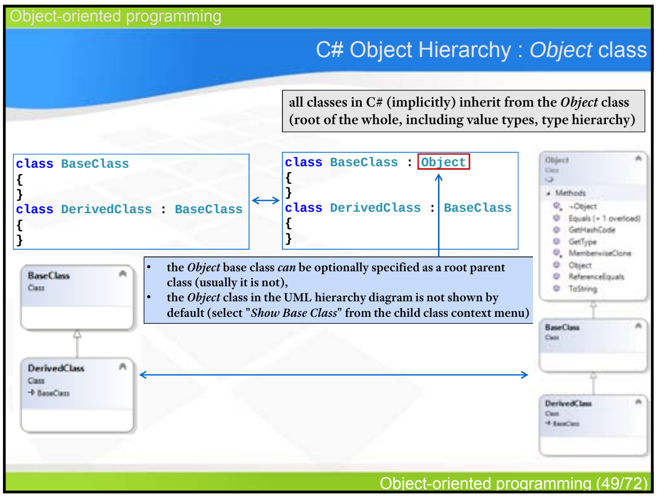
overriding virtual methods (#6/6)

```
class A
{
  public virtual void Speak()
  { Console.WriteLine("class A method invoked on behalf of object " + this); }
}
class B : A
{
  public override void Speak()
  { base.Speak();
    Console.WriteLine("class B method invoked on behalf of object " + this); }
}
class C : B
{
  public new void Speak()
  { base.Speak();
    Console.WriteLine("class C method invoked on behalf of object " + this); }
}
 // ...
A a = new A(); a.Speak();

B b = new B(); b.Speak();
a = b;         a.Speak();

C c = new C(); c.Speak();
a = c;         a.Speak();
b = c;         b.Speak();
```

```
class A method invoked on behalf of object C
class B method invoked on behalf of object C
```

Object-oriented programming (48/72)

48



Object-oriented programming

C# Object Hierarchy : *Object* class

all classes in C# (implicitly) inherit from the *Object* class (root of the whole, including value types, type hierarchy)

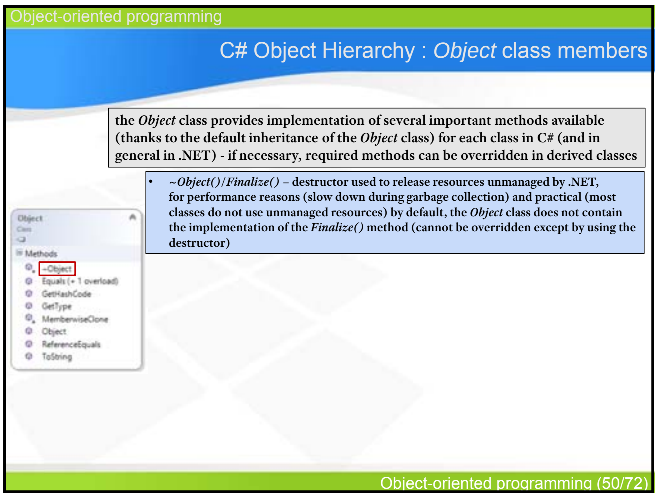
```
class BaseClass
{
}
class DerivedClass : BaseClass
{
}
```

```
class BaseClass : Object
{
}
class DerivedClass : BaseClass
{
}
```

- the *Object* base class *can* be optionally specified as a root parent class (usually it is not),
- the *Object* class in the UML hierarchy diagram is not shown by default (select "*Show Base Class*" from the child class context menu)

Object-oriented programming (49/72)

49



Object-oriented programming

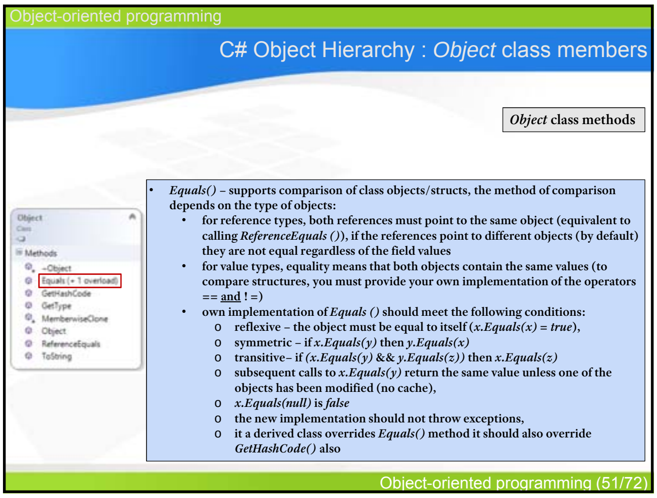
C# Object Hierarchy : *Object* class members

the *Object* class provides implementation of several important methods available (thanks to the default inheritance of the *Object* class) for each class in C# (and in general in .NET) - if necessary, required methods can be overridden in derived classes

- *~Object()/Finalize()* – destructor used to release resources unmanaged by .NET, for performance reasons (slow down during garbage collection) and practical (most classes do not use unmanaged resources) by default, the *Object* class does not contain the implementation of the *Finalize()* method (cannot be overridden except by using the destructor)

Object-oriented programming (50/72)

50



Object-oriented programming

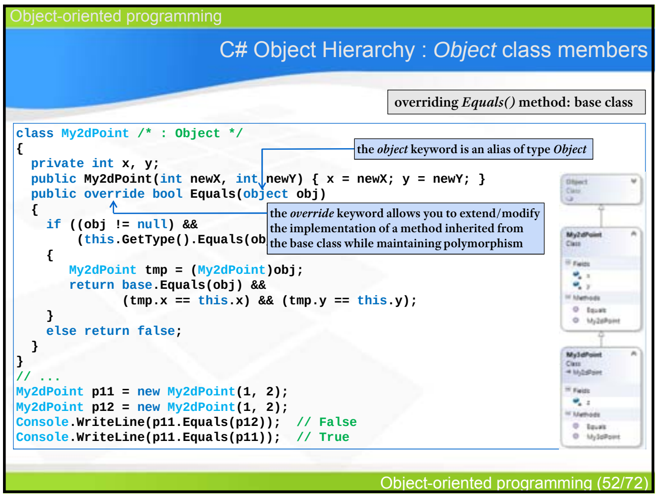
C# Object Hierarchy : *Object* class members

***Object* class methods**

- *Equals()* – supports comparison of class objects/structs, the method of comparison depends on the type of objects:
  - for reference types, both references must point to the same object (equivalent to calling *ReferenceEquals ()*), if the references point to different objects (by default) they are not equal regardless of the field values
  - for value types, equality means that both objects contain the same values (to compare structures, you must provide your own implementation of the operators == and ! =)
  - own implementation of *Equals ()* should meet the following conditions:
    - reflexive – the object must be equal to itself (*x.Equals(x) = true*),
    - symmetric – if *x.Equals(y)* then *y.Equals(x)*
    - transitive– if *(x.Equals(y) && y.Equals(z))* then *x.Equals(z)*
    - subsequent calls to *x.Equals(y)* return the same value unless one of the objects has been modified (no cache),
    - *x.Equals(null)* is *false*
    - the new implementation should not throw exceptions,
    - it a derived class overrides *Equals()* method it should also override *GetHashCode()* also

Object-oriented programming (51/72)

51



Object-oriented programming

C# Object Hierarchy : *Object* class members

overriding *Equals()* method: base class

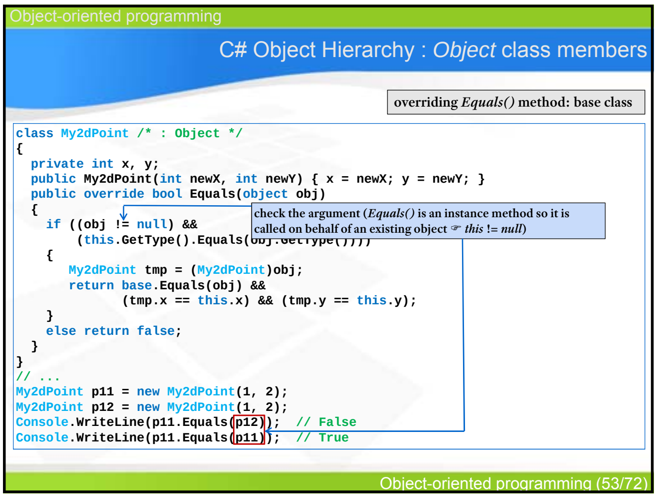
```
class My2dPoint /* : Object */
{
  private int x, y;
  public My2dPoint(int newX, int newY) { x = newX; y = newY; }
  public override bool Equals(object obj)
  {
    if ((obj != null) &&
        (this.GetType().Equals(ob
    {
      My2dPoint tmp = (My2dPoint)obj;
      return base.Equals(obj) &&
             (tmp.x == this.x) && (tmp.y == this.y);
    }
    else return false;
  }
}
// ...
My2dPoint p11 = new My2dPoint(1, 2);
My2dPoint p12 = new My2dPoint(1, 2);
Console.WriteLine(p11.Equals(p12));  // False
Console.WriteLine(p11.Equals(p11));  // True
```

the *object* keyword is an alias of type *Object*

the *override* keyword allows you to extend/modify the implementation of a method inherited from the base class while maintaining polymorphism

Object-oriented programming (52/72)

52



Object-oriented programming

C# Object Hierarchy : *Object* class members

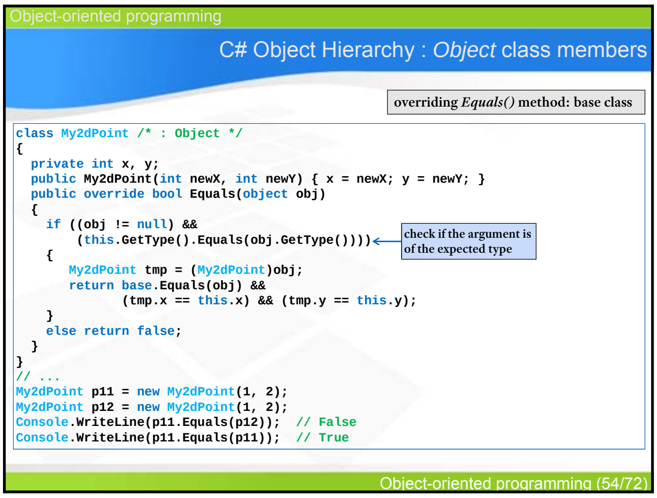
overriding *Equals()* method: base class

```
class My2dPoint /* : Object */
{
  private int x, y;
  public My2dPoint(int newX, int newY) { x = newX; y = newY; }
  public override bool Equals(object obj)
  {
    if ((obj != null) &&
        (this.GetType().Equals(obj.GetType()))))
    {
      My2dPoint tmp = (My2dPoint)obj;
      return base.Equals(obj) &&
             (tmp.x == this.x) && (tmp.y == this.y);
    }
    else return false;
  }
}
// ...
My2dPoint p11 = new My2dPoint(1, 2);
My2dPoint p12 = new My2dPoint(1, 2);
Console.WriteLine(p11.Equals(p12));  // False
Console.WriteLine(p11.Equals(p11));  // True
```

check the argument (*Equals()* is an instance method so it is called on behalf of an existing object ☞ *this != null*)

Object-oriented programming (53/72)

53



Object-oriented programming

C# Object Hierarchy : *Object* class members

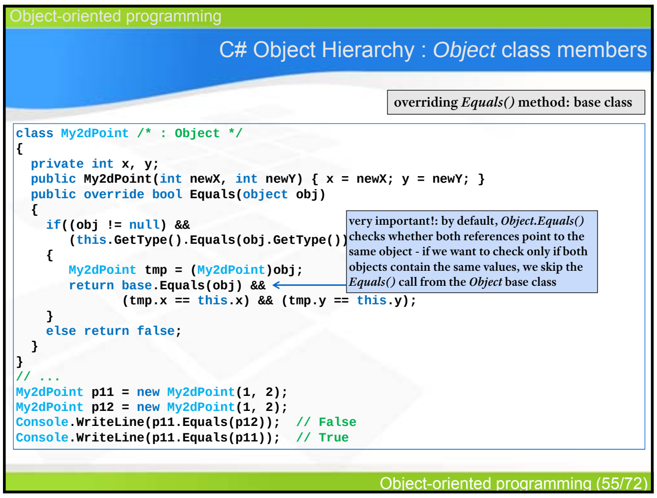
overriding *Equals()* method: base class

```
class My2dPoint /* : Object */
{
  private int x, y;
  public My2dPoint(int newX, int newY) { x = newX; y = newY; }
  public override bool Equals(object obj)
  {
    if ((obj != null) &&
        (this.GetType().Equals(obj.GetType())))
    {
      My2dPoint tmp = (My2dPoint)obj;
      return base.Equals(obj) &&
             (tmp.x == this.x) && (tmp.y == this.y);
    }
    else return false;
  }
}
// ...
My2dPoint p11 = new My2dPoint(1, 2);
My2dPoint p12 = new My2dPoint(1, 2);
Console.WriteLine(p11.Equals(p12));  // False
Console.WriteLine(p11.Equals(p11));  // True
```

check if the argument is of the expected type

Object-oriented programming (54/72)

54



Object-oriented programming

C# Object Hierarchy : *Object* class members

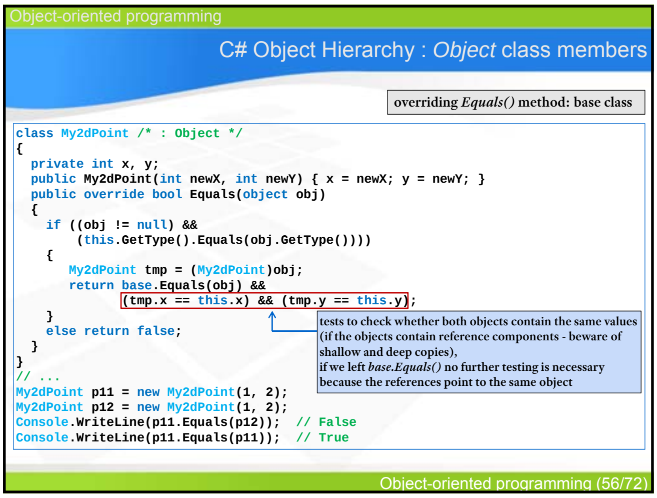
overriding *Equals()* method: base class

```
class My2dPoint /* : Object */
{
  private int x, y;
  public My2dPoint(int newX, int newY) { x = newX; y = newY; }
  public override bool Equals(object obj)
  {
    if((obj != null) &&
       (this.GetType().Equals(obj.GetType())))
    {
       My2dPoint tmp = (My2dPoint)obj;
       return base.Equals(obj) &&
              (tmp.x == this.x) && (tmp.y == this.y);
    }
    else return false;
  }
}
// ...
My2dPoint p11 = new My2dPoint(1, 2);
My2dPoint p12 = new My2dPoint(1, 2);
Console.WriteLine(p11.Equals(p12));  // False
Console.WriteLine(p11.Equals(p11));  // True
```

very important!: by default, *Object.Equals()* checks whether both references point to the same object - if we want to check only if both objects contain the same values, we skip the *Equals()* call from the *Object* base class

Object-oriented programming (55/72)

55



Object-oriented programming

C# Object Hierarchy : *Object* class members

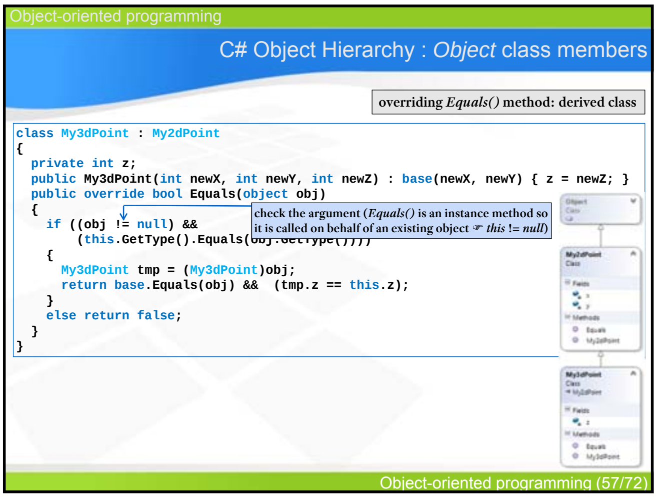
overriding *Equals()* method: base class

```
class My2dPoint /* : Object */
{
  private int x, y;
  public My2dPoint(int newX, int newY) { x = newX; y = newY; }
  public override bool Equals(object obj)
  {
    if ((obj != null) &&
        (this.GetType().Equals(obj.GetType())))
    {
       My2dPoint tmp = (My2dPoint)obj;
       return base.Equals(obj) &&
              (tmp.x == this.x) && (tmp.y == this.y);
    }
    else return false;
  }
}
// ...
My2dPoint p11 = new My2dPoint(1, 2);
My2dPoint p12 = new My2dPoint(1, 2);
Console.WriteLine(p11.Equals(p12));  // False
Console.WriteLine(p11.Equals(p11));  // True
```

tests to check whether both objects contain the same values (if the objects contain reference components - beware of shallow and deep copies),
if we left *base.Equals()* no further testing is necessary because the references point to the same object

Object-oriented programming (56/72)

56



Object-oriented programming

C# Object Hierarchy : *Object* class members

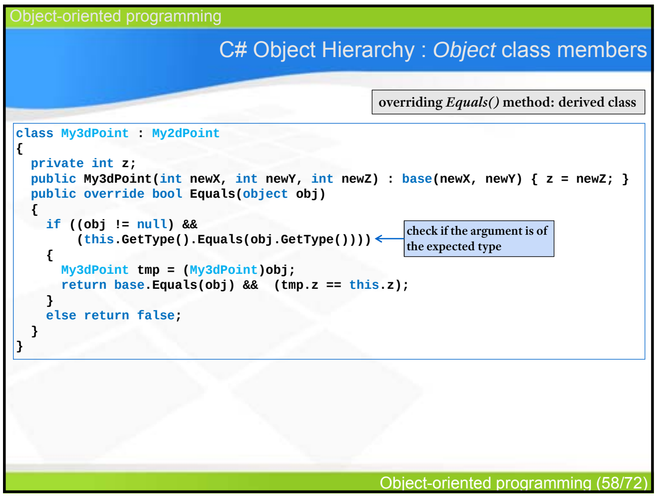
overriding *Equals()* method: derived class

```
class My3dPoint : My2dPoint
{
  private int z;
  public My3dPoint(int newX, int newY, int newZ) : base(newX, newY) { z = newZ; }
  public override bool Equals(object obj)
  {
    if ((obj != null) &&
        (this.GetType().Equals(obj.GetType())))
    {
      My3dPoint tmp = (My3dPoint)obj;
      return base.Equals(obj) &&  (tmp.z == this.z);
    }
    else return false;
  }
}
```

check the argument (*Equals()* is an instance method so it is called on behalf of an existing object ☞ *this* != *null*)

Object-oriented programming (57/72)

57



Object-oriented programming

C# Object Hierarchy : *Object* class members

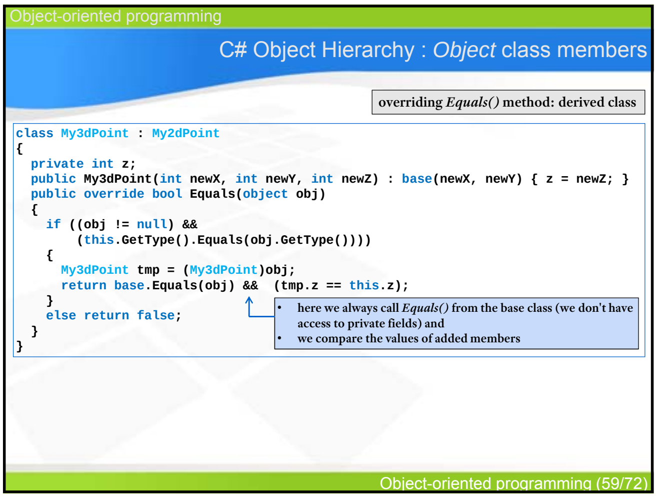
overriding *Equals()* method: derived class

```
class My3dPoint : My2dPoint
{
  private int z;
  public My3dPoint(int newX, int newY, int newZ) : base(newX, newY) { z = newZ; }
  public override bool Equals(object obj)
  {
    if ((obj != null) &&
        (this.GetType().Equals(obj.GetType())))
    {
      My3dPoint tmp = (My3dPoint)obj;
      return base.Equals(obj) &&  (tmp.z == this.z);
    }
    else return false;
  }
}
```

check if the argument is of the expected type

Object-oriented programming (58/72)

58



Object-oriented programming

C# Object Hierarchy : *Object* class members

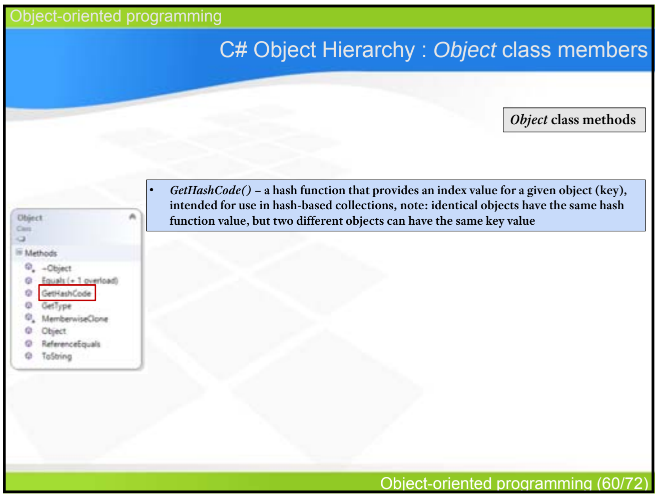
overriding *Equals()* method: derived class

```
class My3dPoint : My2dPoint
{
  private int z;
  public My3dPoint(int newX, int newY, int newZ) : base(newX, newY) { z = newZ; }
  public override bool Equals(object obj)
  {
    if ((obj != null) &&
        (this.GetType().Equals(obj.GetType())))
    {
      My3dPoint tmp = (My3dPoint)obj;
      return base.Equals(obj) &&  (tmp.z == this.z);
    }
    else return false;
  }
}
```

- here we always call *Equals()* from the base class (we don't have access to private fields) and
- we compare the values of added members

Object-oriented programming (59/72)

59



Object-oriented programming

C# Object Hierarchy : *Object* class members

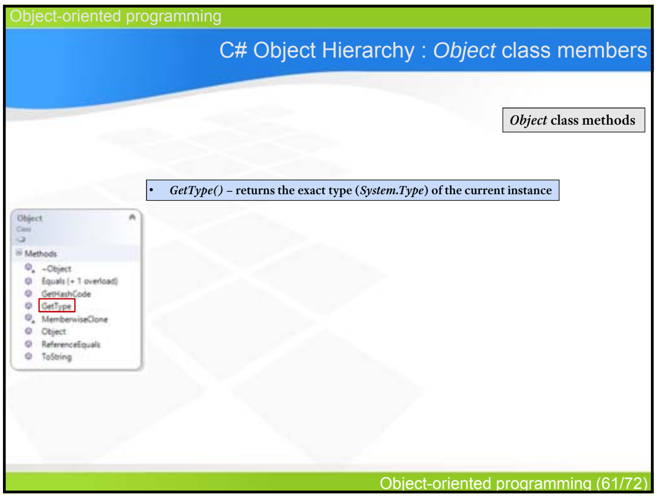
Object class methods

- *GetHashCode()* – a hash function that provides an index value for a given object (key), intended for use in hash-based collections, note: identical objects have the same hash function value, but two different objects can have the same key value

Object
Class
Methods
~Object
Equals (+ 1 overload)
GetHashCode
GetType
MemberwiseClone
Object
ReferenceEquals
ToString

Object-oriented programming (60/72)

60

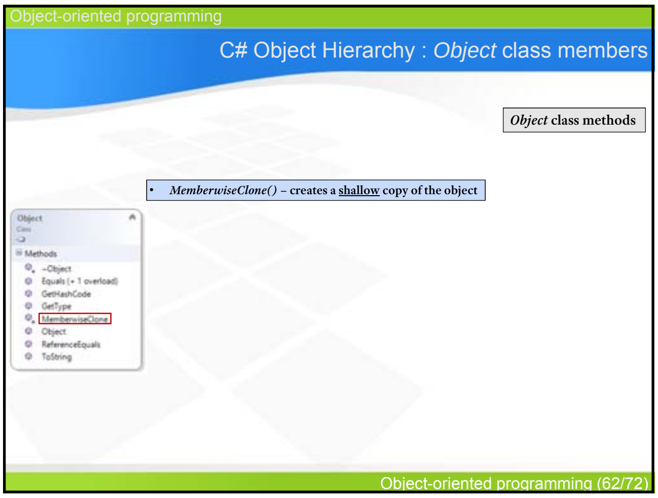


C# Object Hierarchy : *Object* class members

Object class methods

- *GetType()* – returns the exact type (*System.Type*) of the current instance

61

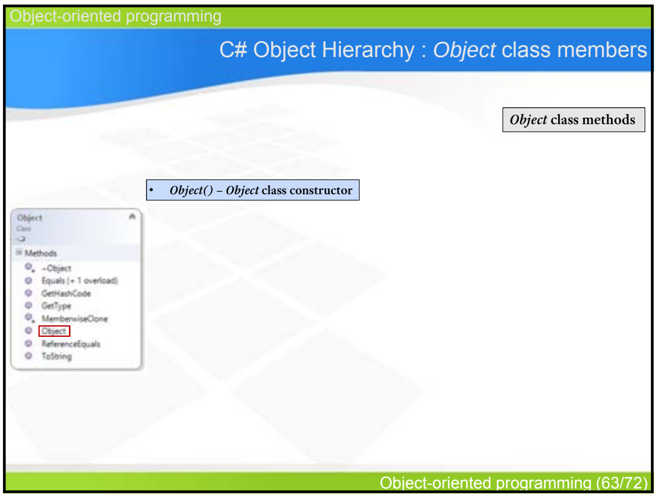


C# Object Hierarchy : *Object* class members

Object class methods

- *MemberwiseClone()* – creates a shallow copy of the object

62

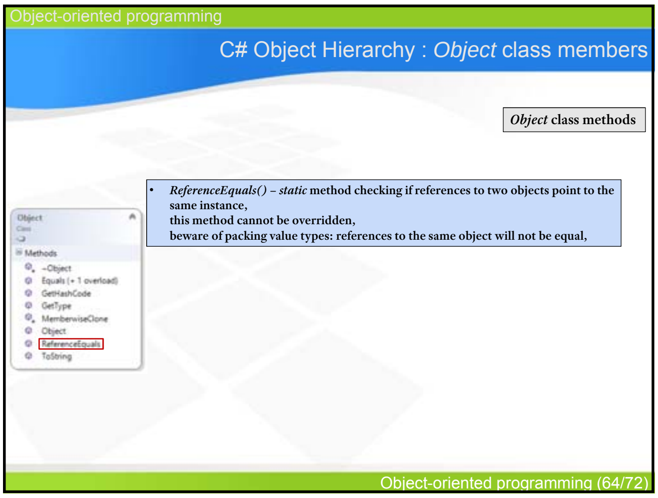


C# Object Hierarchy : *Object* class members

Object class methods

- *Object()* – *Object* class constructor

63

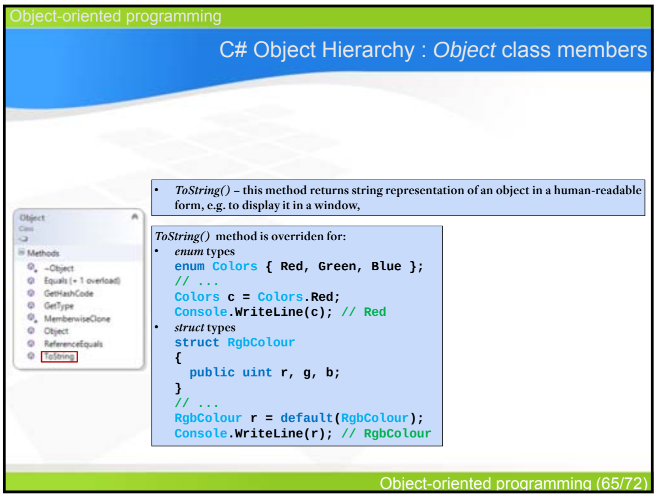


C# Object Hierarchy : *Object* class members

Object class methods

- *ReferenceEquals()* – *static* method checking if references to two objects point to the same instance,
  this method cannot be overridden,
  beware of packing value types: references to the same object will not be equal,

64



C# Object Hierarchy : *Object* class members

- *ToString()* – this method returns string representation of an object in a human-readable form, e.g. to display it in a window,

ToString() method is overriden for:

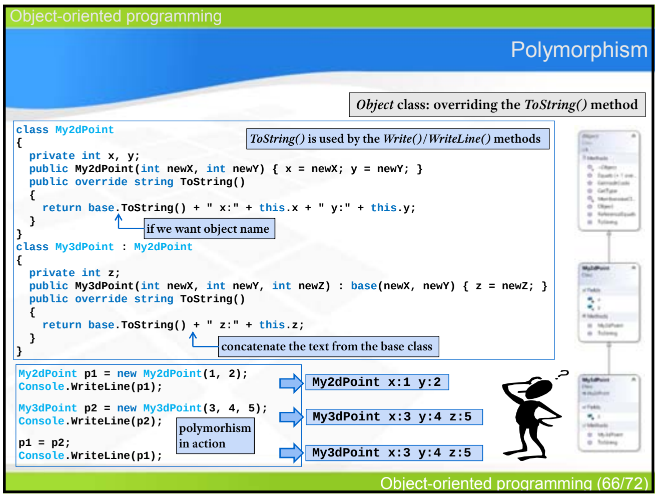
- *enum* types

```
enum Colors { Red, Green, Blue };
// ...
Colors c = Colors.Red;
Console.WriteLine(c); // Red
```

- *struct* types

```
struct RgbColour
{
  public uint r, g, b;
}
// ...
RgbColour r = default(RgbColour);
Console.WriteLine(r); // RgbColour
```

65



Polymorphism

Object class: overriding the *ToString()* method

ToString() is used by the *Write()*/*WriteLine()* methods

```
class My2dPoint
{
  private int x, y;
  public My2dPoint(int newX, int newY) { x = newX; y = newY; }
  public override string ToString()
  {
    return base.ToString() + " x:" + this.x + " y:" + this.y;
  }
}
class My3dPoint : My2dPoint
{
  private int z;
  public My3dPoint(int newX, int newY, int newZ) : base(newX, newY) { z = newZ; }
  public override string ToString()
  {
    return base.ToString() + " z:" + this.z;
  }
}
```

if we want object name

concatenate the text from the base class

```
My2dPoint p1 = new My2dPoint(1, 2);
Console.WriteLine(p1);

My3dPoint p2 = new My3dPoint(3, 4, 5);
Console.WriteLine(p2);

p1 = p2;
Console.WriteLine(p1);
```

polymorhism in action

```
My2dPoint x:1 y:2
```

```
My3dPoint x:3 y:4 z:5
```

```
My3dPoint x:3 y:4 z:5
```

66

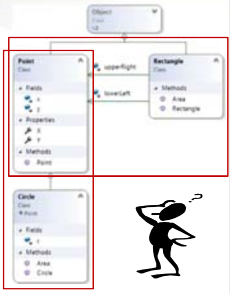
Composition „has-a” relationship (#1/2)

```
class Point
{
 private double x, y;
 public Point(double newX, double newY){ x = newX; y = newY; }
 public double X { get { return x; } }
 public double Y { get { return y;  } }
}
class Circle : Point
{
 private double r;
 public Circle(double newX, double newY, double newR)
     : base(newX, newY) { r = newR; }
 public double Area() { return Math.PI * r * r; }
}
class Rectangle
{
 private Point lowerLeft, upperRight;
 public Rectangle(double x1,double y1,double x2,double y2)
  {
   lowerLeft = new Point(x1, y1);
   upperRight = new Point(x2, y2);
  }
  public double Area()
  { return Math.Abs((lowerLeft.X-upperRight.X)*(lowerLeft.Y-upperRight.Y)); }
}
```

base class

derived class – "is-a" inheritance

"has-a" composition - instance of one class contains instance(s) of other class(es)



- aggregation - the class "has" object(s) of another class, but they can exist independently (the car "has" an engine)
- composition - a special case of aggregation, the owned object cannot exist separately (the book "has" a chapter)

Composition „has-a” relationship (#2/2)

comparison of "is-a" and "has-a" relationships

```
class Point
{
 private double x, y;
 public Point(double newX, double newY){ x = newX; y = newY; }
 public double X { get { return x; } }
 public double Y { get { return y;  } }
}
class Circle : Point
{
 private double r;
 public Circle(double newX, double newY, double newR)
     : base(newX, newY) { r = newR; }
 public double Area() { return Math.PI * r * r; }
}
class Rectangle
{
 private Point lowerLeft, upperRight;
 public Rectangle(double x1,double y1,double x2,double y2)
  {
   lowerLeft = new Point(x1, y1);
   upperRight = new Point(x2, y2);
  }
  public double Area()
  { return Math.Abs((lowerLeft.X-upperRight.X)*(lower
}
```

base class

- the child class inherits members of the base class,
- object creation and initialisation is performed out automatically by constructors
- the child class has access to the protected items of the base class

delegation: *lowerLeft* i *upperRight* are inaccessible from outside

- class contains objects of other classes
- independently creates and initialises the objects of contained classes
- there is no access to protected members of the included classes
- provides internal class services in the form of delegations

SOLID principles

SOLID – a mnemonic acronym for five design principles intended to make software designs more understandable, flexible and maintainable. A a subset of many principles promoted by American software engineer and instructor Roberta C. Martina (*Uncle Bob* ☺). They apply (not only) to any object-oriented design:

- *SRP* – Single Responsibility Principle: a class should only have a single responsibility, that is, only changes to one part of the software's specification should be able to affect the specification of the class,
- *OCP* – *Open-Closed Principle*: "Software entities ... should be open for extension, but closed for modification.", classes should be opened for extension (adding new code) but closed for modification (existing, checked code),
- *LSP* – *The Liskov Substitution Principle*: "Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.",
- *ISP* – *The Interface Segregation Principle*: "Many client-specific interfaces are better than one general-purpose interface.",
- *DIP* – *The Dependency Inversion Principle*: One should "depend upon abstractions, [not] concretions."

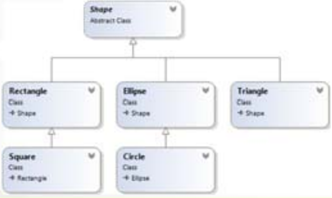
Should rectangle inherit from square?

The rectangle-square (or ellipse-circle) problem is a school example of problems that can be encountered when constructing a class hierarchy modelling relationships between real-world objects.

According to mathematical definitions, a square is a special case of a rectangle: one that has all sides of the same length. Similarly, a circle is a special example of an ellipse: one that has an equal large and small axis.

Direct mapping of mathematical rules into a hierarchy:

Such mapping may violate SOLID rules, in particular the LSP rule: the base class contains methods that operate on the object in a way that violates a more restrictive invariant in a derived class (= the square is more restrictive than the rectangle, the prisoner is not a special case of a person).



Should rectangle inherit from square?

Implementation:

must be overriden in derived class

```
abstract class Shape
{
 abstract public double Area { get; }
}
```

```
class Rectangle : Shape
{
  public virtual double Width
  {
    get;
    set;
  }
  public virtual double Height
  {
    get;
    set;
  }
  public override double Area
  {
    get { return Width * Height; }
  }
}
```

```
class Square : Rectangle
{
  public override double Width
  {
    get { return base.Width; }
    set { base.Width = base.Height = value; }
  }
  public override double Height
  {
    get { return base.Height; }
    set { base.Width = base.Height = value; }
    }
    // public override double Area
    // {
    //    get { return Width * Height; }
    // }
}
```

redundant

Should rectangle inherit from square?

```
Rectangle RealRectangle = new Rectangle();
RealRectangle.Width = 3;
RealRectangle.Height = 4;
Console.WriteLine("Area 12 vs {0}", RealRectangle.Area);
Square RealSquare = new Square();
RealSquare.Height = 3;
Console.WriteLine("Area 9 vs {0}", RealSquare.Area);
Rectangle RectangleSubtype = new Square();
RectangleSubtype.Width = 5;
Console.WriteLine("Area 25 vs {0}", RectangleSubtype.Area);
Rectangle[] rectangles=new Rectangle[] {RealRectangle,RealSquare,RectangleSubtype };
foreach (var rectangle in rectangles)
{
   Console.Write(rectangle + " " + rectangle.Area + '\t');
   rectangle.Width = rectangle.Width * 2;
   Console.WriteLine(rectangle.Area);
}
```

```
Area 12 vs 12
```

```
Area 9 vs 9
```

```
Area 25 vs 25
```

increase area 2x

```
Rectangle 12    24
Square 9        36
Square 25       100
```

- 💣 reverse hierarchy Rectangle-Square?
- ? derive both classes directly from *Shape*?
- ✓ interfaces!