



Vývoj aplikací v prostředí .NET

© Katedra řídicí techniky,
ČVUT-FEL Praha

5. přednáška

Virtuální metody



Omluva

*Vzhledem k tomu,
že se předmět přednáší
i v angličtině a není
v mých silách vyrobit
dvojjazyčné prezentace,
nejsou všechny následující
snímky v češtině.
Děkuji za pochopení.*



- Dědění rozšiřuje vlastnosti základní třídy přidáním nové funkčnosti
- Virtuální metoda **umí při dědění přesměrovat volání** původní metody v základní třídě na jinou metodu v odvozené třídě -> **změna se týká i všech metod základní třídy, které virtuální metodu používají.**
- Ale Java tohle umí běžně...
 - Ano, v Javě jsou všechny metody vždy virtuální
- V C++ a C# se můžeme rozhodnout, zda virtuality využijeme, či nikoliv.

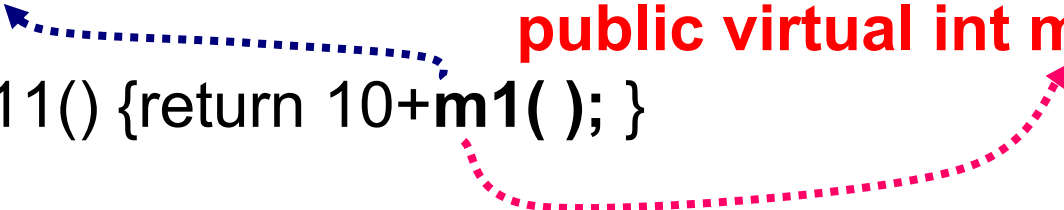
Virtuální versus nevirtuální metody

```
class AA
```

```
{
```

```
public int m1() {return 1;} ⇔
```

```
public virtual int m1() { return 1;}
```



```
public int m11() {return 10+m1( ); }
```

```
}
```

```
class BB : AA
```

```
{
```

```
public int i;
```

```
public new int m1() {return -1;} ⇔
```

```
public override int m1() {return -1;}
```

```
public BB() { i=m11();}
```

```
}
```

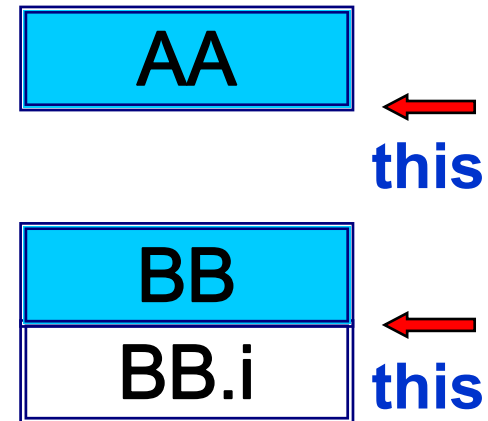
- *In C#, we must mark the redeclaration of `m1()` virtual method either by **override** keyword or by **new** keyword. It is required for increasing the comprehensibility of C# code.*
 - ***override** defines the replacement of `m1()` methods by a new code*
 - ***new** means new representations of `m1()` method*

Non-virtual Methods → Addresses

```
class AA {  
    public int m1() {return 1;}  
    public int m11() {return 10+m1( ); }  
}  
  
class BB : AA {  
    public int i;  
    public new int m1() {return -1;}  
    public BB() { i=m11(); }  
}
```

*Instances created by
constructors*

AA.m1() { return 1; }
AA.m11() { return 10+call AA.m1; }
BB.m1() { return -1;}
BB.BB() { i=call AA.m11(); }



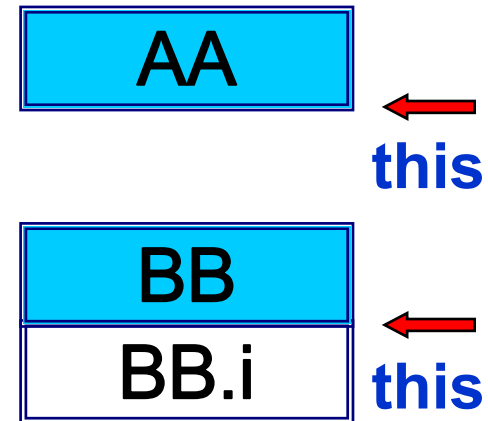
Inline Non-virtual methods

```
class AA {  
    public int m1() {return 1;}  
    public int m11() {return 10+m1( ); }  
}  
  
class BB : AA {  
    public int i;  
    public new int m1() {return -1;}  
    public BB() { i=m11(); }  
}
```

In "release" compilation, such methods are compiled by direct insertions of their codes (inline).

Instances created by constructors

AA.m1()	{ return 1; }	→	1
AA.m11()	{ return 11; }	→	11
BB.m1()	{ return -1; }	→	-1
BB.BB()	{ i=11; }	→	11



Virtual Methods → References

```
class AA {  
    public virtual int m1() {return 1;}  
    public int m11() {return 10+m1( ); }  
}  
  
class BB : AA {  
    public int i;  
    public override int m1() {return -1;}  
    public BB() { i=m11(); }  
}
```

Inline compilation is impossible!

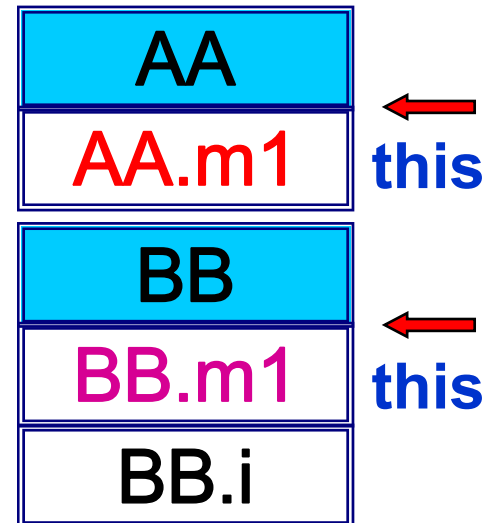
*Instances created by
constructors*

AA.m1() { return 1; }

AA.m11() {return 10+call [this+0]; }

BB.m1() { return -1; }

BB.BB() { i=call AA.m11(); }



Can we replace virtual methods?

- We can create the same effect without virtual methods with the aid of delegates = pointers to methods

```
delegate int m1Delegate();  
class AA  
{ protected int m1Virtual() { return 1; }  
  protected m1Delegate m1;    // destination of m1 call  
  public AA() { m1=m1Virtual; } // initialization of m1  
  public int m11() { return 10+ m1(); }  
}  
class BB : AA  
{ protected int m1Override() { return -1; }  
  public BB() : base() { m1=m1Override; }  
}
```

Non-virtual versus virtual methods 1/5

```
[STAThread] static void Main(string[] args) {
```

```
AA aa=new AA();
```

```
int x=aa.m1();           //      1      ⇔      1
```

```
x= aa.m11();            //     11     ⇔     11
```

Instance aa

→ **AA.m1() { return 1; }**

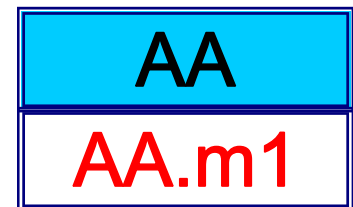
AA.m11() {return 10+call AA.m1; }

AA.m1() { return 1; }

AA.m11() { return 10+call [this+0]; }



← **this**



← **this**

```
BB bb=new BB();
```

`x= bb.m1();` `//` `-1` \Leftrightarrow `-1`

x= bb.m11(); *//* **11** \Leftrightarrow **9**

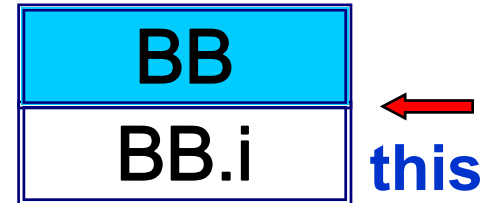
Instances created by constructors

```
AA.m1() { return 1; }
```

```
AA.m11() {return 10+call AA.m1; }
```

```
BB.m1() { return -1;}
```

```
BB.BB() { i=call AA.m11(); }
```

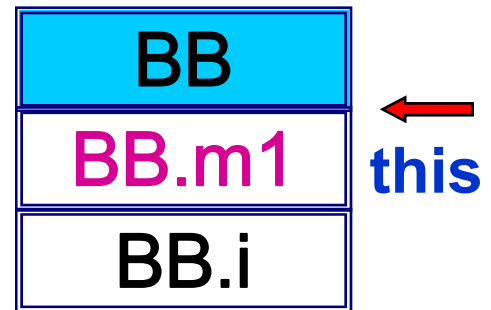


```
AA.m1() { return 1; }
```

```
AA.m11() {return 10+call [this+0]; }
```

```
BB.m1() { return -1;}
```

```
BB.BB() { i=call AA.m11(); }
```



Non-virtual versus virtual methods 3/5

If we create an instance with the aid of BB() constructor then m1() calls are redirected to m1() of BB class **public override int m1() { return -1; }**

The change of the instance type to its base class persists the destination method of m1() calls.

```
[STAThread] static void Main(string[] args) {
```

```
AA aab = new BB();
```

```
x= aab.m11();           //      11      ⇔      9
```

```
x= aab.m1();            //      1       ⇔     -1
```

```
}
```

Non-virtual versus virtual methods 4/5

AA baa = new BB();

x= baa.m11(); // 11 ⇔ 9

x= baa.m1(); // **1** ⇔ **-1**

AA.m1() { return 1; }

AA.m11() {return 10+call AA.m1; }

BB.m1() { return -1;}

BB.BB() { i=call AA.m11(); }

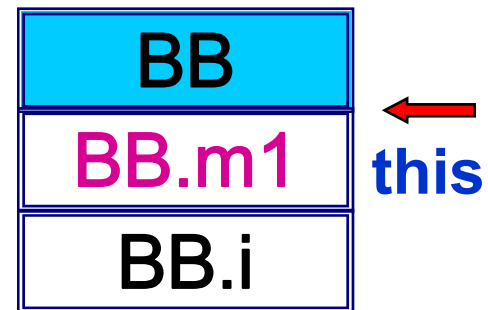
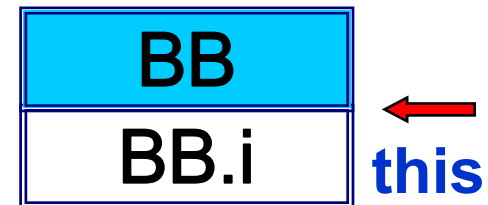
AA.m1() { return 1; }

AA.m11() {return 10+call [this+0]; }

BB.m1() { return -1;}

BB.BB() { i=call AA.m11(); }

Instances created by constructors



Kdy se projeví virtualita? 4/4 Souhrn

```
[STAThread] static void Main(string[] args) {
```

```
AA aa=new AA();
```

```
int x=aa.m1();
```

```
x= aa.m11();
```

```
BB bb=new BB();
```

```
x= bb.m1();
```

```
x= bb.m11();
```

```
AA baa = new BB();
```

```
x= baa.m11();
```

```
x= baa.m1();
```

```
}
```

Deklarace metody AA.m1 jako virtuální se v následujících 2 řádcích neprojeví – třídu vytvořil konstruktor AA

// 1 ⇔ 1

// 11 ⇔ 11

Deklarace metody AA.m1 jako virtuální se neprojeví ani v následujícím řádku – zde se vždy volá metoda BB.m1

// -1 ⇔ -1

// 11 ⇔ 9 Případ A

viz další snímek

// 11 ⇔ 9 Opět Případ A

// 1 ⇔ -1 Případ B

■ Cíl virtuální metody se určuje při vytváření instance objektu

- ☐ cíl závisí jen použitém konstruktoru objektu
- ☐ virtuální metodu lze změnit jedině děděním
- ☐ **změněná virtuální metoda volá novou funkci**

Případ A ■ ...novou funkci volají i všechny další metody základní třídy, jejichž kód obsahuje odkaz na virtuální metodu

Případ B ■ ...na novou funkci směřuje i přímé volání virtuální metody bez ohledu na přetypování

- An override method provides a new implementation of a member inherited from a base class. The method overridden by an override declaration is known as the overridden base method. The overridden base method must have the same signature as the override method.
- **You cannot override a non-virtual or static method.** The overridden base method must be virtual, abstract, or override.
- **An override declaration cannot change the accessibility** of the virtual method. Both the override method and the virtual method must have the same access level modifier.
- You cannot use the modifiers new, static, virtual, or abstract to modify an override method.
- An overriding property declaration must specify **the exact same access modifier**, type, and name as the inherited property, and the overridden property must be virtual, abstract, or override.

- *Virtual methods are less effective than non/virtual methods; they require more memory for necessary virtual tables and they are called by slower indirect addresses. It is also impossible to compile them by inline insertions of their codes.*
- *In the lecture, we placed virtual table in instances of classes, i.e. each instance has its own virtual tables. It increases invocations of methods, but it consumes more memory. There are other more frequent methods.*
 - *For instance, instances of classes created by the same constructor can share their virtual tables. They contain only references to them. Such compilation save memory, but it requires one additional indirect addressing. Before invoking virtual method, a program must read the reference to a virtual table.*

override: C# versus C++

C# code

```
class AA
{
    public virtual int m1() { return 1;}
    public int m11() {return 10+m1(); }
}
class BB : AA
{
    public int i;
    public override int m1()
    {return -1;}
    public BB() { i=m11();}
}
```

C++ code

```
≡ class AA
{ public:
    virtual int m1() {return 1;}
    int m11() {return 10+m1(); }
};
class BB : AA
{ public:
    int i;
    virtual int m1() {return -1;}
    BB() { i=m11();}
}; // virtual has no influence, it is only recommended
```

override: C# versus Java

C# code

```
class AA
{
    public virtual int m1()
        { return 1;}
    public int m11() {return 10+m1(); }
}
class BB : AA
{
    public int i;
    public override int m1() {return
        -1;}
    public BB() { i=m11();}
}
```

Java code

```
≡ class AA
{ public int m1() {return 1;}
  public int m11()
    { return 10+m1(); }
}
class BB extends AA
{ public int i;
  public int m1() {return -1;}
  public BB() { i=m11();}
}
```

C# code

```
class AA
{ // virtual nemá zde vliv na BB
  public virtual int m1() {return 1;}
  public int m11() {return 10+m1(); }
}
class BB : AA
{
  public int i;
  public new int m1() {return -1;}
  public BB() { i=m11();}
}
```

C++ code

```
≡ class AA
{ public:
    int m1() {return 1;}
    int m11() {return 10+m1(); }
};
class BB : AA
{ public:
    int i;
    int m1() {return -1;}
    BB() { i=m11();}
};
```

new: C# versus Java

C# code

```
class AA
{
    public virtual int m1() {return 1;}
    public int m11() {return 10+m1(); }
}
class BB : AA
{
    public int i;
    public new int m1() {return -1;}
    public BB() { i=m11();}
}
```

≈

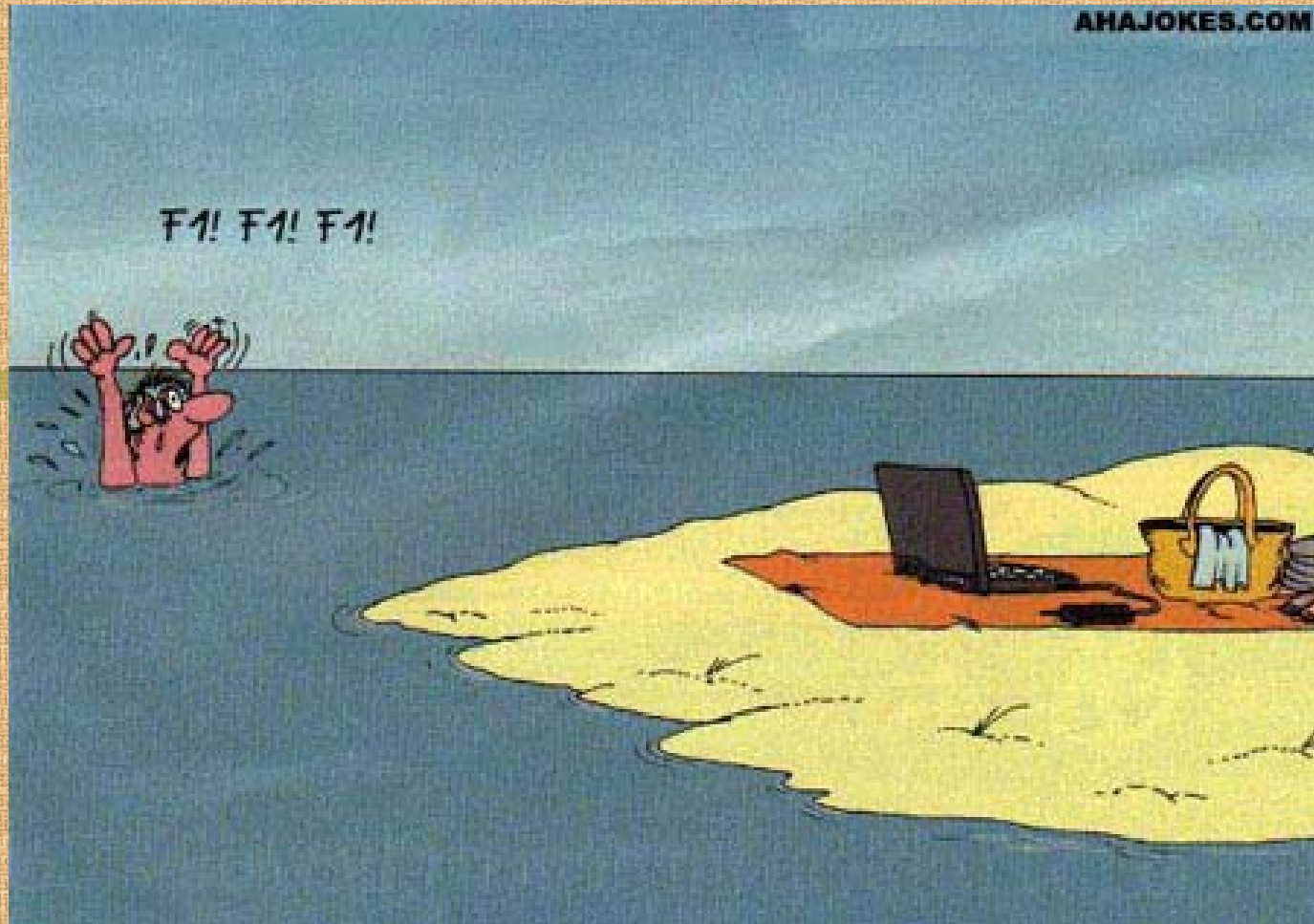
Java code

```
class AA
{
    public static int m1() {return 1;}
    public int m11() {return 10+m1(); }
}
class BB extends AA
{
    public int i;
    public static int m1() {return -1;}
    public BB() { i=m11();}
}
```

?

/ There is no direct equivalent,
because static method have
different behavior*/*

Virtual methods – what for?



Usage of Virtual Methods 1/2

- *The implementation of a virtual member can be changed by an overriding member in a derived class. By this way, we also change behavior of all methods that use virtual member*
- *In a way, virtual methods are replaceable drawers in base classes* →



*Burning Giraffe (Woman with Drawers),
Salvador Dali, 1937, Kunstmuseum Basel*

1st example: modifying .NET debugger

class Pos

```
{ protected int x, y;
```

```
/*...*/
```

```
public Pos(int x, int y) { this.x = x; this.y = y; }
```

```
public Pos() : this(0, 0) { }
```

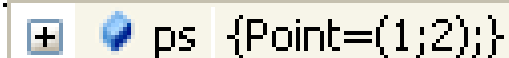
// virtual method defined in Object, implicit base class

```
public override string ToString()
```

```
{ return String.Format("Point=({0};{1});", x, y); }
```

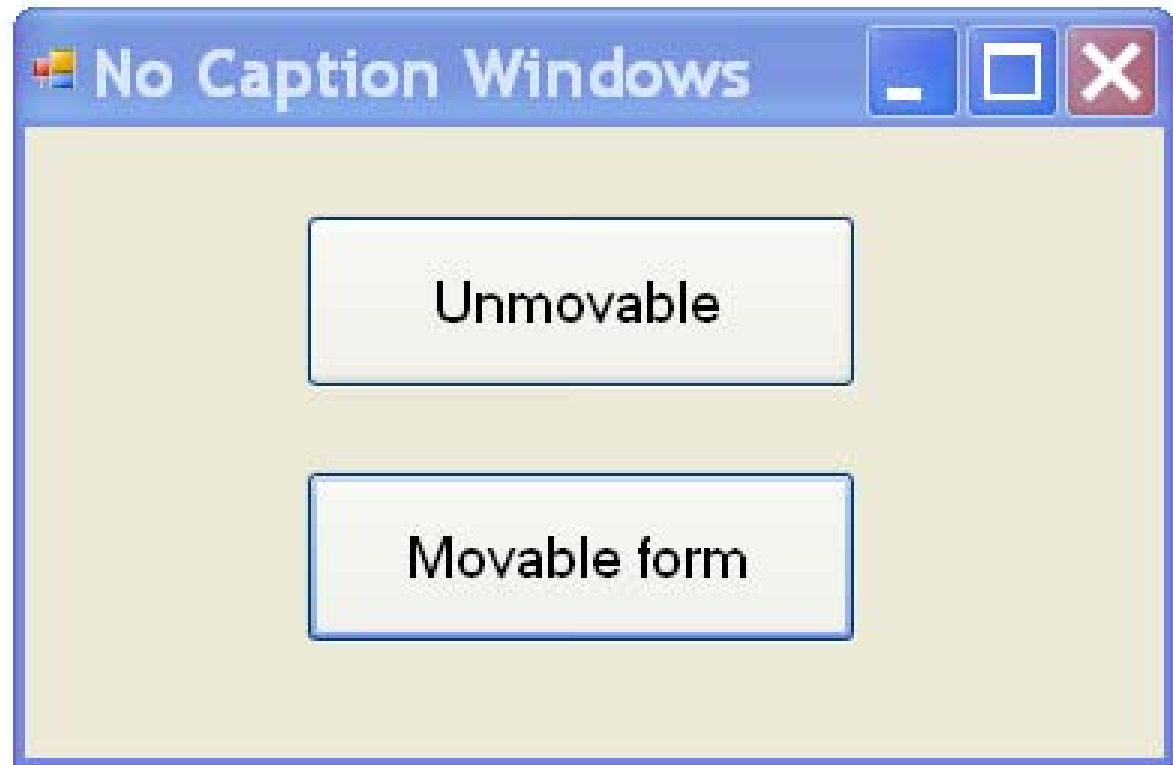
```
}
```

```
Pos ps = new Pos(1, 2);
```

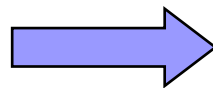
A screenshot of a debugger tooltip. It shows a plus icon, a blue dot icon, the variable name 'ps', and the string representation '{Point=(1;2);}'.

Debugger displays tooltips containing ToString() results.

2nd example Window without caption



*How to move
it to a new
position?*



`FormBorderStyle=None`

Close

2nd example Movable window without caption

```
public partial class FormMovable : Form
{
    public FormMovable() { InitializeComponent(); }
    protected override void WndProc(ref Message m)
    {
        //http://msdn.microsoft.com/en-us/library/ms645618(VS.85).aspx
        if (m.Msg == 0x0084) //WM_NCHITTEST - non-client-hit-test
        {
            base.WndProc(ref m);
            if (m.Result == (IntPtr)1) //HTCLIENT - client area
                m.Result = (IntPtr)2; //HTCAPTION - caption
        }
        else base.WndProc(ref m);
    }
}
```


Usage of Virtual Methods 2/2

- *They allow creating collections of object inherited for a same base class with virtual methods. Each derived class has it own override implementation of the virtual methods – they are invoked with the aid of type casting.*



Life Is Full Of Difficult Decisions

<http://imagecache2.allposters.com/>

3rd example: Collections 1/2

```
class VEC
```

```
{ public virtual string name() { return "??";}  
  public void AddName(string s) { s = s+name()+'.';}  
};
```

```
class Circle : VEC
```

```
{ public override string name() {return "Circle";}  
};
```

```
class Square : VEC
```

```
{ public override string name() {return "Square";}  
};
```


3rd example: Collections 2/2

```
[STAThread] static void Main(string[] args)
{
    // dynamic array with collection of references
    ArrayList list = new ArrayList();
    // constructor redirects name() to Circle.name() / Square.name()
    list.Add(new Circle()); list.Add(new Square());
    list.Add(new Circle());
    string s="";
    // we apply method name() to all instances in the collection
    foreach(object o in list)
        s = s + ( (VEC) o ).name()+". ";
    Console.WriteLine(s);
    // result "Circle.Square.Circle."
}
```

// we call directly VEC.AddName()

foreach(object o in list)

((VEC) o).AddName(s2);

Console.WriteLine(s);

// output “Circle.Square.Circle.”

```
class VEC
```

```
{ public virtual string name() { return "??";}
```

```
    public void AddName(string s) { s = s+name()+'.';}
```

```
};
```

Vícenásobná dědičnost

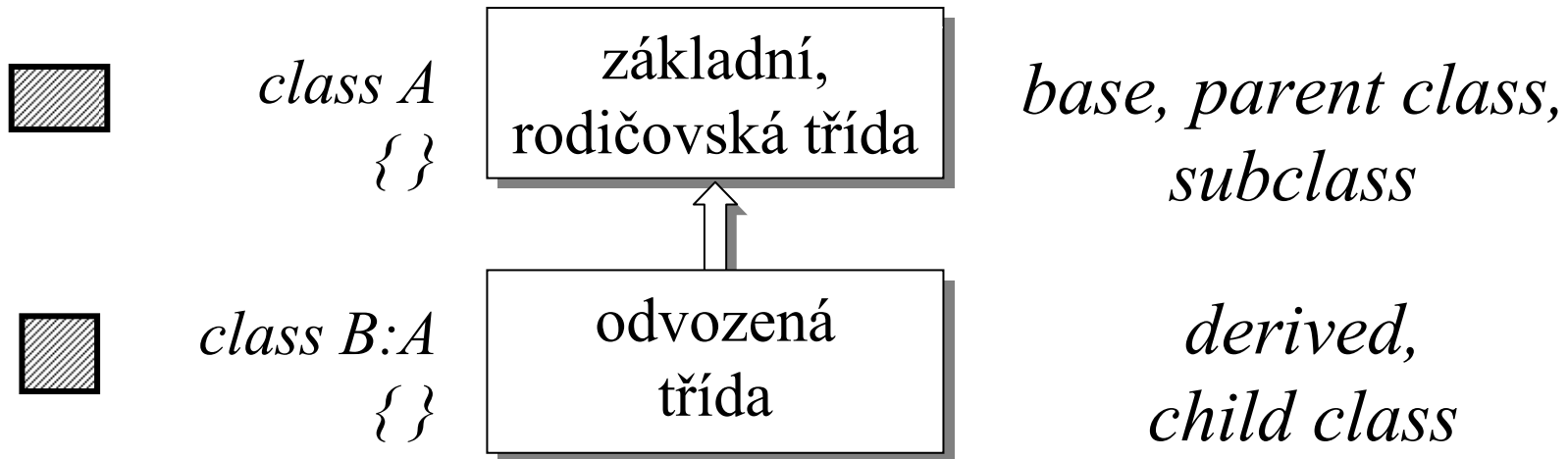
výhody
i problémy



Problémy dědičnosti objektů

- Dědičnost se koncepčně považuje za zobecnění vztahu "být", avšak její konkrétní implementace mívá své vlastnosti.
- Jsou čtvercové formy odvozeny od obdélníků?

[Baclawski, Indurkha, 1994].



■ *Jsou čtvercové formy odvozeny od obdélníků?*

□ Jak kdy ...

- všechny operace dovolené s obdélníky
nejdou provést se čtverci bez ztráty jejich
"čtvercovitosti", jako například zvětšení nebo
zmenšení podle obecného měřítka $m_x \neq m_y$

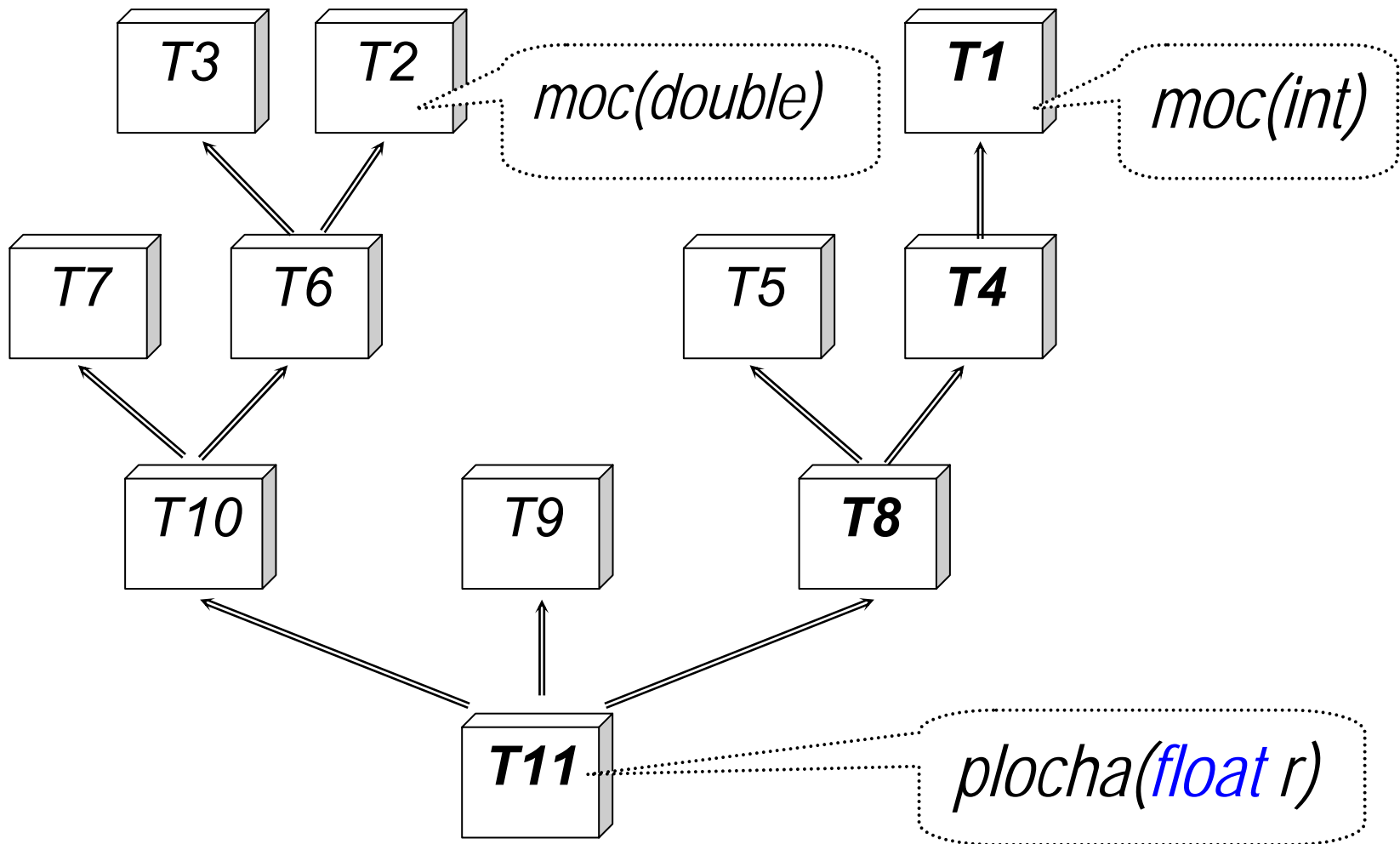
Dědění vytváří vedlejší závislosti mezi třídami!

Vícenásobná dědičnost v C++ 2/3

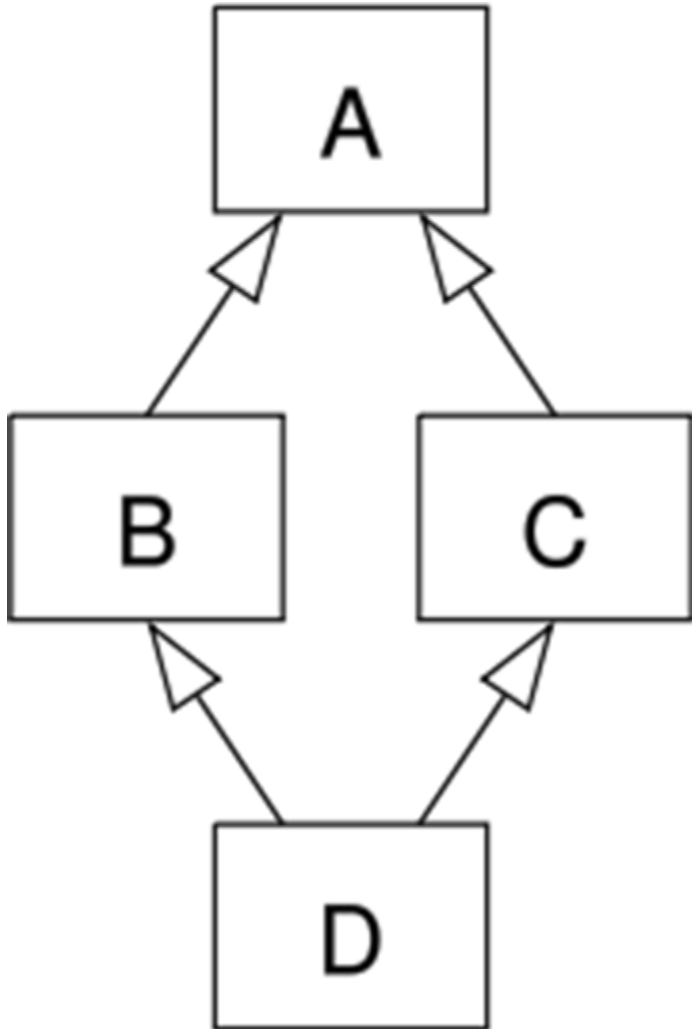
```
class T1 { public: int  moc(int i) { return i*i; } };
class T2 { double moc(double d) { return d*d; } /* ... */ };
class T3 { /* ... */ }; class T4 : public T1 { /* ... */ };
class T5 { /* ... */ };
class T6 : public T3, public T2 { /* ... */ };
class T7 { /* ... */ };
class T8 : public T5, public T4 { /* ... */ };
class T9 { /* ... */ };
class T10 : public T7, public T6 { /* ... */ };
class T11 : public T10, public T9, public T8
{
    public: float plocha(float r) { return M_PI * moc(r); }
};
int _tmain(int argc, _TCHAR* argv[])
{ T1 t1; t1.plocha(5); return 0; }
```

Překlad dá chybu: "ambiguous access of 'moc' in 'T11'."

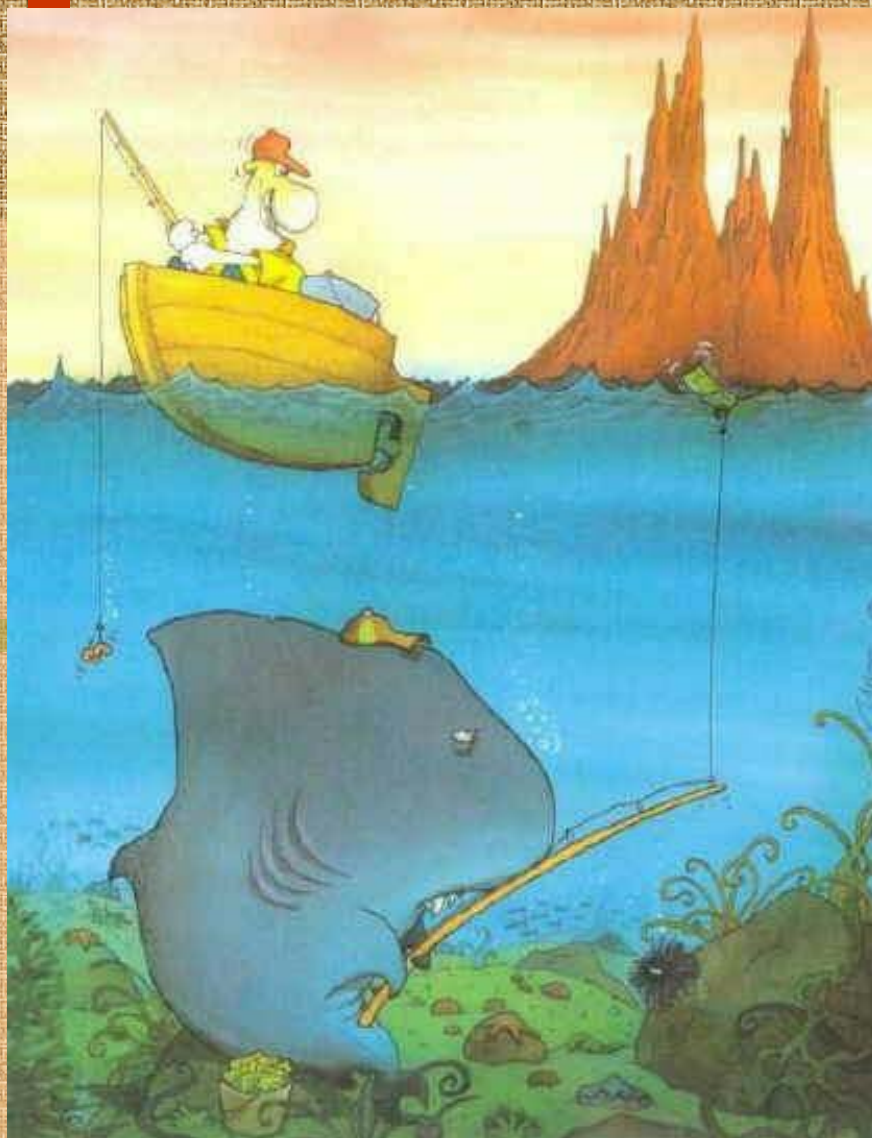
Výsledná struktura



Multiple inheritance - diamond problem



- *There is debate as to whether implementing and using multiple inheritance is easier than using single inheritance and software design patterns...*
- *but we needed multiple inheritance to describe more complex objects!*



Interface

Interface v C# je podobná Javě

*Jazyk C# používá shodný koncept jako Java. Povoluje vícenásobnou dědičnost jen speciální třídám - **interface**.*

- interface = čistě abstraktní třída obsahující výhradně deklarace hlaviček.
- pouze předepisuje tvar - nesmí zahrnovat implementaci.
- má implicitně všechny členy **public abstract (virtual)** - před nimiž nesmí být jakékoliv modifikátory!
- obsahuje výhradně metody (*methods*), vlastnosti (*properties*), indexery (*indexers*) a události (*events*).
- nesmí obsahovat pole (*fields*), konstanty, konstruktory, destruktory, operátory, zřetězené deklarace tříd.
- nesmí mít statické prvky.
- smí rozšiřovat deklarace jiných interface.
- dovoluje vícenásobné dědění ve strukturách, třídách i jiných interface.

```
public interface IList : ICollection, IEnumerable
{ int Add (object value); // předpis tvaru metody
    /*...*/
    bool IsReadOnly { get; } // předpis struktury property
    /*...*/
    object this [int index] { get; set; }
                                // předpis struktury pro indexer
}
```

Interface je předpis povinné struktury a tu lze dědit několikanásobně:

```
public interface I1 { void m(); }
```

```
public interface I2 { void m(); }
```

```
public interface I3 { void m(); }
```

```
public interface I4: I1, I2 { void m2(); }
```

```
// možno i new void m();
```

```
public class A { /* ... nějaké deklarace ... */ }
```

```
public class B : A, I1
```

```
{ public void m() { Console.WriteLine("M");} }
```

```
public class C : B, I1, I3, I4
```

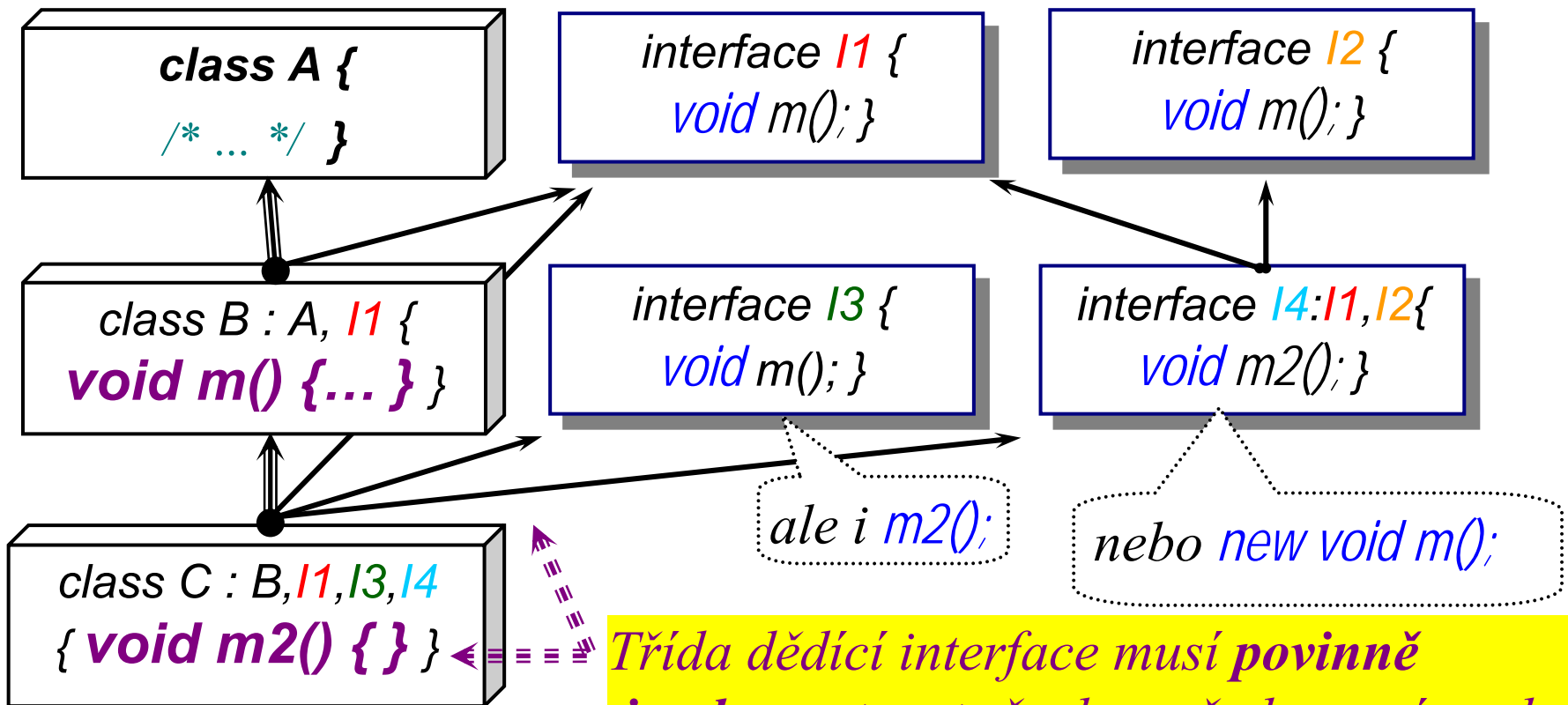
```
{ public void m2() { Console.WriteLine("M2");} }
```

Vzniklá struktura

```
public class A { /* ... nějaké deklarace ... */ }
```

```
public class B : A, I1 { public void m() { Console.WriteLine("M");} }
```

```
public class C : B, I1, I3, I4  
{ public void m2() { Console.WriteLine("M2");} }
```



Třída dědící interface musí povinně implementovat všechny předepsané prvky!

Přetypování na interface

Třidu můžeme vždy přetypovat na typ některé její interface!

```
[STAThread] static void Main(string[] args)
{ C c = new C(); c.m(); c.m2();           // výstup M // výstup M2
  I1 i1 = c; i1.m();                     // výstup M
  I2 i2 = c; i2.m();                     // výstup M
  I3 i3 = c; i3.m();                     // výstup M
  I4 i4 = c; i4.m2();                    // výstup M2
}
```

```
public class A { /* ... nějaké deklarace ... */ }
public class B : A, I1 { public void m() { Console.WriteLine("M");} }
public class C : B, I1, I3, I4
{ public void m2() { Console.WriteLine("M2");} }
```


BubbleSort bez interface - nepoužitelný jinde

```
class Bod                                // třída Bod
{
    public int x, y;
    public int GetMax() { return x>y ? x : y; }
    public Bod(int x0, int y0) { x=x0; y=y0; }
}

class BubbleSort
{
    static public void Sort(Bod [ ] pole)
    {
        bool ok = false; Bod tmp;
        for(int i=pole.Length; i>1 && !ok; i--)
        {
            ok = true;
            for(int j=0; j<i-1; ++j)
            {
                if(pole[j].GetMax()>pole[j+1].GetMax())
                { tmp=pole[j]; pole[j]=pole[j+1]; pole[j+1]=tmp; ok=false;} }
            } // Sort()
        } // Bubble Sort
    }
```


Volání BubbleSort metody

```
[STAThread] static void Main(string[] args)
{ Bod [ ] body = { new Bod(8,7), new Bod(3,7),
                  new Bod(5,1), new Bod(1,3) };
  BubbleSort.Sort(body);
  foreach(Bod b in body)
      Console.WriteLine("{0},{1} ",b.x,b.y);
      //výstup (1,3) (5,1) (3,7) (8,7)
  Thread.Sleep(Timeout.Infinite);
      // zastav vlákno - sám si zavřu okno
      // alternativa Thread.Sleep(10000); //10000 ms
}
```

IComparable interface

```
interface IComparable // definováno v knihovně .NET
{
    int CompareTo(object obj);
    // výsledek <0: this<obj, 0: this==obj, >0: this>obj
}
```

```
class Bodl : Bod, IComparable // přidáme interface
{
    public Bodl(int x0, int y0): base(x0,y0) { }
```

// implementace metody z interface musí být typu public

```
    public int CompareTo(object obj)
    {
        return GetMax() - ( (Bodl) obj
    ).GetMax();
}
```

```
class BubbleSortI
```

```
{ /* třídění dovoluje teď zpracovat každou třídu, která zahrnuje  
   IComparable interface */
```

```
static public void Sort(IComparable [ ] pole)
```

```
{ bool ok = false; IComparable tmp;  
  for(int i=pole.Length; i>1 && !ok; i--)  
  { ok = true;  
    for(int j=0; j<i-1; ++j)  
      if(pole[ j ].CompareTo(pole[ j+1 ])>0)  
      { tmp=pole[j]; pole[j]=pole[j+1];  
        pole[j+1]=tmp; ok=false; }  
    }  
  }  
}
```

```
[STAThread] static void Main(string[ ] args)
{
    Bodl [ ] body2 = {new Bodl(8,7), new Bodl(3,7),
                        new Bodl(5,1), new Bodl(1,3) };
    BubbleSortl.Sort(body2);
    foreach(Bodl b in body2)
        Console.WriteLine("{0},{1} ",b.x,b.y);
    Thread.Sleep(Timeout.Infinite);
    // zastav vlákno - sám si zavřu okno
}
}
```

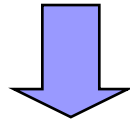
Proč jsme nepoužili metodu Swap()?

```
static void Swap(ref IComparable o1, ref  
IComparable o2)
```

```
{ IComparable tmp; tmp=o1; o1=o2; o2=tmp; }
```

```
/* ..... */
```

```
IComparable tmp=pole[j];  
pole[j]=pole[j+1]; pole[j+1]=tmp;
```



```
Swap(ref pole[j], ref pole[j+1]);
```

Proč jsme nepoužili metodu Swap()?

```
class BubbleSortI // třídění prvků s IComparable
{
    static void Swap(ref IComparable o1, ref IComparable o2)
    {
        IComparable tmp; tmp=o1; o1=o2; o2=tmp;
    }
    static public void Sort(IComparable [] pole)
    {
        /* příkazy for-cyklu */
        if(pole[j].CompareTo(pole[j+1])>0)
        {
            // tmp=pole[j]; pole[j]=pole[j+1]; pole[j+1]=tmp; // OK
            💣 Swap(ref pole[ j ], ref pole[ j+1]);
            💣 // kód metající výjimku
            ok=false;
        }
    }
}
```

Vysvětlení Erica Gunnersona, člena C# vývojového týmu

```
static void Swap( ref IComparable o1, ref IComparable o2)  
{ IComparable tmp; tmp=o1; o1=o2; o2=tmp; }
```

C# nedovolí volání podobné Swap funkce (ale C++ ano)

Swap(ref pole[j], ref pole[j+1]);

*v tomto případě, protože uvedený kód by nebyl typově bezpečný.
Kdyby se povolil, pak by se vykonalo i volání funkce:*

```
public static void Swap(ref object a, ref object b)  
    {a = 5; b = "Hello"; }
```

■ ***Předávají-li se reference na prvky pole,
musí se použít původní typ.***

Možné použití Swap

```
static void Swap2(ref Bodl b1, ref Bodl b2)
```

```
{ Bodl tmp; tmp = b1; b1=b2; b2=tmp; }
```

```
/*... v Main() pak užíjeme třeba...*/
```

```
Bodl [ ] bodyA = { new Bodl(8,7), new Bodl(3,7),  
                    new Bodl(5,1), new Bodl(1,3) };
```

```
Swap2(ref bodyA[0], ref bodyA[3]);
```

Tato Swap() by se dala v BubbleSortI použít jedině takto

```
IComparable b1=pole[j], b2=pole[j+1];
```

```
Swap(ref b1, ref b2); pole[j]=b1; pole[j+1]=b2;
```

Podobná konstrukce nepřináší žádnou úsporu kódu.

- *Třída Array nabízí statickou metodu Sort() s implementací QuickSort třídění. Ta vyžaduje rozhraní IComparable - deklarace naší BodI ho již obsahuje:*

```
BodI [ ] bodyA = { new BodI(8,7), new BodI(3,7),  
                  new BodI(5,1), new BodI(1,3) };
```

```
// voláme statickou metodu Sort() třídy Array  
Array.Sort(bodyA);
```

```
foreach(BodI b in bodyA)  
    Console.WriteLine("{0},{1} ",b.x, b.y);
```

- Metoda QuickSort třídí sice s průměrnou složitostí $O(n \log n)$, *ale v nejhorším případě se složitostí $O(n^2)$ – to nastává někdy, pokud třídíme již téměř utříděné pole.*
- *Nejhorší případ pro BubbleSort má také $O(n^2)$, nejlepší $O(n)$.*
- *Nejhorší případ pro BubbleSort ovšem nastává za jiné situace než u QuickSort, takže BubbleSort může někdy být i rychlejší než QuickSort 😊*
- *Závěr: Každý způsob třídění se hodí pro jinou situaci.*

Literatura pro zájemce:

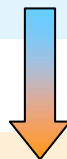
http://en.wikipedia.org/wiki/Category:Sort_algorithms

- IDisposable je také rozhraní, ale bude o **jeho správné** definici bude více až při výkladu správy paměti

```
interface IDisposable
{ void Dispose();
}
```

```
class MojeTridaSDispose : IDisposable
{ public void Dispose() { /* rušení systémových zdrojů */ }
}
```

```
using (MojeTridaSDispose mtd = new MojeTridaSDispose ())
{ /* operace s mojí třídou */
}
```



```
// výsledek překladač  
try { MojeTridaSDispose mtd = new MojeTridaSDispose();  
    /* operace s mojí třídou */  
}  
finally { if (mtd != null) mtd.Dispose(); }
```

Explicitní specifikace rozhraní 1/2

```
interface Prvni { void Proved(); }
interface Druhe { void Proved(); }
class Tester : Prvni, Druhe
{
    public void Proved() { Console.WriteLine("Proved"); }
}
[STAThread] static void Main(string[] args)
{
    Tester tester = new Tester();
    ((Prvni) tester ).Proved();    /*1*/ → Proved
    ((Druhe) tester ).Proved();    /*2*/ → Proved
    tester.Proved();              /*3*/ → Proved
    Prvni t1 = tester; t1.Proved();    /*4*/ → Proved
    Druhe t2 = tester; t2.Proved();    /*5*/ → Proved
}
```

Explicitní specifikace rozhraní 2/2

```
class Tester : Prvni, Druhe
{ public void Proved(){ Console.WriteLine("Proved"); }      /*D**Poznámka*/
// Z implementačních důvodů musí být explicitní deklarace private
  private void Prvni.Proved() { Console.WriteLine("Proved1"); }
  private void Druhe.Proved() { Console.WriteLine("Proved2"); }
}
```

se odezva změní na

```
[STAThread] static void Main(string[] args)
{
    Tester tester = new Tester();
    ((Prvni) tester).Proved();    /*1*/→ Proved1
    ((Druhe) tester).Proved();    /*2*/→ Proved2
    tester.Proved();             /*3*/→ Proved
    Prvni t1 = tester; t1.Proved();    /*4*/→ Proved1
    Druhe t2 = tester; t2.Proved();    /*5*/→ Proved2
}
```

Poznámka: Vynecháme-li deklaraci `Proved()` označenou `/*D**`, ohlásí se chyba v příkazu `tester.Vykonej(); /*3*/ ...Class1.Tester' does not contain a definition for 'Proved'`

Zvýšení bezpečnosti kódu operátory **is**

Deklarace metody **public int CompareTo(object obj)**

{ return GetMax()-((Bodl)obj).GetMax(); }

předpokládala, že argument obj odvozený od Bodl. Bezpečnější je ověřit si typ v run-time, třeba v Debug verzi.

#if DEBUG

public int CompareTo(object obj)

{ if(obj **is Bodl) return GetMax()-(Bodl) obj).GetMax()**

else throw(new System.Exception("Neni zděděno od Bodl.")); }

#else

public int CompareTo(object obj)

{ return GetMax()-(Bodl)obj).GetMax(); }

#endif

Jiné řešení – operátor *as*

Operátor *as* je ekvivalentní konstrukci

expression *as* type \equiv expression is type ?

(type)expression : (type)null;

např. **Bodl b = obj *as* Bodl;**

/ \equiv Bodl b= b is Bodl ? (Bodl) obj : (Bodl) null; */*

if(obj != null) return GetMax()-b.GetMax();

else throw(new System.Exception(

"Neni odvozeno od typu Bodl."));



5/3/2010

© K 13135, CVUT FEL Praha



No Break Yet !

*Do not leave,
we will play with
Ecq class*

Příklad: EkgData ze cvičení

```
public class EcgData // selected properties
{
    public int SelectedSignal - get, set
    public int CountOfSignals - get
    public int Length - get,
    public int this[int index] - get
    public int Max - get,
    public int Min - get
    protected void CreateMinMax(); // selected method
}
```

■ V příkladu demonstrujeme užití

- **dědění** – pro přidání nové funkčnosti ke stávajícímu kódu
- **virtuální metody** – pro modifikaci chování základní třídy pomocí dědění
- **abstraktní metody** – pro úsporu případných nevyužitých operací
- **interface** – nabídneme částečnou funkčnost a poté zas využijeme cizí funkčnost (foreach)

- Třída EcgData nabízí plnou funkčnost.
- Pokud uživateli některá její funkce chybí, může si ji přidat pomocí dědění.

```
class EcgDataEx : EcgData
{ public EcgDataEx(....) : base(...) { /* */ }
  public Bitmap GetBitmap()
  { Bitmap bmp = new Bitmap(640, 480);
    using (Graphics g = Graphics.FromImage(bmp))
    { Rectangle iecg = new Rectangle(0, Min, 10 * SampleRate, Max - Min);
      Rectangle obrazek = new Rectangle(new Point(0,0), bmp.Size);
      NastavTransformaci(g,iecg,obrazek);
      NakresliGraf(g, intervalEcg, obrazek);
    }
    return bmp;
  }
}
```

Použití nového objektu

```
class Program
{ static void Main(string[] args)
    { /*...*/
        EcgDataEx ecgex = new EcgDataEx(...);
        /*...*/
        try { Bitmap bitmap = ecgex.GetBitmap();
            bitmap.Save(@"C:\ADRVAN\ECG.jpg",
                System.Drawing.Imaging.ImageFormat.Jpeg);
        }
        catch (Exception ex) { }
    }
}
```

- Třída EcgData počítá minima a maxima signálů, ale použitý způsob nemusí vždy vyhovovat, například, pokud signál obsahuje rušivé špičky

```
public EcgData(...) // konstruktor EcgData
{
    CreateMinMax();
}
protected virtual void CreateMinMax()
{
    /*... virtuální metodu si může uživatel nahradit... */
}
```

//!!! atributy u min a max polí musíme změnit na **protected**

- Virtuální funkcí mohu modifikovat chování EkgData

```
class EkgDataV : EkgData
{
    public EkgDataV(...) : base(soubor) { }
    protected override void CreateMinMax()
    {
        /* uživatelův algoritmus */
    }
}

class Program {
    static void Main(string[] args)
    {
        EkgDataV ekgex = new EkgDataV(...);
    }
}
```

Abstraktní metoda

- Třída EcgData pořád obsahuje CreateMinMax, i když uživatel tenhle kód třeba vůbec nepoužije.
- Udělejme tedy CreateMinMax jako abstraktní metodu.

```
abstract class EcgData
```

```
{ protected int [ ] min;
```

```
protected int [ ] max;
```

```
/*...*/
```

```
protected abstract CreateMinMax();
```

```
protected EcgData(...) // konstruktor EcgData
```

```
{ /* další operace konstruktoru */
```

```
}
```

```
}
```

Do not define public constructors
in abstract types.

Constructors with public visibility are for
types that can be instantiated. Abstract
types can never be instantiated.

Nyní se musí EcgData vždy zdědit

```
class EcgDataA : EcgData
{ public EcgDataA(...) : base(...) { }
  protected CreateMinMax()
  { /* uživatelův algoritmus */
..
  }
}

class Program
{ static void Main(string[] args)
{
  EcgDataA ekgex = new EcgDataA(...);
  EcgData ekg = new EcgData("person1.ecg");
// objekt EkgData nelze teď konstruovat, jen použít pro dědění
}
}
```

- ...vylepšili jsme algoritmus pro min a max - *ted' není tak špatný, už obsahuje filtraci rušivých špiček* - mohl by se hodit i jinde.
- Nabídneme částečnou funkčnost EcgData pomocí interface IData

interface IData

```
{  
    int this[int index] { get; } // indexer  
    int Length { get; } // počet prvků jako int  
}  
  
/* ... */
```


Přidáme do EcgData interface

- EcgData odvodíme od IData - musíme přitom implementovat všechny prvky požadované interface, ale ty už máme...

```
class EcgData : IData
{
    private Int16[][] ecgSignal;
    private int selectedSignal = 0;
    public int Length
    {
        get { return ecgSignal[selectedSignal].Length; }
    }
    public int this[int index]
    {
        get { return index >= 0 && index < Length
                ? ecgSignal[selectedSignal][index] : 0; }
    }
    /* další prvky, vlastnosti a metody */
}
```

- Statická metoda FilteredMax nemá přístup na členy třídy a využívá jen prvky zahrnuté v objektu typu IData

```
static public int FilteredMax(IData idata, out min)
{
    min0 = int.MaxValue; int max0 = int.MinValue;
    /* Náš algoritmus */
    min=min0; return max0;
}

protected virtual CreateMinMax()
{
    /* pro každý signal voláme v cyklu for(...) */
    max[SelectedSignal]
        = FilteredMax(this, out min[SelectedSignal]);
}
```

Vytvoříme TestData třídu

```
class TestData : IData
{
    int [] pole;
    public TestData(int size)
    {
        pole = new int[size];
        for (int i = 0; i < pole.Length; i++)
            pole[i] = (int)(int.MaxValue / 2 * Math.Sin(i / 10.0));
    }
    public int this[int index]
    {
        get { return index >= 0 && index < pole.Length - 1
                ? pole[index] : 0;
        }
    }
    public int Length { get { return pole.Length; } }
}
```

- Pro využití FilteredMax() musí třída pouze implementovat prvky předepsané v IData rozhraní

```
class Program
{
    static void Main(string[] args)
    { TestData test = new TestData(10000);
      int min, max =
          EcgData.FilteredMax(test, out min);
    } // min = - int.MaxValue / 2,
      // max= int.MaxValue / 2
    }
```

Dědičnost

- + přidává další funkčnost k základním třídám
- + odvozená třída zdědí schopnosti základní třídy
- + chování zdědění prvků možno modifikovat
- 👉 pouze jednoduchá dědičnost
- ⇒ *rozšíření hotových tříd o nové možnosti*

Souhrn: Nástroje pro polymorfismus 2/3

Interface - *nelze vytvářet jejich instance, musí se zdědit*

- + několikanásobná dědičnost
- + různé třídy mohou mít stejnou interface
- + lze snadno dodat k již hotovým třídám
- 👉 neobsahují implementace, vše nutno napsat po zdědění.

⇒ *vhodné pro poskytnutí drobné funkčnosti - nabízíme ostatním: pokud si implementujete interface, můžete používat...*

Souhrn: Nástroje pro polymorfismus 3/3

Abstraktní třídy - *mají vlastnosti normálních tříd i interface.*

+ oproti interface mohou mít i částečnou funkčnost

👉 nedovolují násobnou dědičnost

⇒ *vhodné pro doplnění stejné funkčnosti skupině tříd, vytváření různých verzí téhož prvku*

abstract class VEC

{ *// změníme virtual na abstract*

public abstract string name();

public void AddName(string s) { s = s+**name()**+'.';}
};

class Circle : VEC

{ **public override** string name() {return "Circle";}
};

class Square : VEC

{ **public override** string name() {return "Square";}
};

Příště knihovny a služby operačního systému

