Vývoj aplikací v prostředí .NET

Katedra řídicí techniky,
ČVUT-FEL Praha

1. přednáška

VAN na Webu

- **Stránka předmětu:**

  dce.fel.cvut.cz

  -> E-kurzy systému Moodle

  -> Vývoj aplikací v **.NET**

  http://support.dce.felk.cvut.cz/e-kurzy/course/view.php?id=58

- V poloprovozu webový server:

  http://dcenet.felk.cvut.cz/

Doporučená literatura

Autor: **Vy, osobně**
Titul: **Váš vlastní program,** vydavatel: *Visual Studio 2008.*

Věřte nám, že tohle je nepřekonatelně nejlepší literatura, z níž se skutečně něco naučíte.

Pro počáteční představu o jazyce zkuste kurzy na WEBu – například:

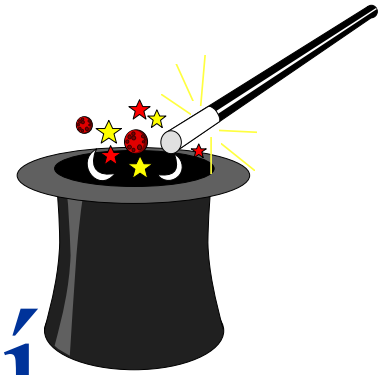
- http://www.functionx.com/csharp/ - velmi dobrý a přehledný kurz C#
- http://www.softsteel.co.uk/tutorials/cSharp/contents.html - 20 lekcí C#
- http://csharpcomputing.com/Tutorials/TOC.htm - *místy trochu zmatený text*

Milovníci papíru si mohou někde půjčit třeba:

- Robinson Simon et al.: **C# 2005 Programujeme profesionálně.** Computer Press 2007, 80-251-1181-4 / 9788025111819 (cca 1390 Kč)
  - *velmi podrobný popis, někdy až příliš,*
  - *naproti tomu jiné profesionální otázky jen lehce naznačené.*
  - *není zahrnuta verze 3.0.*

Úvodem něco o programování

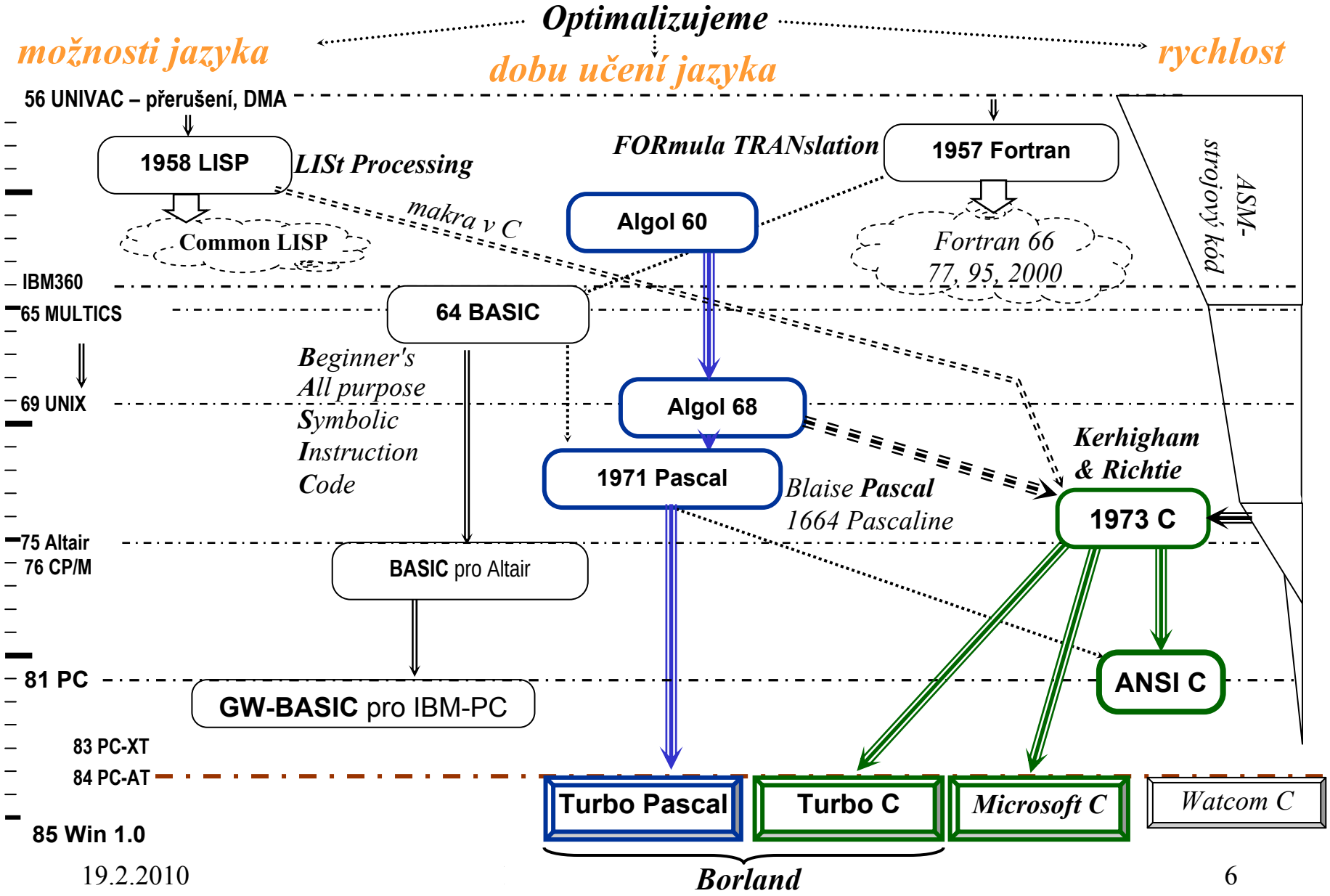
Programování



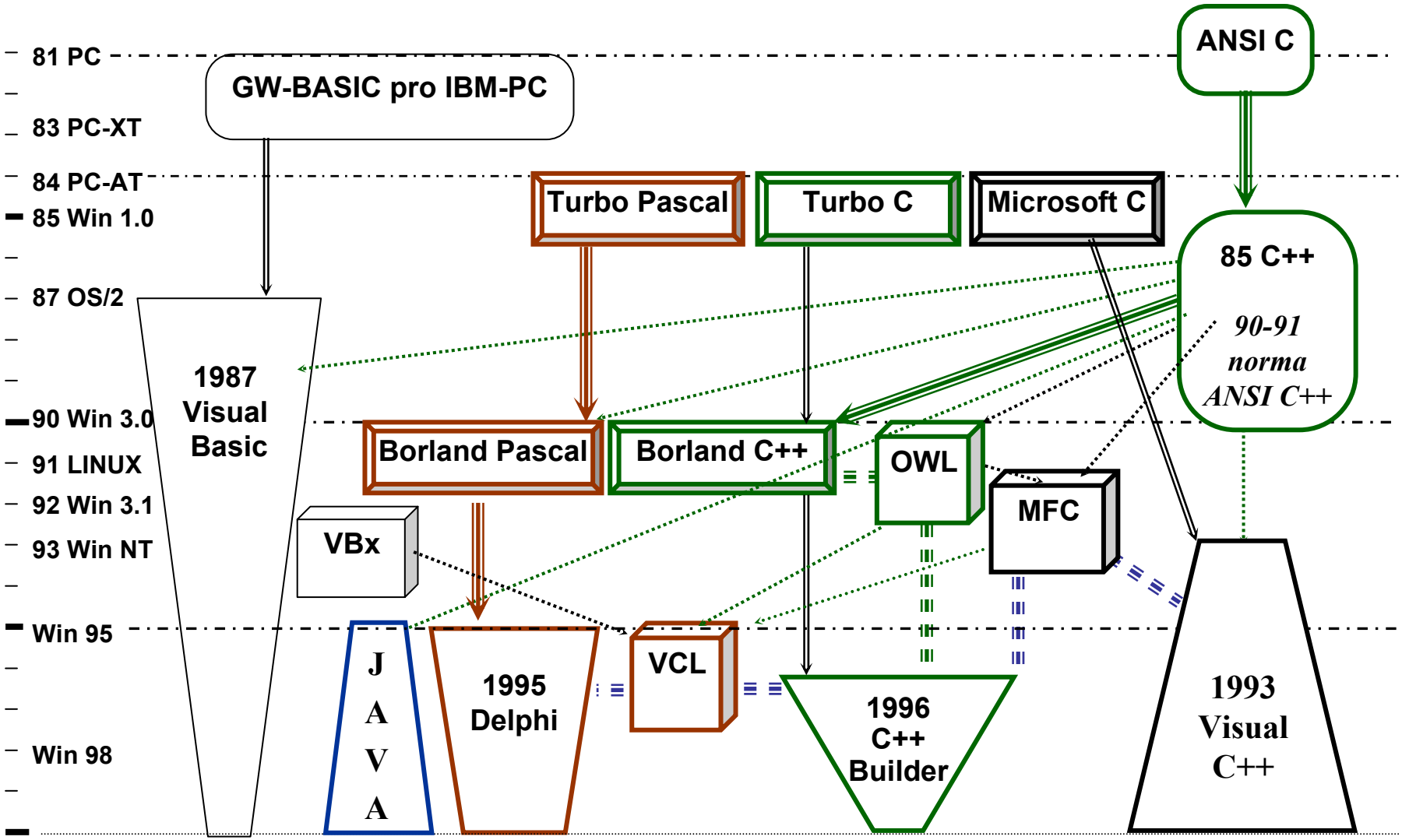
a n e b

co zde zůstalo po dávných OS…

Programování do roku 1985

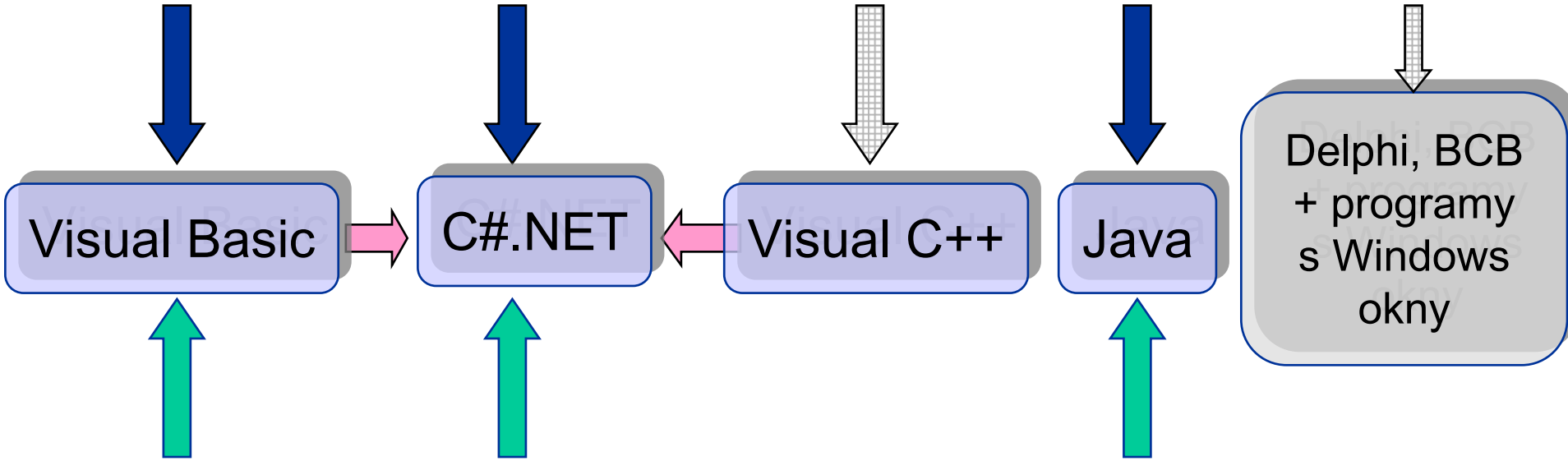


Programování na konci 2. tisíciletí



Začátek 3. tisíciletí

Programování Windows



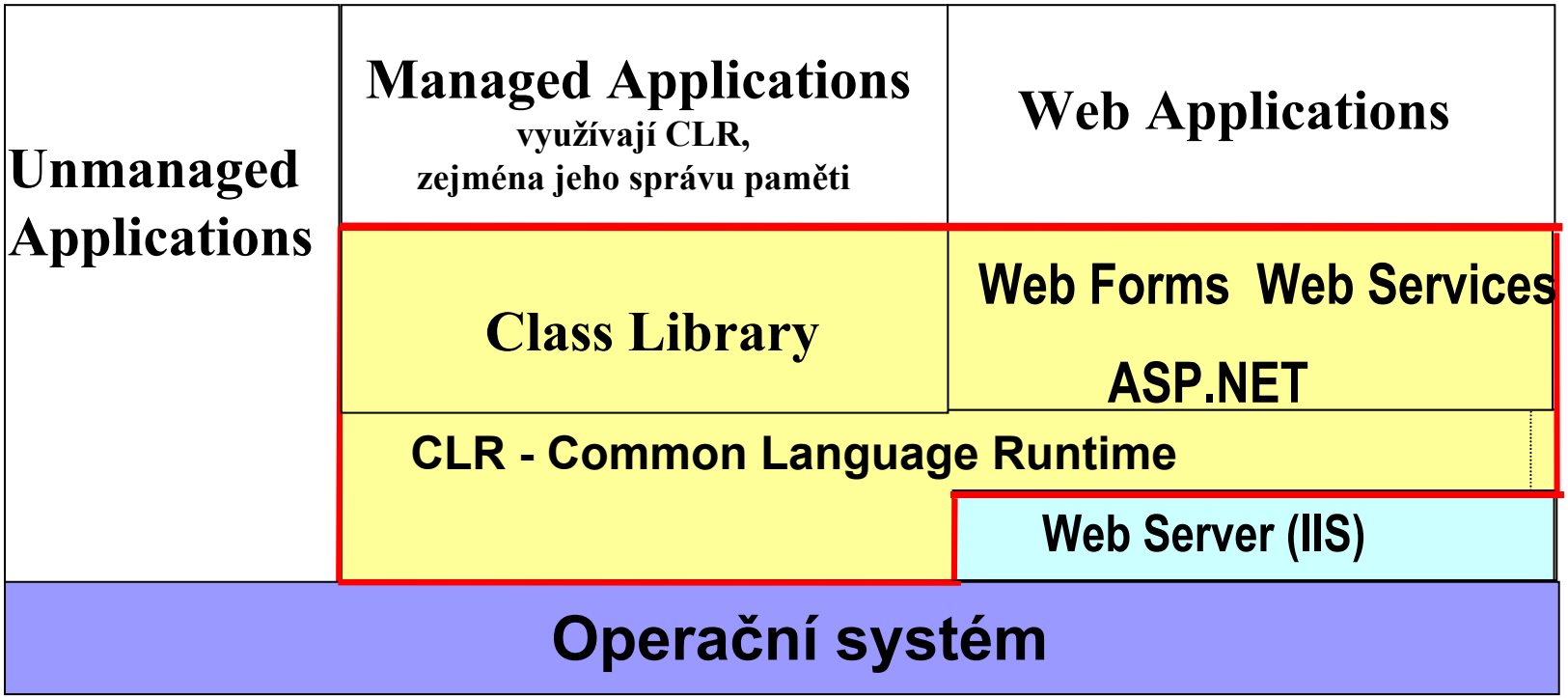
Programování webových stránek

… a také něco málo o .NET Frameworks

aneb na úvod trošku
nu[d|t]né teorie

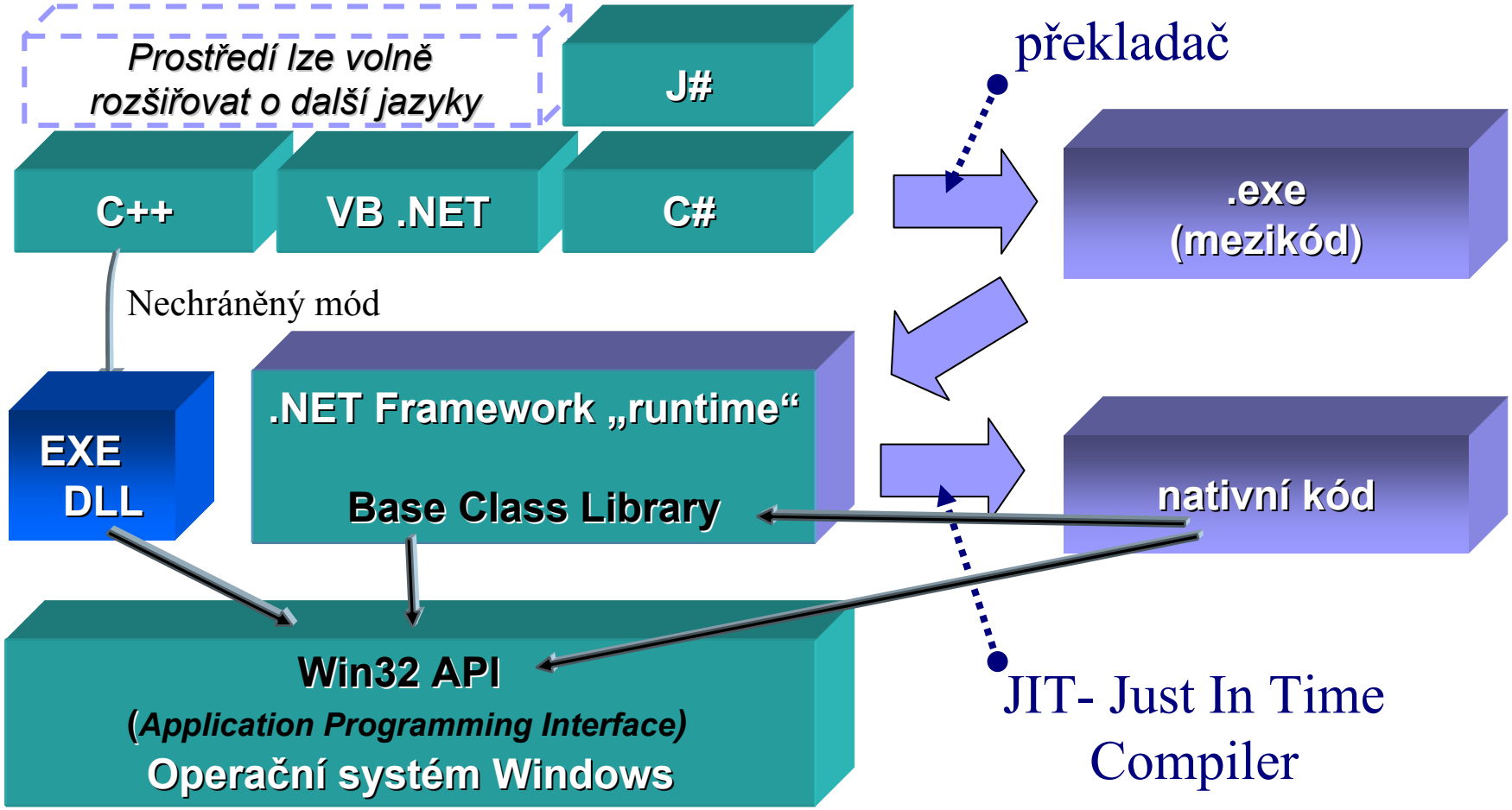
Co znamená NET.Framework?

- ***Nové prostředí pro aplikace**, považované za objektové API, dovolující lepší vzájemné sdílení kód a služeb.*



Co znamená NET.Framework?

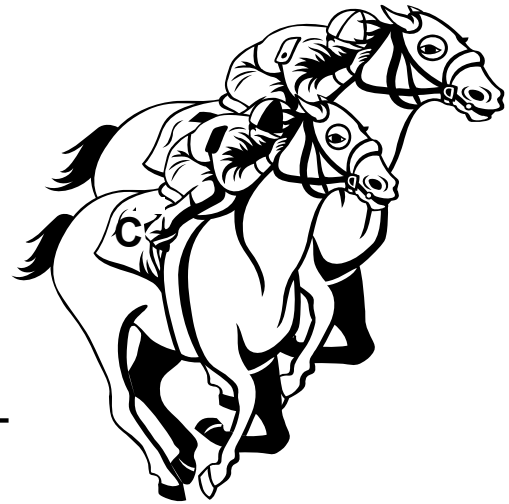
- Multi-jazykové vývojové prostředí pro Windows



Závod Java versus C#

1990 Java navržena jako univerzální jazyk
V domácích systémech neuspěla
Poté upravena pro kabelovou televizi
- další neúspěch

1994 se začíná rozvíjet WEB



1995 Java upravena pro WEB pro názvem Oak

1996 (23.ledna) vypuštění první JDK 1.0
Microsoft kupuje licenci Java

1997 Sun žaluje Microsoft za úspěšné Visual J++
Následuje série soudních procesů (1997-2002)
– Sun žaloval Microsoft za to, že používal modifikaci Java (JDK 1.1.4) vytvářející kód kompatibilní pouze s Windows, což porušilo licenční ujednání.
Microsoft musí z nařízení soudu stáhnout Visual J++ a také MS VM pro Java.

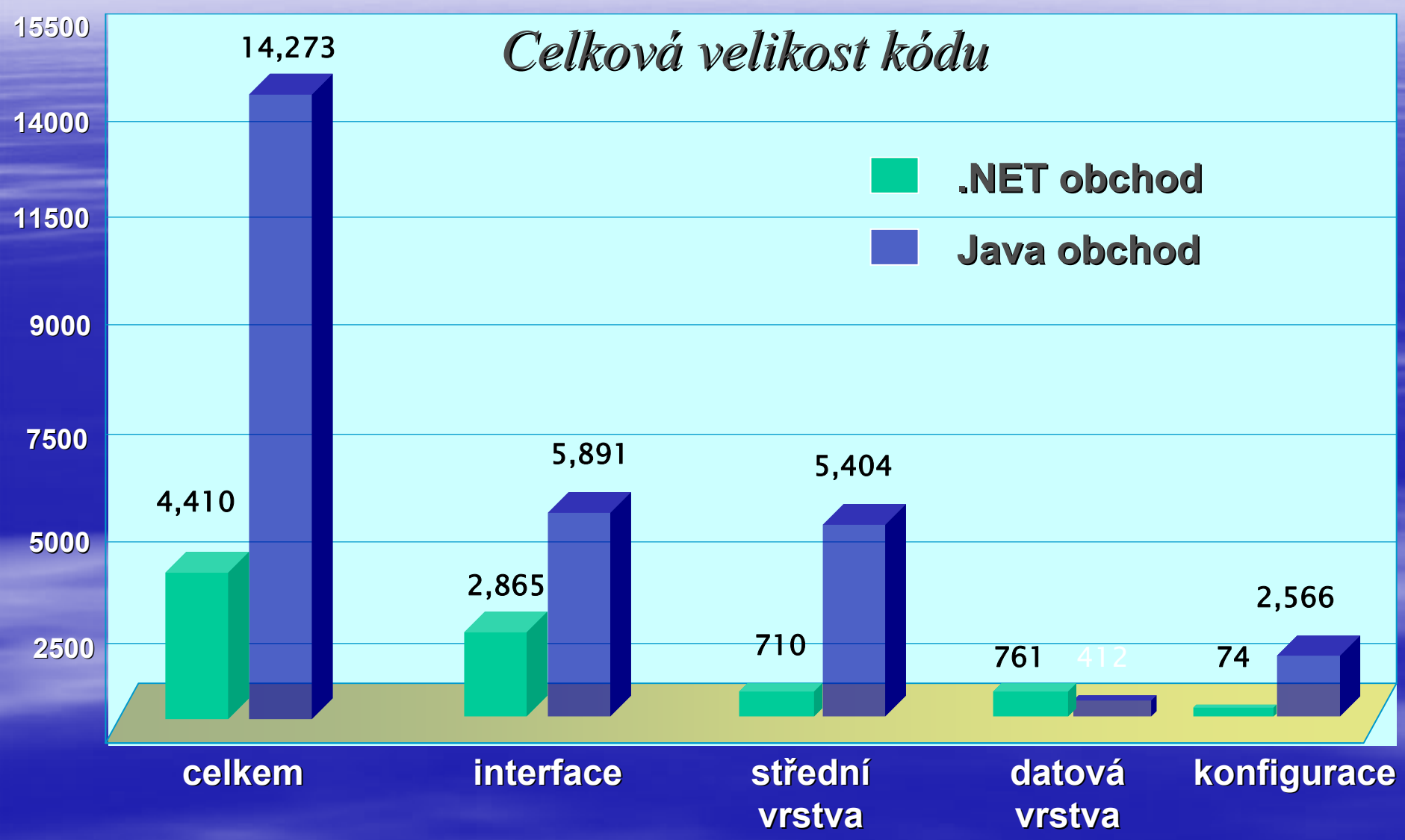
1999 Microsoft nahrazuje Visual J++ jazykem nově navrženým C#.NET, který obsahuje řadu vylepšení

2005 Microsoft uveřejnil „Conversion Assistant 3.0“ dovolující úplný převod Java kódu veze <5 do C#. Součást Visual Studio 2005
Sun převzal část idejí C# do poslední verze Java 5.0

2009 Oracle kupuje Sun

Hlavní výhoda -> WEB: Pet-shop

převzato od [Bent Thomsen, Dept. of Computer Science Aalborg University]



Vybrané odkazy Java vs. C#

- C# for Java Developers

  (http://msdn.microsoft.com/en-us/library/ms228358.aspx)
- Java vs. C# Code for Code Comparison

  (http://www.javacamp.org/javavscsharp/)
- Comparison of Java and C Sharp

  (http://en.wikipedia.org/wiki/Comparison_of_C_Sharp_and_Java)

Na síti existuje hodně srovnání, avšak pozor

- *často obsahují chyby – nutno ověřit*
- *starší popisy neplatí 100 % - oba jazyky se vyvíjejí...*

K čemu jsou vlastně objekty?

Co je vlastně objektem?

- **OOP (Objektove Orientované Programování) prohlašuje vše za objekt.**
- *Počítač je určitě objektem, ale co vítr z ventilátoru, barva, teplota?*
- **Jak popsat vztahy mezi reálnými objekty?**
  - Vztahy mezi objekty můžeme rozdělit na dvě hlavní skupiny - "mít" relace a "být" relace (*is-a relation, has-a relation*):
    - *Relace "**mít**" vyjadřuje vztahy "skládá se z", "obsahuje", apod. Objekt počítač **má** objekty disky, procesor, kryt, atd. Objekt kryt **má** zas objekty šroubky, plechy.*
    - *Relace "**být**" vyjadřuje obecné vztahy. Počítač **je** stroj.*

Objektové programování podporuje

- relaci "**být**" pomocí dědičnosti
  - od základního objektu "stroj" můžeme odvodit specializovaný stroj "počítač".
- relaci "**mít**" popíšeme přidáním vlastněných tříd jako členů jiných objektů.
  Jiná podpora neexistuje v OOP.
- *Skutečný počítač je však současně stroj, elektrické zařízení, zdroj rušení a mnoho dalšího. Jak tohle popíšeme?*

Jak tedy chápat objekty?

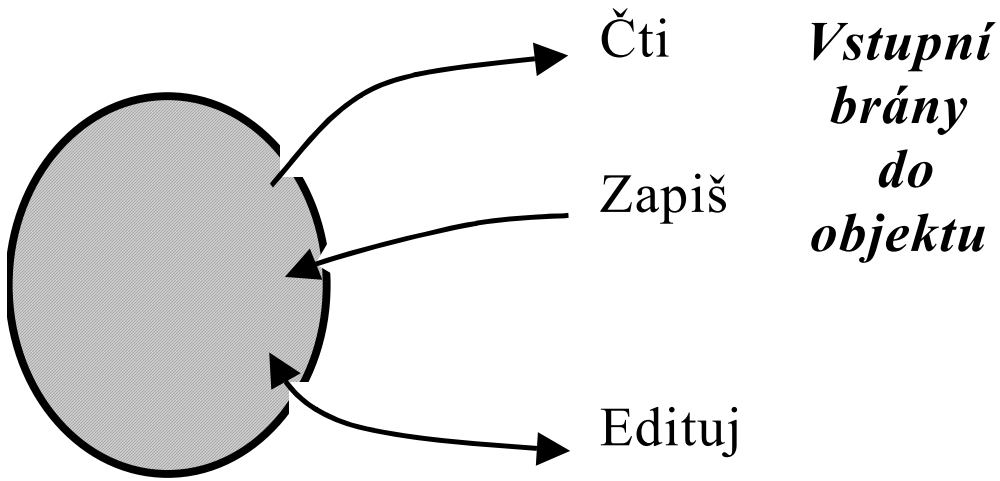
- spojují data s funkcemi, které s nimi pracují
- zapouzdřují data
- dovolují někdy popsat aproximovat jednodušší vztahy reálného světa

- ***Objekty řeší architekturu softwaru, a ne reality ← obtížně vyjadřují složitější vztahy!***

Objekty je šikovná programovací technika – když se umí používat!

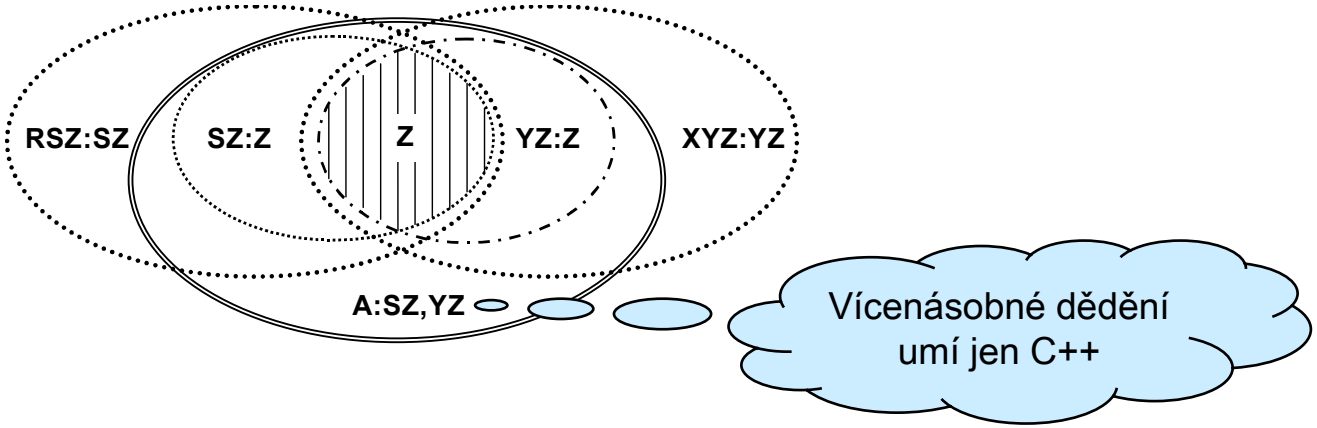
Objekt - zapouzdření dat (encapsulation)

- izolace nebezpečných operací
- vazba mezi daty a metodami, které s nimi manipulují
- public metody a data - vnější ovladače objektu
- private, protected metody a data – vnitřní operace

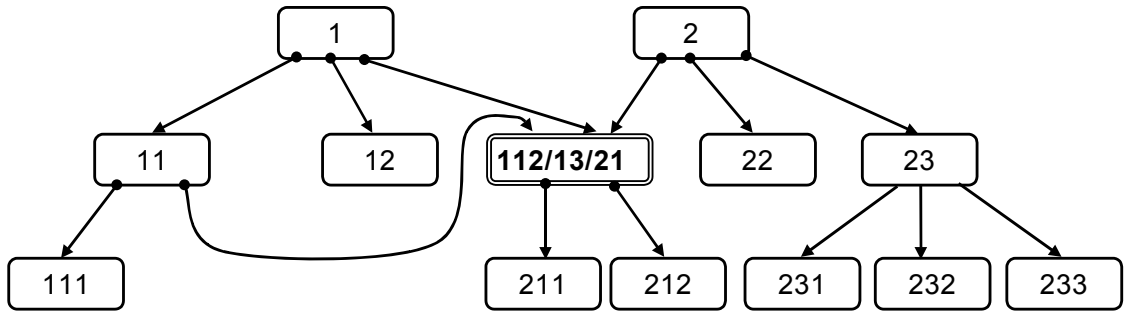


Práce s objektem

- Dědění (inheritance) - skládání operací objektu z podmnožin



- ***Objekt jako prvek objektu -*** vytvoření lesovité struktury nadřízených a podřízených prvků



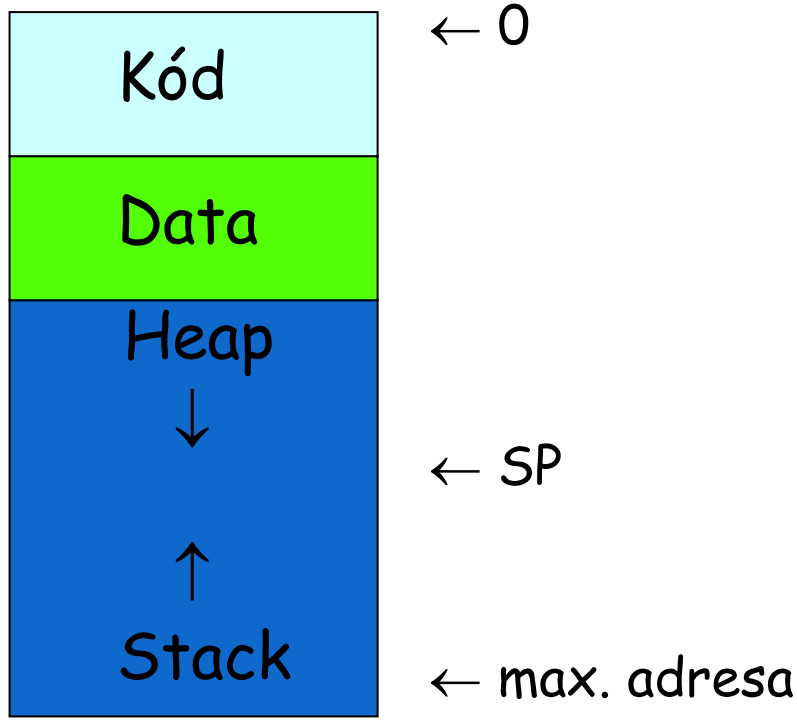
Objekty a jejich instance

Malé opakování

Stack versus heap

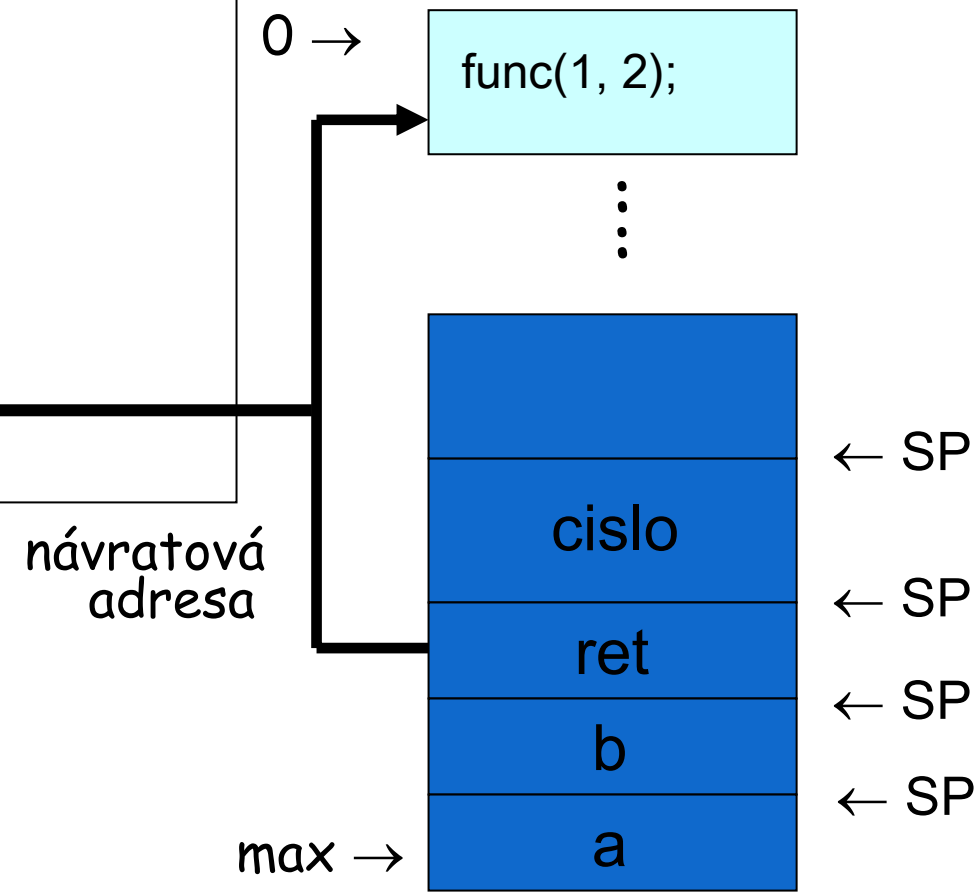
Organizace obecného programu

- **Kód** – kód programu
- **Data** – statické proměnné
- **Heap** – referenční typy
- **Stack**
  - Parametry funkcí
  - Návratové adresy
  - Hodnotové lokální proměnné



Animace volání funkce

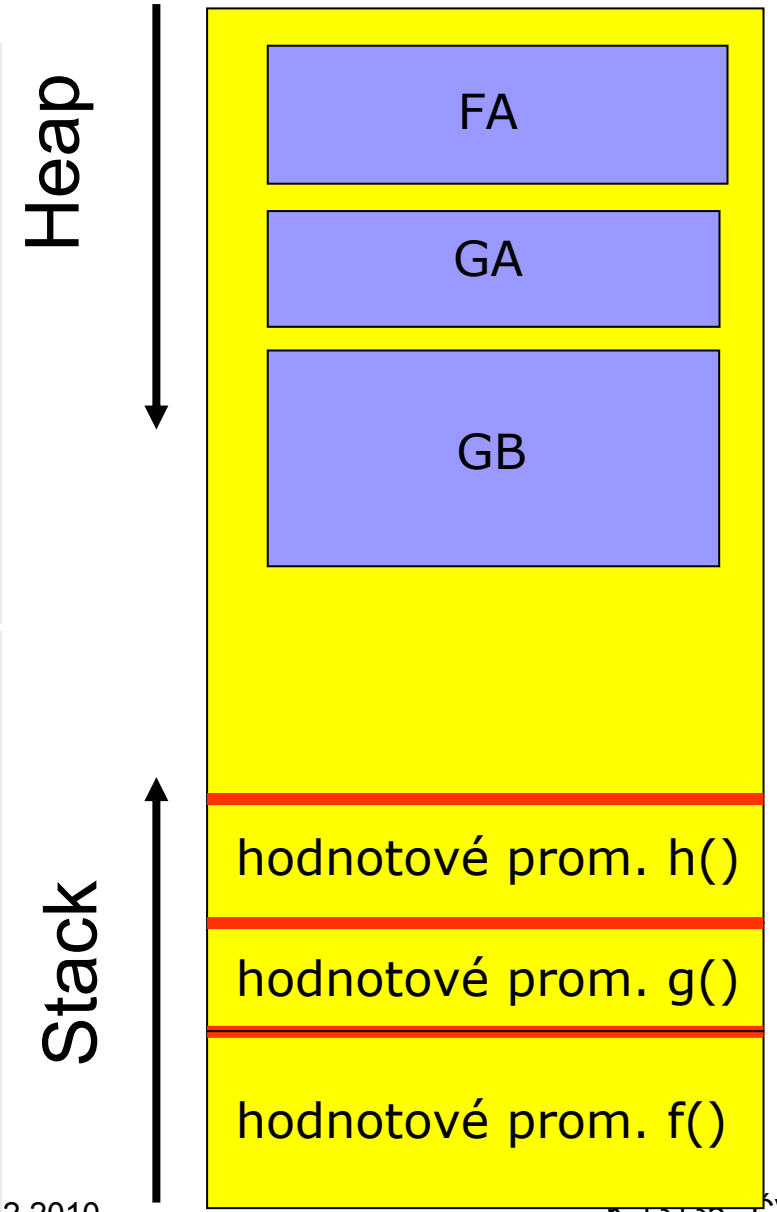
```
void static funkce(int a, int b)
{
    double cislo;
}
void  static Prog()
{
    func(1, 2);
}
```



Animace práce se stack-zásobník a heap-halda

Garbage Collector

Function Call & Return



- f() vytvoří ref. typ FA
- f() volá g()
- g() vytvoří ref. typ GA
- g() volá h()
- h() končí
- g() vytvoří ref. typ GB
- g() končí

Objektové prostředí neznamená automaticky objektový program

Neobjekty a objekty

Ryze neobjektový program

- *I v objektovém prostředí lze psát neobjektově*

```
static int GetMax(int x, int y) { return x>y ? x : y; }
static void Main(string[] args)
   {    int x1=1, y1=2; int m1 = GetMax(x1,y1);
        int x2=4, y2=5; int m2 = GetMax(x2,y2);
   }
```

V tomto programu chybí vazba dat do celků
→ nepřehlednost

Data v objektech

```
struct Bod  { public int x, y;          }
static int GetMax(Bod b)
{   return b.x>b.y ? b.x : b.y; }
static void Main(string[] args)
{       Bod b1; b1.x=1; b1.y=2; int m1 = GetMax(b1);
        Bod b2; b2.x=4; b2.y=5; int m2 = GetMax(b2);
}
```

Program zachází dobře se skupinami dat, ale za cenu složitějšího kódu funkcí
→ nutnost odkazovat v nich na prvky

Konečně objektový program

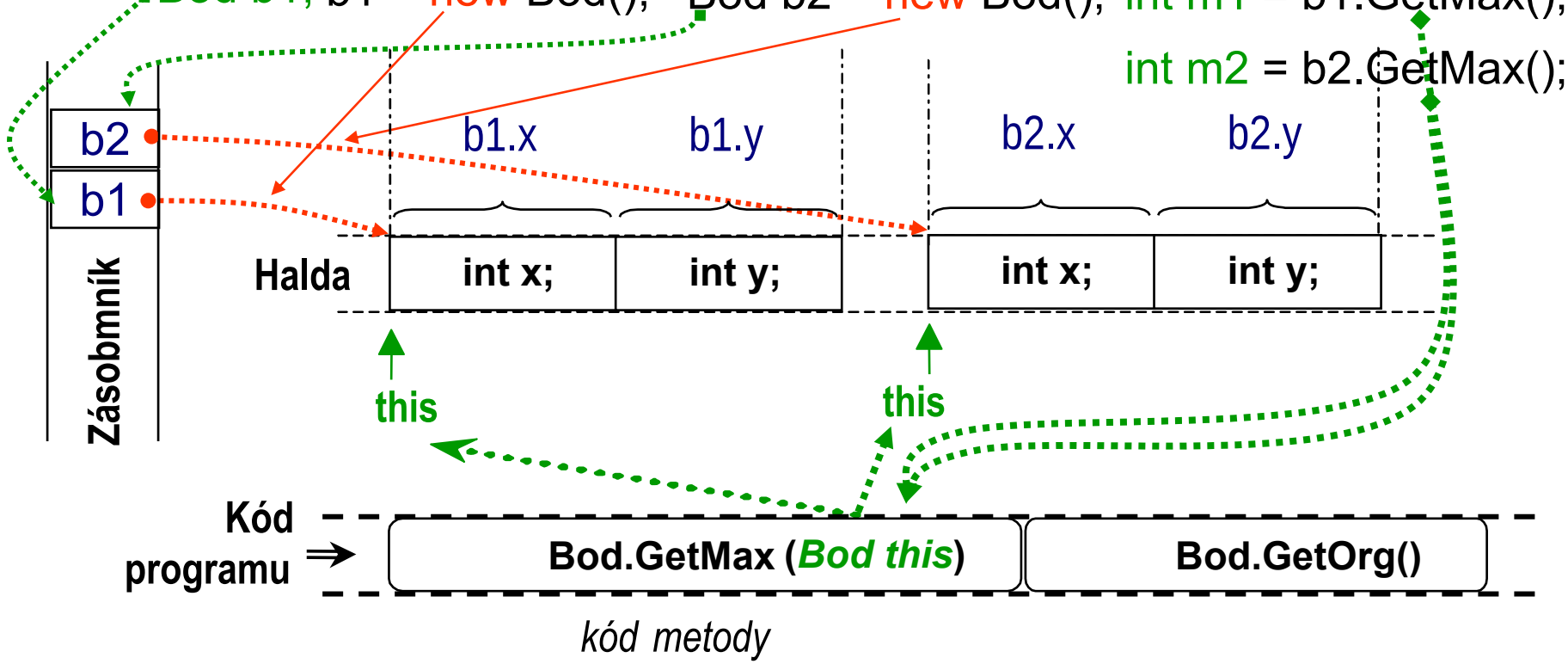
```
class Bod
{   public int x, y;  // vždy začínáme od dat ← odtud OOP
    public int GetMax() { return x>y ? x : y; }
    static public int GetOrg  { return 0; }
}
```

```
//  metodu Bod.GetMax() lze zapsat i jako:
public int GetMax() { return this.x>this.y ? this.x : this.y; }
// ve static public int GetOrg – nelze odkazovat na x, y
```

```
//  vnitřně je Bod.GetMax() přeloženo
public int GetMax(Bod this)          // automaticky vloženo
      { return this.x>this.y ? this.x : this.y; }
// do static public int GetOrg – překladač nepřidá this
```

Uložení jednotlivých instancí objektů

```
class Bod {  public int x, y;
             public int GetMax() { return x > y ? x : y; }
             static int GetOrg() { return 0; }                }
static void Main(string[] args)
 {  Bod b1; b1 = new Bod();  Bod b2 = new Bod();  int m1 = b1.GetMax();
                                                  int m2 = b2.GetMax();
```



Klíčové slovo *this*

- *okazuje na instanci dat objektu, ze které byla příslušná metoda vyvolaná;*
- *v kódu metody se obvykle nepoužívá - překladač ho doplňuje automaticky;*
- *implicitně se uvádí jen výjimečně, když*
  - *chceme zdůraznit, že pracujeme s elementy objektu*
  - *nebo pro vyloučení záměny s jiným parametrem*

```
public void InitX(int x) { this.x=x; }
```

- *statické metody nemají **this**. Nelze ho v nich používat, ale díky tomu je můžeme volat bez instance objektu.*

C# class a struct

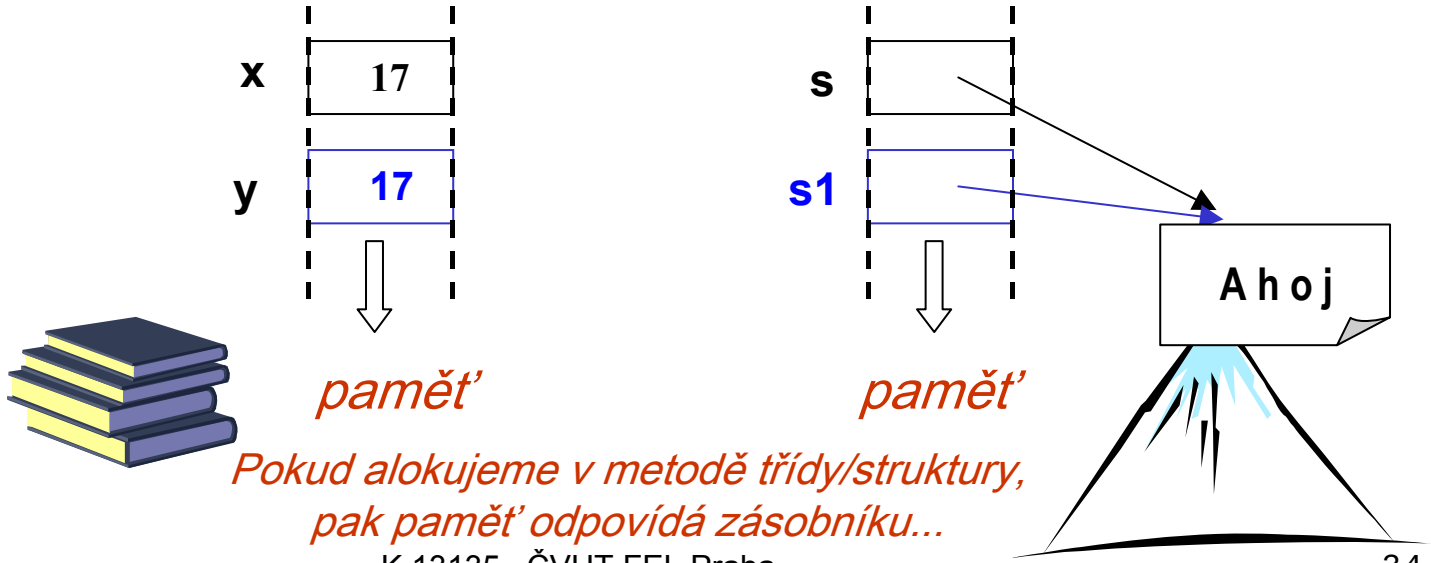
Třída a struktura

```
class Bod
 {   public int x, y;
    public int GetMax()
      { return x > y ? x : y; }
    static int GetOrg()
      { return 0; }
 }
static void Main(string[] args)
{   Bod b1 = new Bod(); // new nutné
    b1.x = 1; b1.y = 2;
    int m1 = b1.GetMax();
    Bod b2 = new Bod();
    b2.x = 1; b2.y = 2;
    int m2 = b2.GetMax();
}
```

```
struct BodS
 {  public int x, y;
    public int GetMax()
      { return x > y ? x : y; }
    static int GetOrg()
      { return 0; }
 }
static void Main(string[] args)
 {  BodS bs1; // !!! možno i s new
    bs1.x = 1; bs1.y = 2;
    int ms1 = bs1.GetMax();
    BodS bs2; // !!! možno i s new
    bs2.x = 1; bs2.y = 2;
    int ms2 = bs2.GetMax();
 }
```

Referenční a hodnotové typy

	Hodnotové typy (Value Types)	**Referenční typy** (Reference Types)
proměnná	hodnota	reference (odkaz) na hodnotu
uložena jako	data	odkaz na haldu (heap)
inicializace	0, false, '\0'	null
přiřazení =	kopie hodnoty	kopie reference
příklad	int x = 17; int y = x;	string s = "Ahoj"; string s1 = s;



Pokud alokujeme v metodě třídy/struktury, pak paměť odpovídá zásobníku...

Uložení class a struct v paměti 1/2

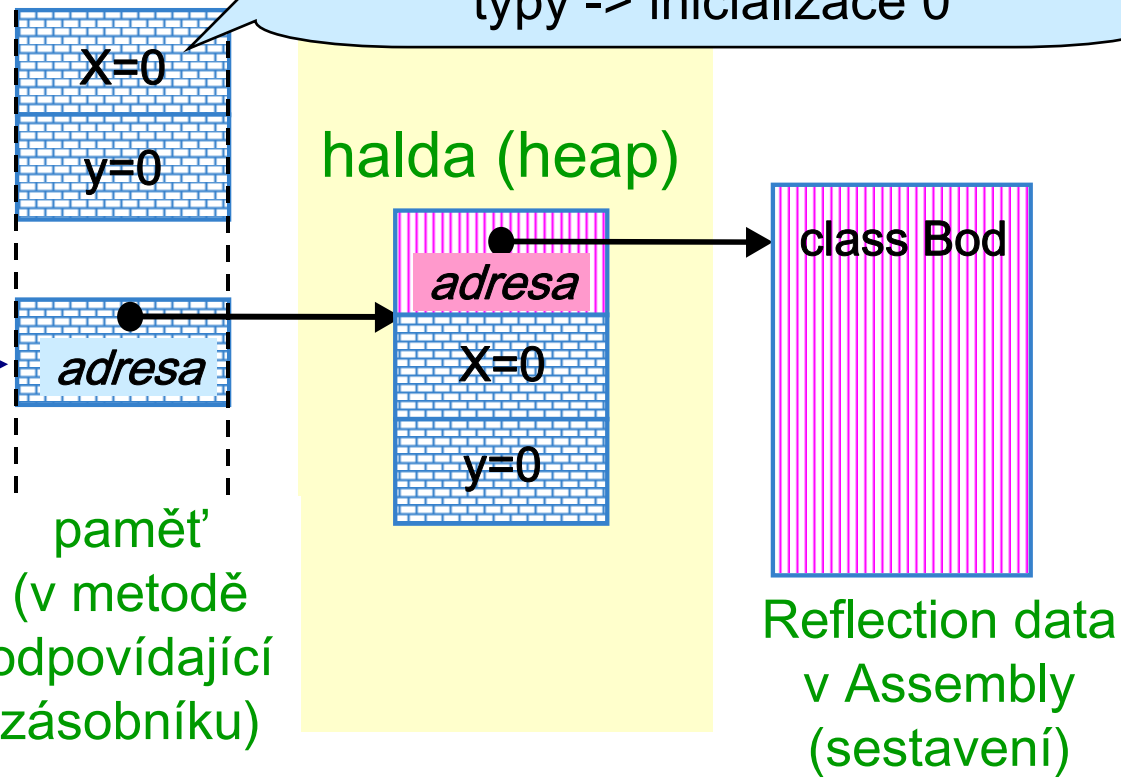
// struct

BodS bs;

// class

Bod bc;

bc = new Bod();

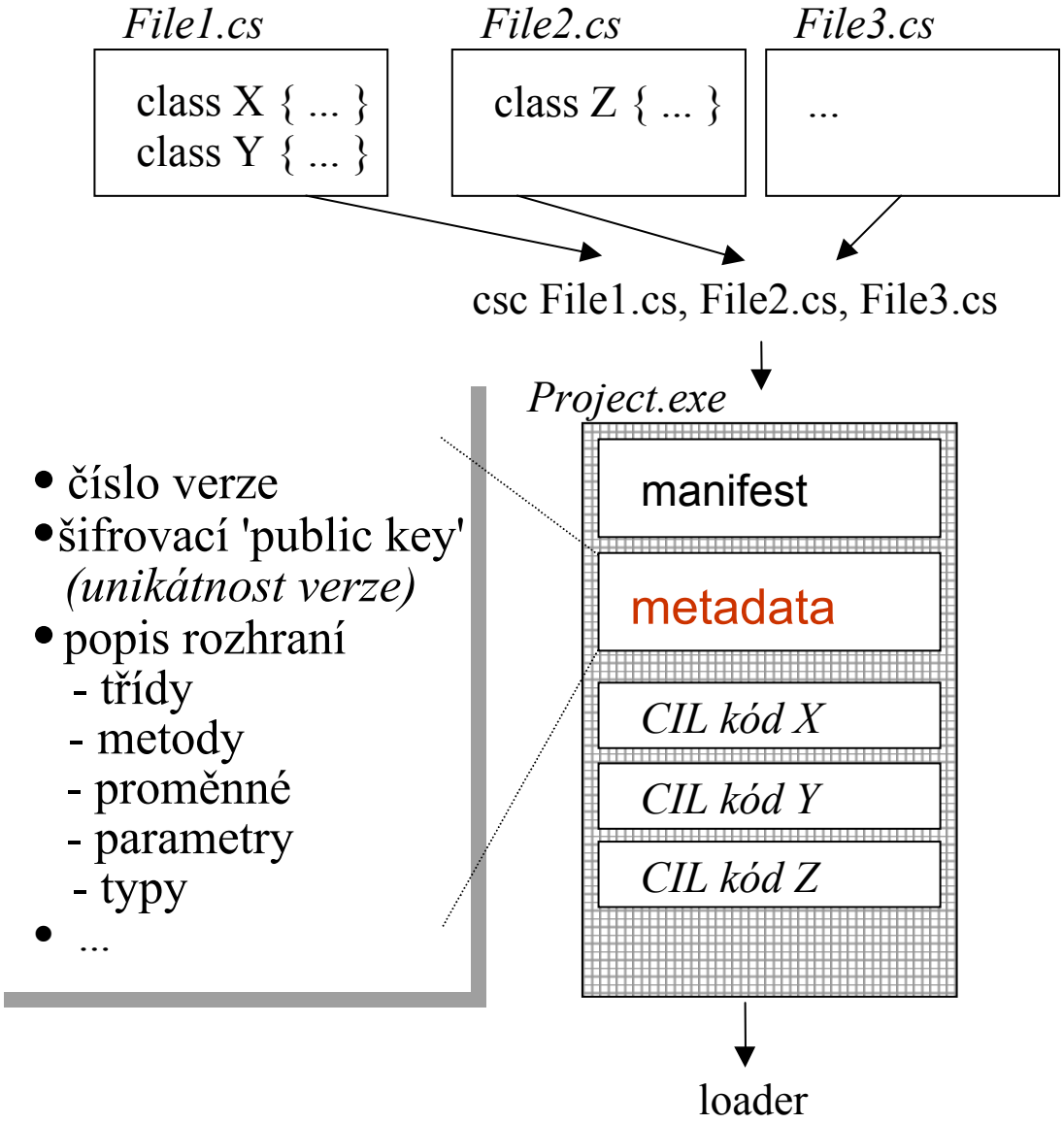


BodS bs; *je totožné* s BodS bs=new BodS();

Bod bc; *je různé od* Bod bc = new Bod();

bezparametrové konstruktory

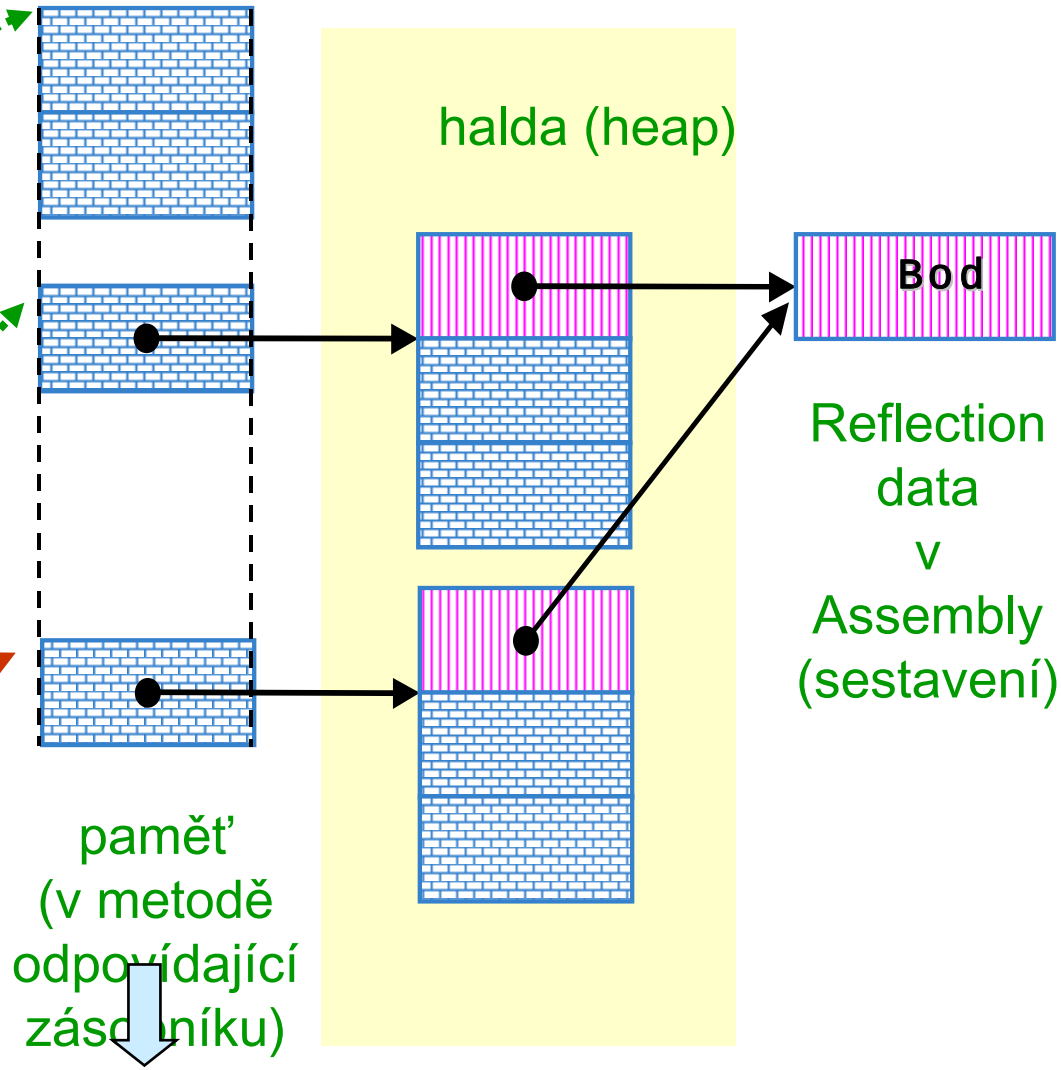
Assembly


File1.cs
class X { ... }
class Y { ... }
File2.cs
class Z { ... }
File3.cs
...
csc File1.cs, File2.cs, File3.cs
Project.exe
manifest
metadata
CIL kód X
CIL kód Y
CIL kód Z
loader
• číslo verze
• šifrovací 'public key' (unikátnost verze)
• popis rozhraní
- třídy
- metody
- proměnné
- parametry
- typy
• ...

Uložení class a struct v paměti 2/2

```
class Bod { int
    x, y; ... }
struct BodS {
    int x, y; ... }
```

- **BodS bs;**
- **Bod bc = new Bod();**
- **Bod bc2 = new Bod();**



Referenční / hodnotový typ

na srovnání class a struct

Výhody C# struct

- *se hodí pro malé datové prvky jako body, např. komplexní čísla. Pole 1000 bodů*

  BodS[ ] pole = new BodS[1000];

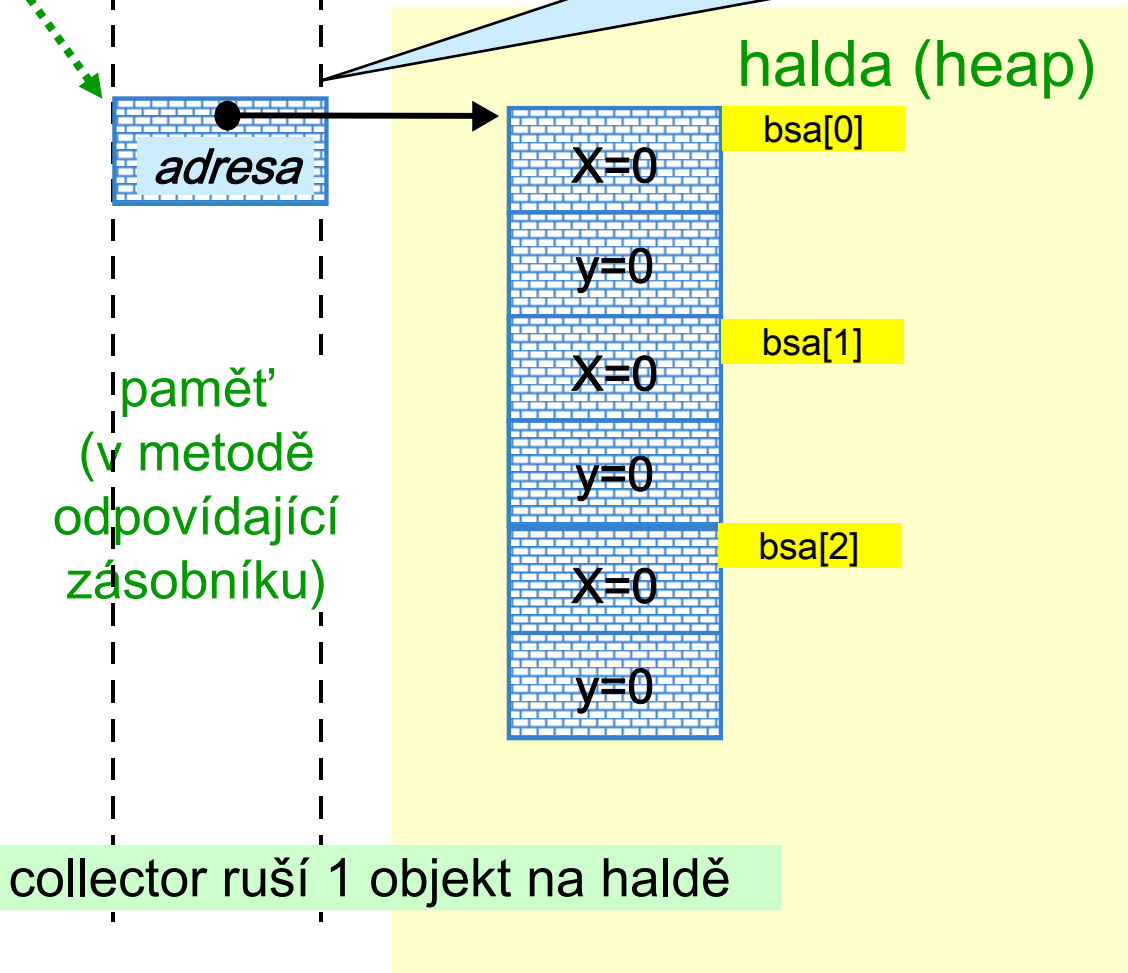
  vyjde efektivněji se struct BodS než s class Bod
- *totéž platí i pro členy třídy. Struct se vloží přímo mezi datové prvky instance. Nevytváří žádné další alokace na haldě.*
- *Ruší se automaticky se svým nadřízeným objektem, tedy třídou, strukturou nebo metodou → nezatěžuje správu haldu.*

Pole se struct v paměti

// struct

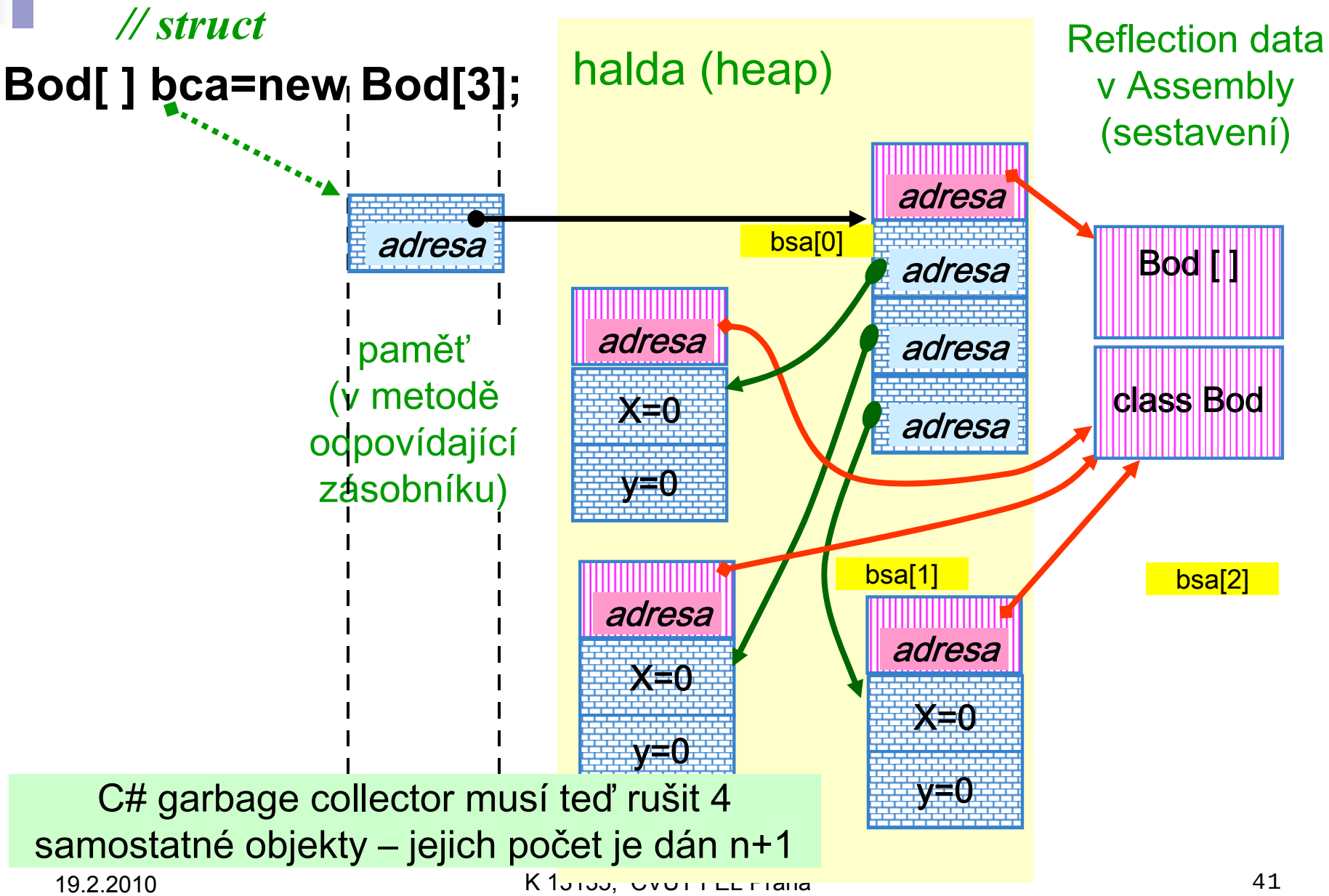
BodS [ ] bsa=new BodS[3];

v C# je pole referenční typ Array



C# garbage collector ruší 1 objekt na haldě

Pole s class v paměti



C# garbage collector musí teď rušit 4 samostatné objekty – jejich počet je dán n+1

Class/struct v jiných jazycích

- ***Java*** *Probraná deklarace* class Bod *se shoduje s C#, ale v Java chybí typ* struct.
- ***C++*** struct a class *se liší jedině výchozím přístupovým atributem -* public *pro C++* struct *a* private *pro C++* class

```
class Bod      // deklarace objektu
 { public:   // přístupový atribut pro následující metody
      int x,y;
      int GetMax() { return x>y ? x : y;  } // inline metoda
      int GetOrg();
};  // V C++ je nutný středník za deklarací třídy
int Bod::GetOrg()  { return 0;  }     // definice metody vně
```

Deklarační prostory

základ pro užívání identifikátorů

Význam deklaračních prostorů

- určují platnost proměnných, rozsah jejich užití (tzv. scope)
- omezují volbu jmen dalších proměnných
- určují, zda se identifikátor vztahuje k členu metody a k jakému
- určují přístup pomocí atributů public, protected, private
- umožní nám pochopit přetypování objektů

Organizace .NET kódu

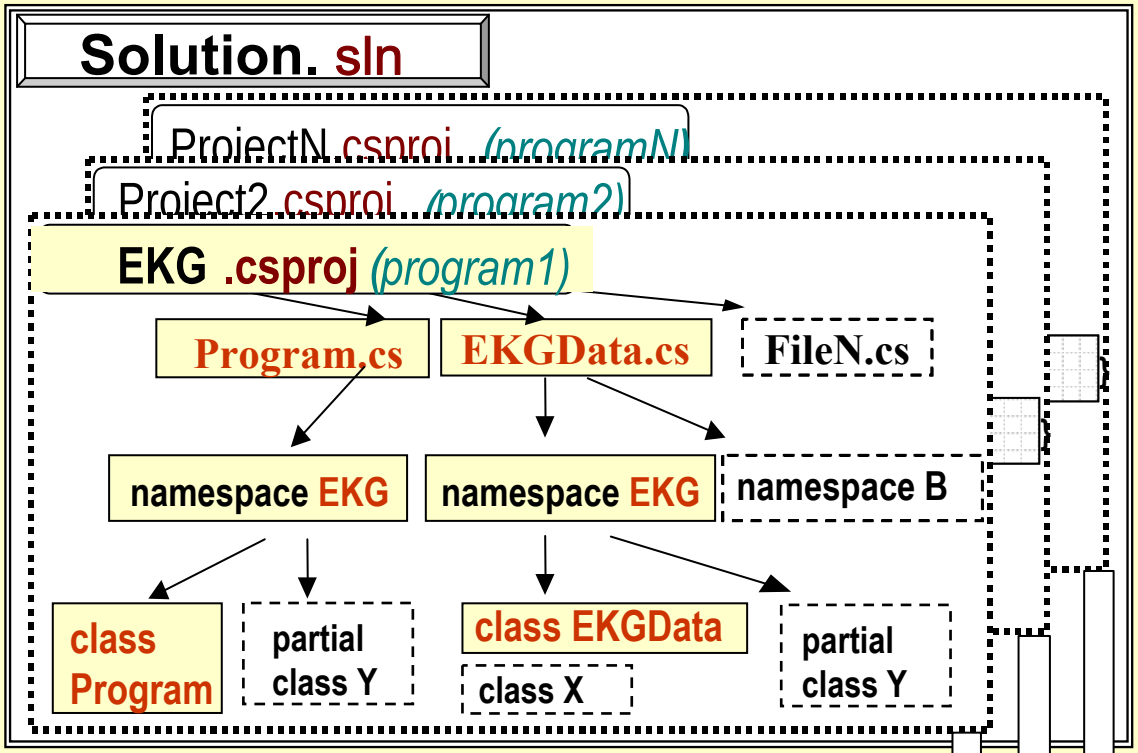
```
namespace Experiments
{ class Program
  { static void Main(string[] args)
    {
    }
  }
  namespace First
  {
    namespace Second
    { class Exp1
      { public static void A() { }
        public static void B() { }
      }
    }
    class Exp2
    { public static void A() { }
      public static void B() { }
    }
  }
}
```

```
using Experiments.First;
using Experiments.First.Second;
```

```
Experiments.First.Second.Exp1.A();
First.Second.Exp1.A();
Experiments.First. Exp2.A();
First.Exp2.A();
```

```
Exp1.A();
Experiments.First.Second.Exp1.A();
Exp2.A();
Experiments.First.Exp2.A();
```

Obecná struktura C# programu



příklady možných cílů překladu

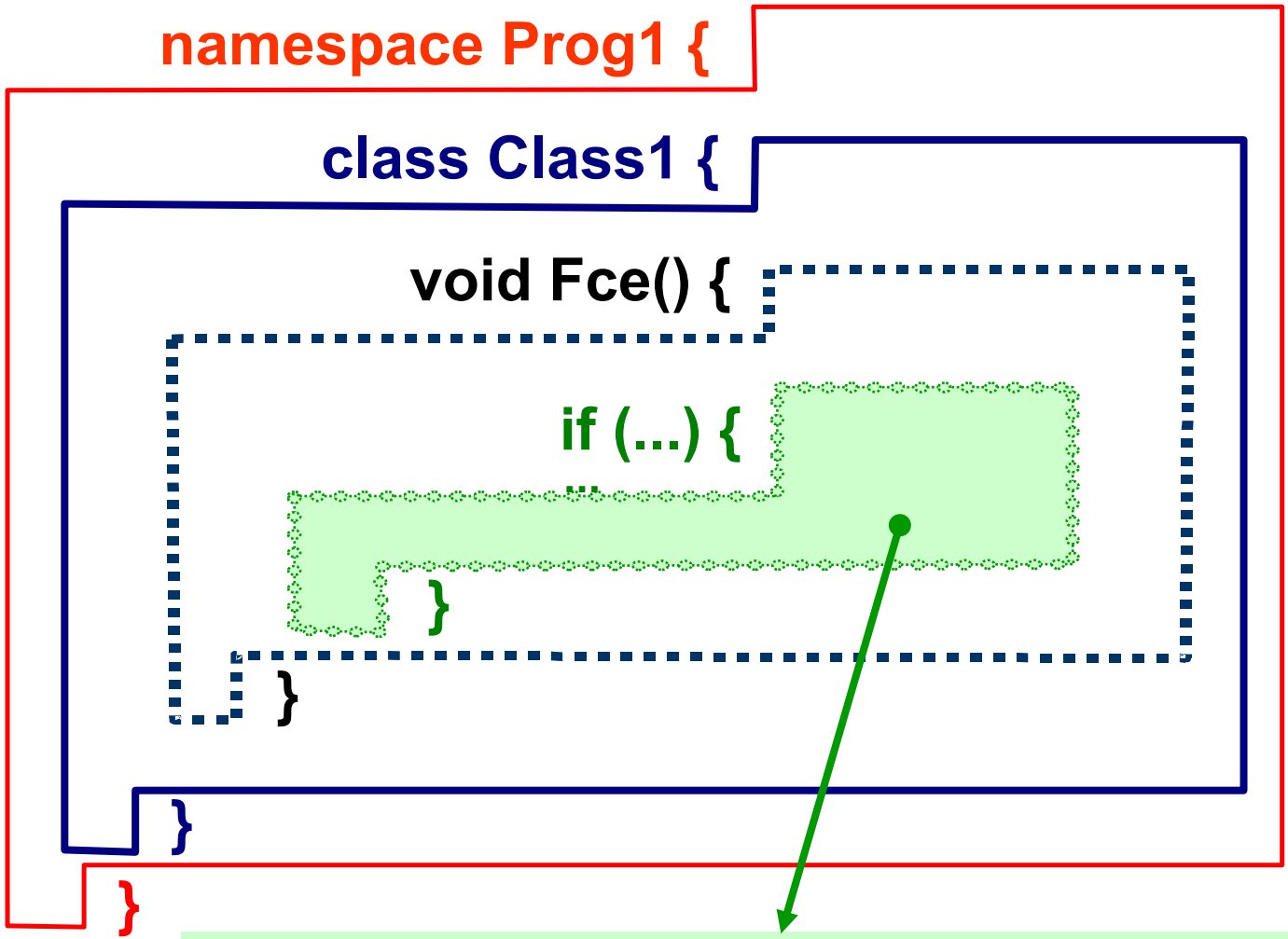
<dir>\ **EKG** \bin\Debug**Project1.exe**
<dir>**Project2**\bin\Release**Project2.exe**
<dir>**ProjectN**\bin\Debug**ProjectN.dll**

- *Jeden C# soubor může obsahovat i **více deklarací tříd**, na rozdíl od Java programu.*
- *Deklarace jedné třídy nebo struktury může být rozložena **do více souborů.**, tzv. "partial classes"*

Deklarační prostory 1/2

- *Deklarace definuje jméno uvnitř deklaračního prostoru, do kterého patří.*
  - **Namespace**: zde deklarujeme třídy (class), rozhraní (interface), struct, enum, delegate
  - **Class, struct, interface** zde deklarujeme pole (field), metody (method) patřící třídám, strukturám.
  - **Blok** deklarace lokálních proměnných
  - **Vnořený blok**: deklarace lokálních proměnných bloku

Deklarační prostory 2/2

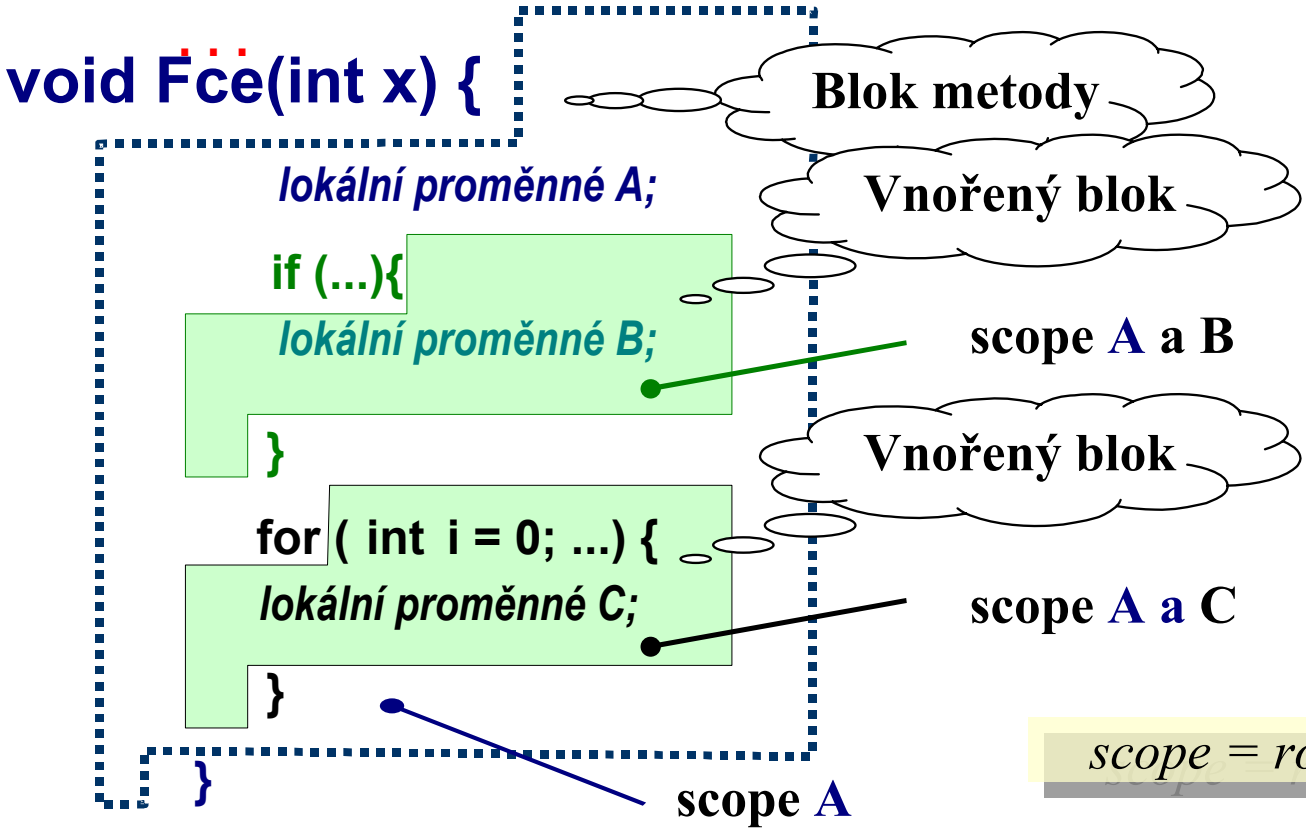


Vnořený příkazový blok není samostatný, patří do deklaračního prostoru metody!

Pravidla pro deklarační prostory

1. Žádné jméno není viditelné mimo svůj deklarační blok.
2. **Každé jméno je viditelné v celém deklaračním prostoru od {**
   - **Výjimku z tohoto pravidla tvoří jen lokální proměnné**
     - ty lze používat jen od deklarace do konce dekl. prostoru.
3. Žádné jméno nesmí být deklarováno vícekrát uvnitř téhož deklaračního prostoru stejné úrovně, s výjimkou přetížených (*overloaded*) členů tříd.
4. Jméno může být však předeklarování ve vnořeném deklaračním prostoru, avšak s výjimkou vnořených příkazových bloků.
5. Znovu deklarované jméno překryje původní proměnnou, změní její stav z visible na hidden

Vnořené příkazové bloky



scope = rozsah platnosti

- Ve vnořeném příkazovém bloku lze deklarovat jen lokální proměnné, které neskrývají identifikátory nadřazeného bloku

Pozn. C++ dovoluje předeklarovat identifikátory i v příkazovém bloku, ale C# tohle zakazuje kvůli usnadnění verifikaci programu.

Příklad

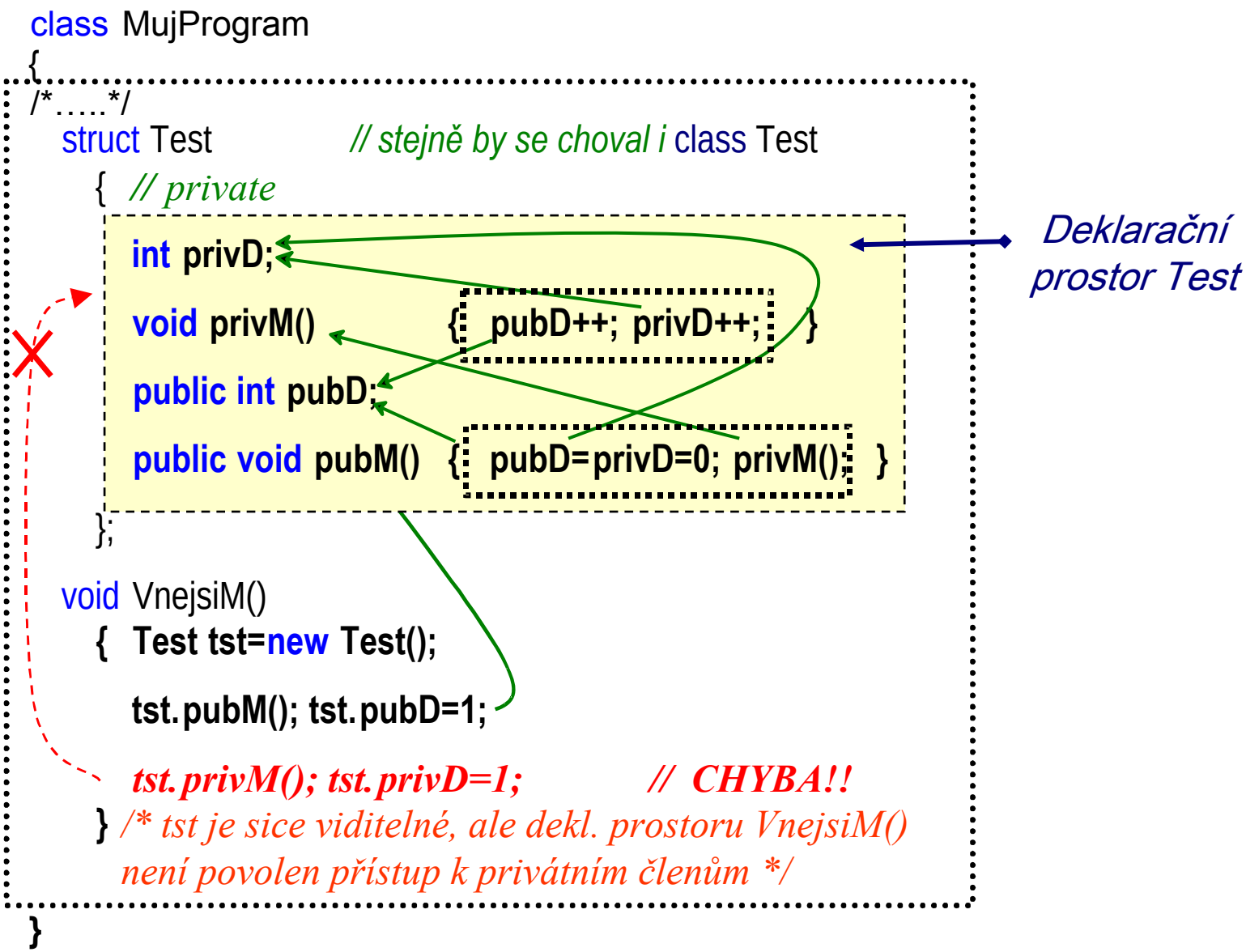
Překrývání identifikátorů proměnných se obecně pokládá za nepřehledný programátorský styl – zde užito jen pro demonstraci vlastností dekl. prostorů

```
void fce(int a) {
   int b;
   if (...) {
         int b;   // chyba: b je již deklarované ve vnějším bloku
         int c;
         int d;
         ...
               } else {
         int a;   // chyba: a je již deklarované ve vnějším bloku
         int d;   // OK: d je v jiném bloku
            }
   for (int x = 0; ...) {...}
   for (int x = 0; ...) {...}        // OK: x jiný lokální blok
   int b;           // chyba: b je již deklarované
}
```

Modifikátory přístupu- *access modifiers*

private	*nejvíce omezený přístup, povoleno manipulovat pouze z téhož deklaračního prostoru (třídy, struktury, interface)*
public	*není stanoveno žádné omezení*
internal	*limituje přístup na soubory téže assembly, kde je jako public*
protected	*limituje přístup na deklarační prostory této a odvozených tříd*
internal protected	*limituje přístup na soubory odvozené třídy z téže assembly*

Příklad užití atributů



Srovnání C#, C++ a Java

C#	C++	Java
private *default*	private *default class*	*private*
public	public *default struct*	*public*
protected	protected	*N/A*
internal		*protected default*
internal protected		*N/A*

Poznámka k jménům proměnných

aneb nejen objekty tvoří dobrý program

Identifikátory

- **Identifikátor = (*písmeno* | '_' | '@') {*písmeno* | *číslice* | '_'}**
- **Unicode!**
  - Jména mohou obsahovat "Unicode escape sequences" \u017E pro ž someName, sum_of3, _10procent, @while, žlutý, \u017Elut\u00FD
- Rozlišují se malá a velká písmena
  - AB, ab, aB, Ab - jsou 4 různá jména
- "@" přinutí překladač brát klíčové slovo (keyword) jako identifikátor - **if** je klíčové slovo **@if** je identikátor ***if***
  - Řetězec "\"D:\\readme.txt\"" rozlišuje 'escape' znaky. Je-li uvozený @, jak \ není 'escape' znak. Všechny vnitřní " se zdvojují a řetězec může mít i několik řádek.

```
@"Jméno souboru:
""D:\readme.txt"""
```

se vypíše jako

Jméno souboru:
"D:\readme.txt"

C# konvence pro identifikátory

- Pascal konvence – *každé první písmeno složeného jména je velké, např.* IndexOf, GetValue()
- Camel konvence – *první slovo má malé písmeno, následující slova pokračují velkým písmenem, např.* boldFont, shortName
- Vše velkým písmem – *používáme pro 2 písmenné akronymy a symboly podmíněného překladu, např. LF, DEBUG, ale Xml*

- public datové členy – podstatné jméno v Pascal konvenci
- public metody – sloveso v Pascal konvenci
- property – přídavné jméno v Pascal konvenci
- parametry – používají Camel konvenci
- akronymy delší než 2 písmena mají ve jménech pouze první písmeno velké, např. class EkgData, int rdEkgTime, double ekgValue

Základní pravidlo pro volbu identifikátorů

Vždy volíme
výstižná jména,
žádné nesmyslné zkratky!

Převzato z: www.wise-women.org/tutorials/spellout/

Převzato z http://www.ahdesign.cz/

Pole v C#

budou na cvičení, a tak trochu o nich

Rád jim pletu hlavy…

Pro samostudium

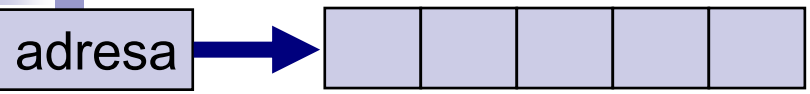
Následující snímky

- jsou určené jako další rozšíření přednášky;
- projdeme je jenom zběžně
- a proto se nebudou ani zkoušet.

System.Array

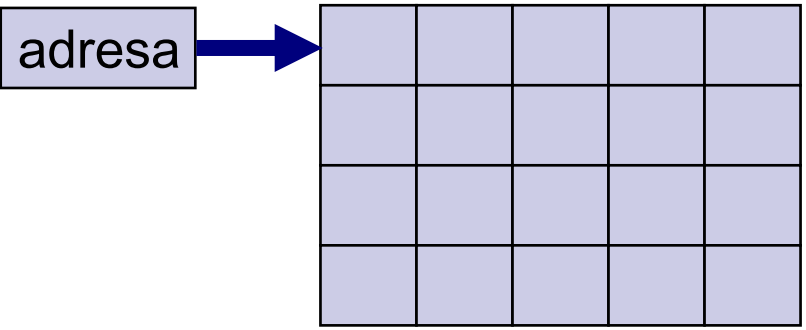
- `System.Array` - základní třída pro všechna pole
- Má několik užitečných metod, např.:
  - `CopyTo(), Find()`
  - `...`
- Několik málo užitečných property jako
  - `Length`
  - `...`
- Mnoho užitečných statických metod jako
  - Array.BinarSearch(); Array.Sort(); Array.Reverse(); Array.Copy();
  - ...

Způsoby uložení polí v paměti

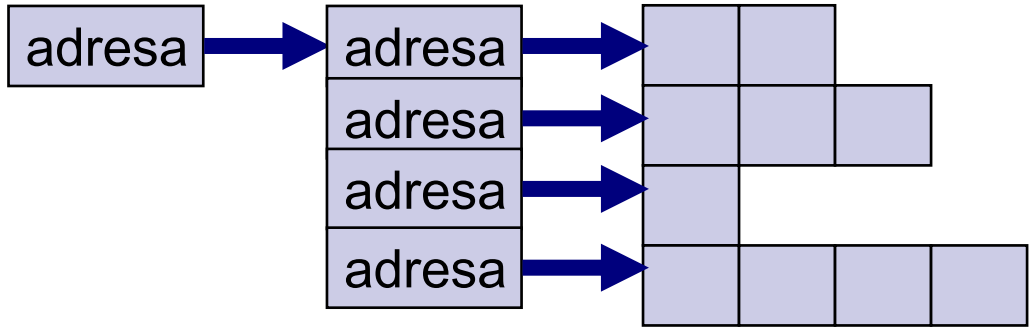


vektor - jednorozměrné pole

- Dvourozměrná pole
  - Pravoúhlé (rectangular) - řádky mají stejnou délku
  - Zubaté (jagged) - pole polí - řádky mají různou délku



pravoúhlé



zubaté

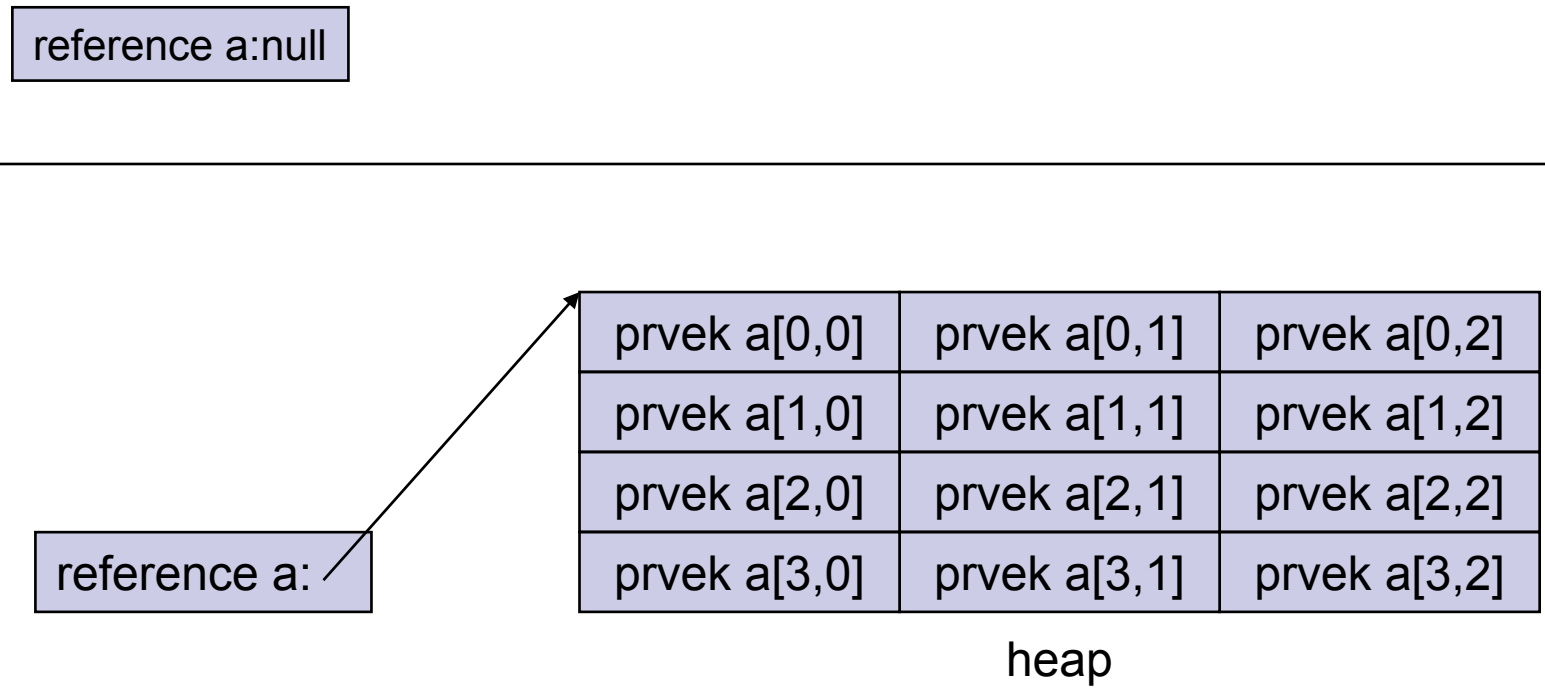
- Vícerozměrná pole
  - opět buď "zubatá" nebo pravoúhlá

Pravoúhlá pole

- Jednorozměrné:
  - `int[] v;`
- I pole musí být vytvořeno
  - `int[] v; // pouze reference`
    `v = new int[4]; // nyní je pole vytvořeno`
- Dvourozměrné:
  - `int[,] a;  //Všimněte si použití čárky`
- I pole musí být vytvořeno
  - `a = new int[4,3]; // nyní je pole vytvořeno`

Prvky v poli

```
int[,] a;
a = new int[4,3];
a[3,2] = 25; // příklad práce
```



Inicializované jednorozměrné pole

- *Všechny tři následující deklarace jsou přípustné pro inicializované pole čtyř bodů Bod:*

```
Bod[ ] body1 = { new Bod(8, 7), new Bod(3, 7),
new Bod(5, 1), new Bod(1, 3) };

Bod[ ] body2 = new Bod[ ] { new Bod(8, 7),
new Bod(3, 7), new Bod(5, 1), new Bod(1, 3) };

Bod[ ] body3 = new Bod[4] { new Bod(8, 7),
new Bod(3, 7), new Bod(5, 1), new Bod(1, 3) };

/* V poslední deklaraci (body3) musí souhlasit počet
inicializací (4) s udanou velikostí pole [4]*/
```

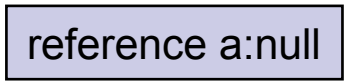
Inicializované vícerozměrná pole

- *Stejně lze inicializovat i pole s více prvky*

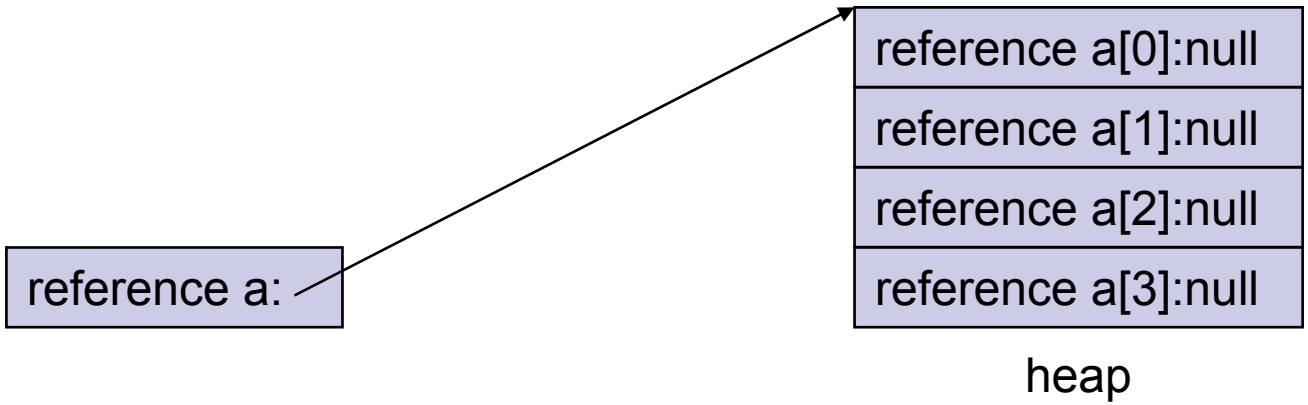
```
int [,] xyint = new int[,] /*new int[2,3]*/
    { { 11, 12, 13 }, { 21, 22, 23 } };
Bod [,] xy = new Bod[,] /* new Bod[2,3]*/
     { { new Bod(1,1), new Bod(1,2), new Bod(1,3) },
       { new Bod(2,1), new Bod(2,2), new Bod(2,3)}
  };
for (int ix = 0; ix < xy.GetLength(0); ix++)  /* 0..1 */
  for (int jy = 0; jy < xy.GetLength(1); jy++) /* 0..2 */
    { Bod b = xy[ix, jy];   int n = xy.Length;  // n=6
    };
```

Zubaté pole (1)

```
int[][] a;      //Všimněte si dvojitých závorek
```

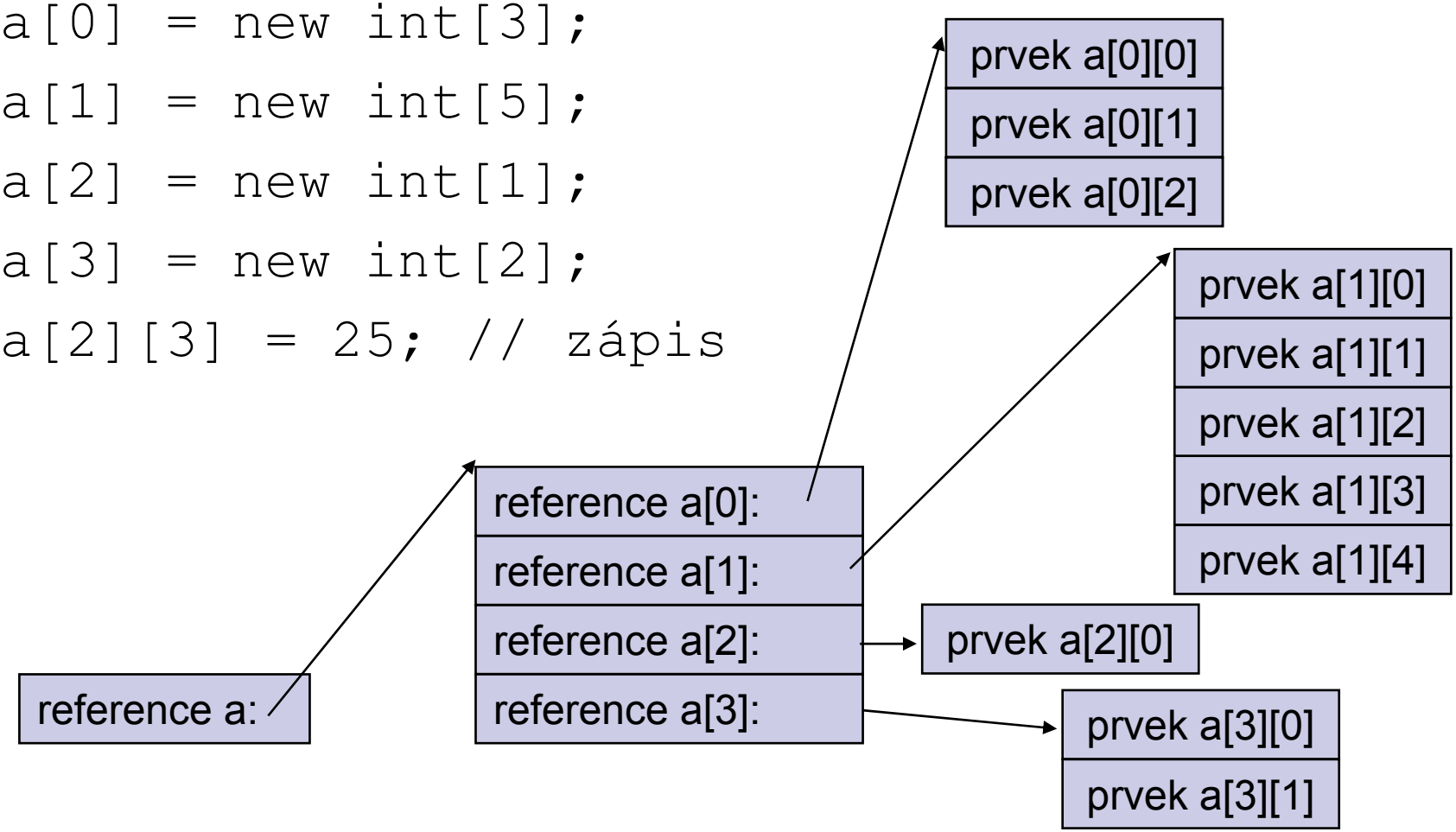


```
a = new int[4][];
```



Zubaté pole (2)

- Každý prvek pole musíme ještě inicializovat
  - `a[0] = new int[3];`
  - `a[1] = new int[5];`
  - `a[2] = new int[1];`
  - `a[3] = new int[2];`
  - `a[2][3] = 25; // zápis`



Inicializované zubaté pole

```
int[ ][ ] b1 = new int [4][ ] {   new int[3] { 00, 01, 02 },
                                  new int[5] { 10, 11, 12, 13, 14 },
                                  new int[1] { 21 },
                                  new int[2] { 30, 31 }
                              };
int[ ][ ] b2 = new int[ ][ ] {    new int[3] { 00, 01, 02 },
                                  new int[5] { 10, 11, 12, 13, 14 },
                                  new int[1] { 21 },
                                  new int[2] { 30, 31 }
                              };
int[ ][ ] b3 = new int[ ][ ] {    new int[ ] { 00, 01, 02 },
                                  new int[ ] { 10, 11, 12, 13, 14 },
                                  new int[ ] { 21 },
                                  new int[ ] { 30, 31 }
                               };
```

```
int i = b1[1 ][2 ]; // výsledek 12
```

Pole jako parametr

- Referenční typ, metoda modifikuje předané pole

```
static void Main()
 { int [ ] ar = new int[ ] {  11, 12, 13 };
     Mocniny(ar);
      // ar -> {  121, 144, 169 }
    }
   static void Mocniny(int [] ar)
    { for (int i = 0; i < ar.Length; i++) ar[i] *= ar[i];
     }
```

Kopie pole 1/3

- Špatně

```
int [ ] copy = ar;
```

Pouze kopie reference (*a shallow copy*)

Kopie pole 2/3

- Pomalé kopírování po elementech

```
int size = ar.Length;
int[] copy = new int[size];
for (int i = 0; i < size; i++) copy[i] = ar[i];
```

- Rychlejší, ale omezené na všechny prvky zdroje

```
const int STARTINDEX = 0; // eventuálně i jiné číslo >= 0
int[ ] copy = new int[ar.Length+ STARTINDEX ];
ar.CopyTo(copy, STARTINDEX);
// vždy se kopírují všechny elementy zdrojového pole, zde ar
```

Array.Copy() - dvě varianty

```
int len = ar.Length;
int [ ] copy  = new int[len];
int [ ] half = new int[len/2];
Array.Copy(  ar, copy, ar.Length); // kopie všeho
Array.Copy(  ar,                   // Zdrojové pole
             len/2,                // Start index zdroje
             half,                 // Cílové pole
             0,                    // start index cíle
             half.Length);         // Kolik prvků kopírovat
```

...KONEC...

a nashledanou příště