Vývoj aplikací v prostředí .NET

Katedra řídicí techniky,
ČVUT-FEL Praha

2. přednáška

Dva cíle dnešní přednášky

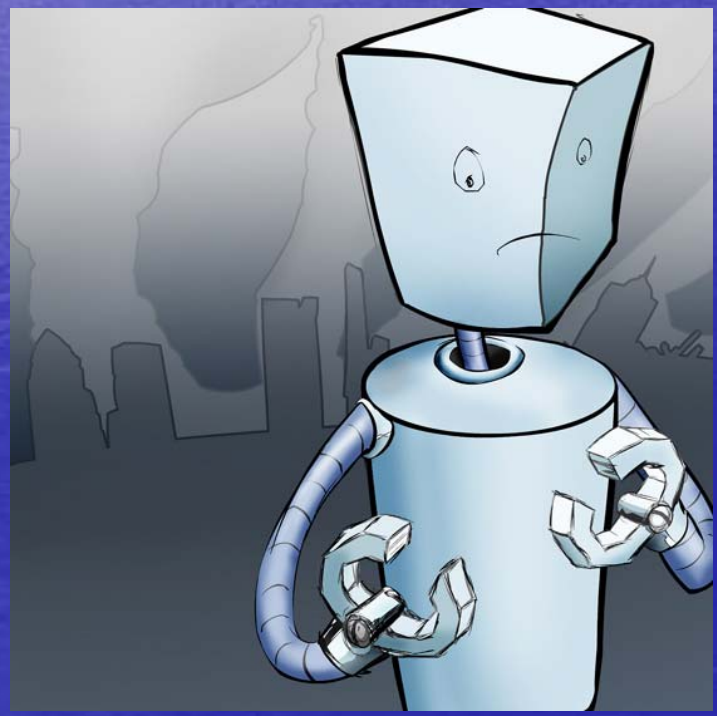
- Konstruktory objektů
- Speciální členy
- Dědičnost objektů

Přetypování objektů

- Property

Delegate + Event

Konstruktory



Konstruktor

- *Speciální metoda volaná automaticky při vzniku objektu*

```
struct BodS  {
  public int x, y;
  public int GetMax() { return x>y ? x : y; }
  public BodS(int x, int y) { this.x=x; this.y=y; }
}

static void Main(string[] args)
{
        Bod b1 = new Bod(1,2);
        Bod b2 = new Bod(4, 5);
 }
```

Vlastnosti konstruktoru

- **Konstruktor nevrací žádný parametr, dokonce ani void**
- **Klíčové slovo new znamená volání konstruktoru** v C#, nikoliv nutně dynamickou alokaci jako v C++ nebo Java. *Systém rozhodne podle typu objektu, kde ho alokuje. Struktury se např. vytvoří vždy na zásobníku.*
- Výchozí (bezparametrový) konstruktor se automaticky vytvoří u struktur a nuluje jejich prvky. A proto ho tam nelze deklarovat.

```
Bod b = new Bod();  // ≈ public Bod() { x=0; y=0; }
```

  - *U struktur můžeme deklararovat jen konstruktor s parametry.*

Konstruktor u struct a class

```
struct BodS  {      public int x, y;
                    public int GetMax() { return x>y ? x : y; }
                    public Bod(int x, int y) { this.x=x; this.y=y; } }
```

```
class Bod    {      public int x, y;
                    public int GetMax() { return x>y ? x : y; }
                    public Bod(int x, int y) { this.x=x; this.y=y; }
                    public Bod() { x=0; y=0; }   }
```

```
BodS b;  // ≡ BodS b = new Bod();
BodS b0 = new Bod();
BodS b1 = new Bod(1,2);
BodS b2 = new Bod(4, 5);
```

```
Bod b; // b=null
Bod b0 = new Bod();
Bod b1 = new Bod(1,2);
Bod b2 = new Bod(4, 5);
```

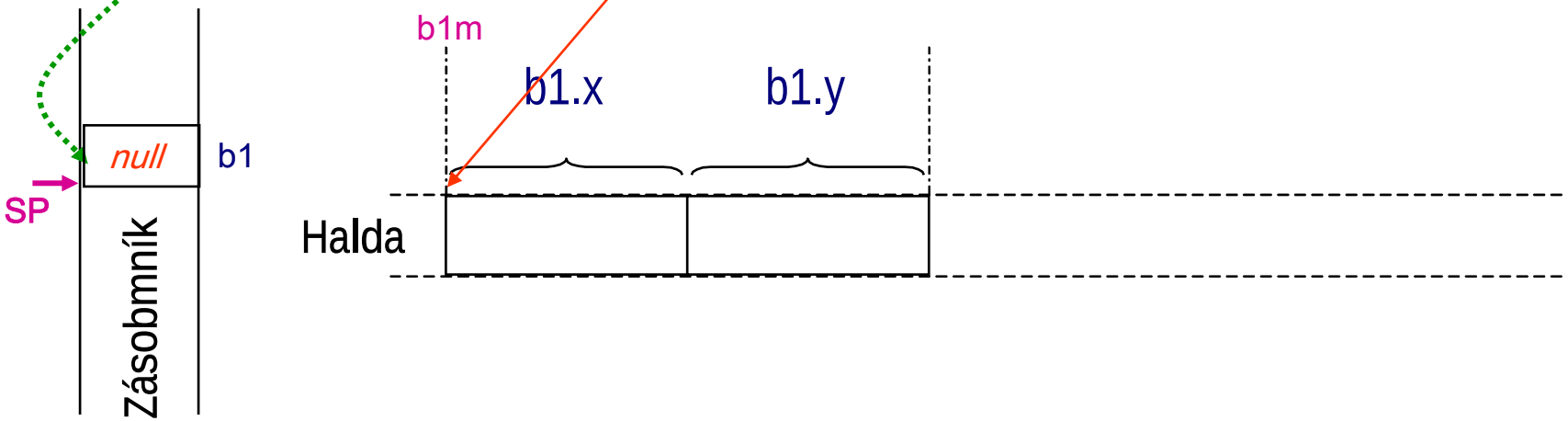
Bezparametrový konstruktor Bod() *se u třídy vytvoří automaticky pouze při absenci deklarací jiných konstruktorů. Má-li třída nějaký jiný konstruktor, pak musíme deklarovat i její bezparametrový, chceme-li ho rovněž používat.*

Konstruktor u typu class

class je referenční typ

Volání konstruktoru

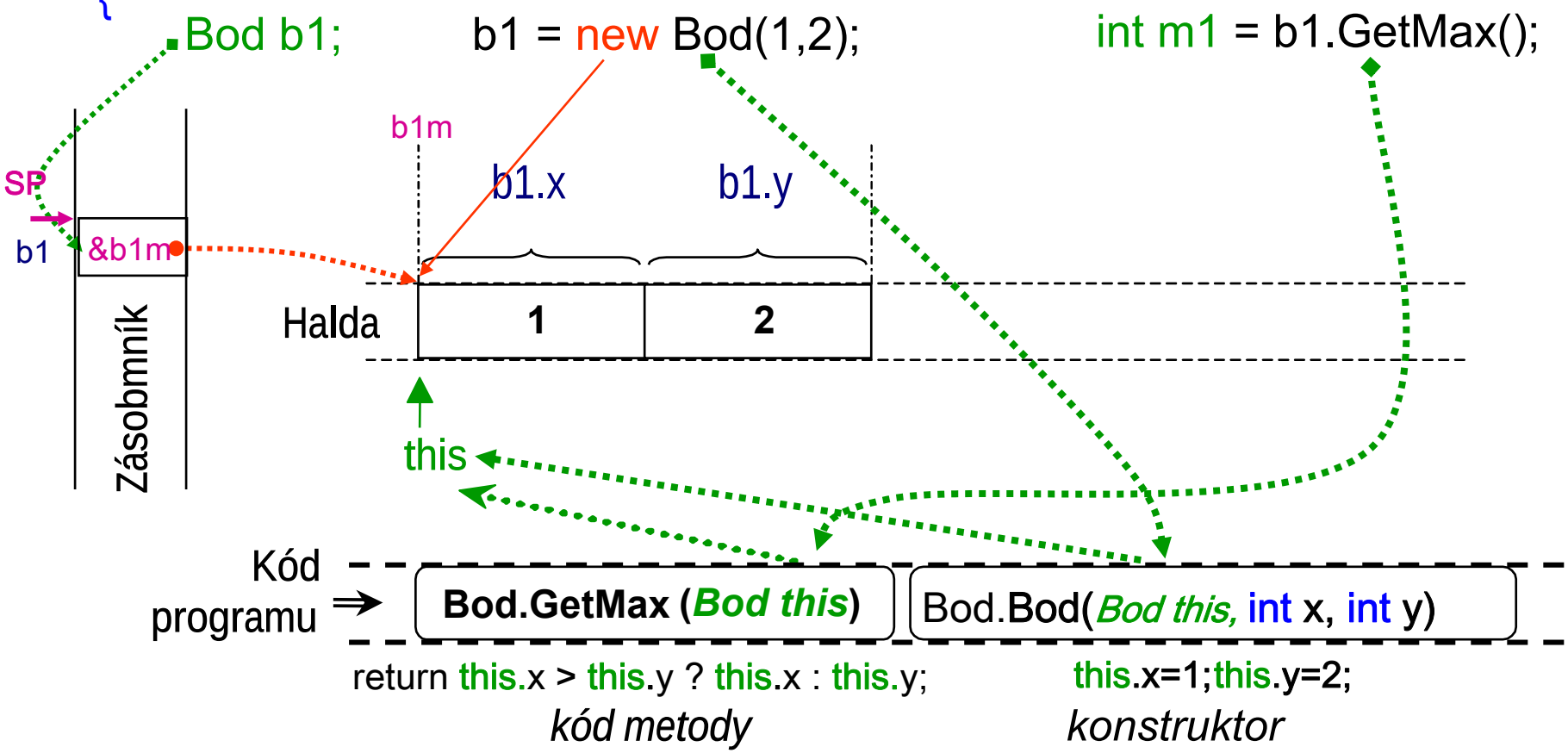
```
class Bod {    public int x, y; public int GetMax() { return x > y ? x : y; }
               public Bod(int x, int y) { this.x=x; this.y=y; }
           }
static void Main(string[] args)
 {
    Bod b1;          b1 = new Bod(1,2);
```



kód metody *konstruktor*

Volání konstruktoru

```
class Bod {    public int x, y; public int GetMax() { return x > y ? x : y; }
               public Bod(int x, int y) { this.x=x; this.y=y; }
           }
static void Main(string[] args)
 {
```

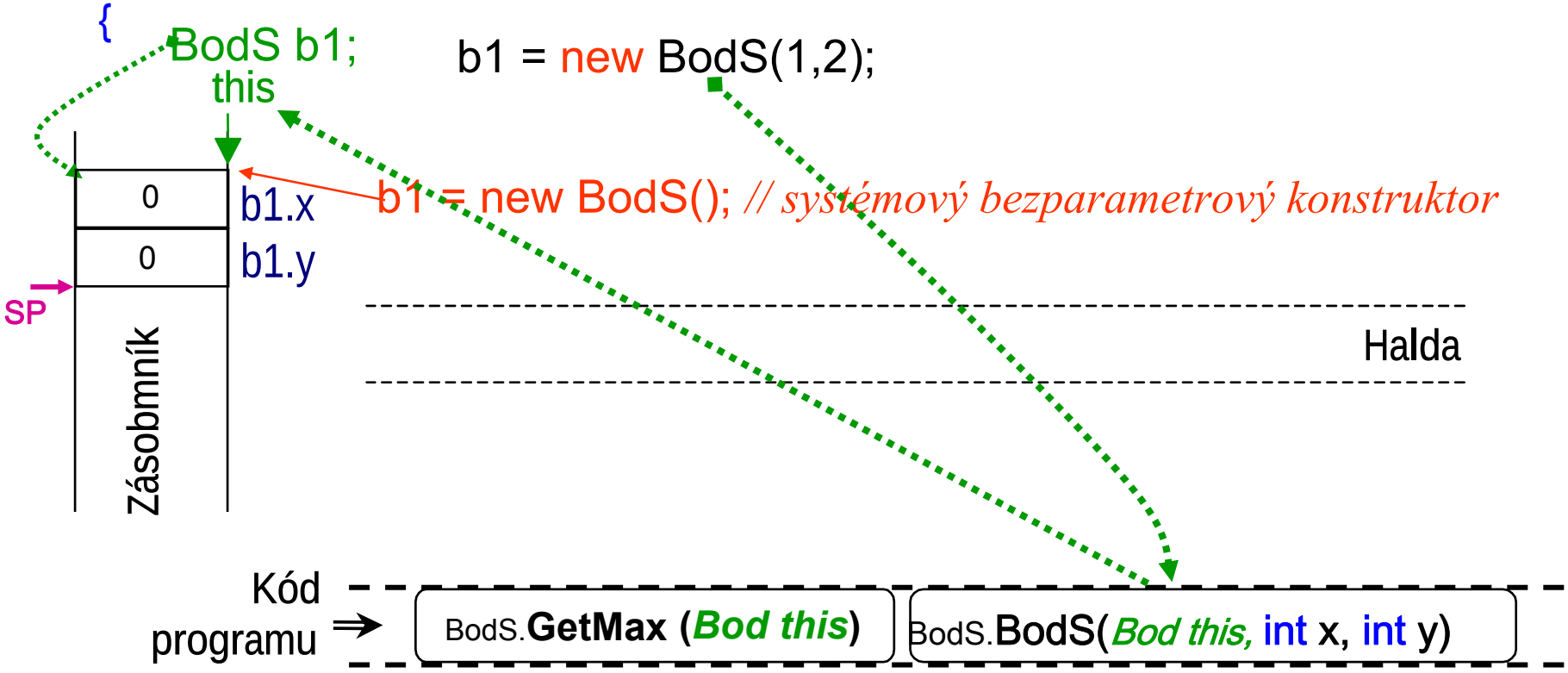


Konstruktor u typu struct

struct je hodnotový typ

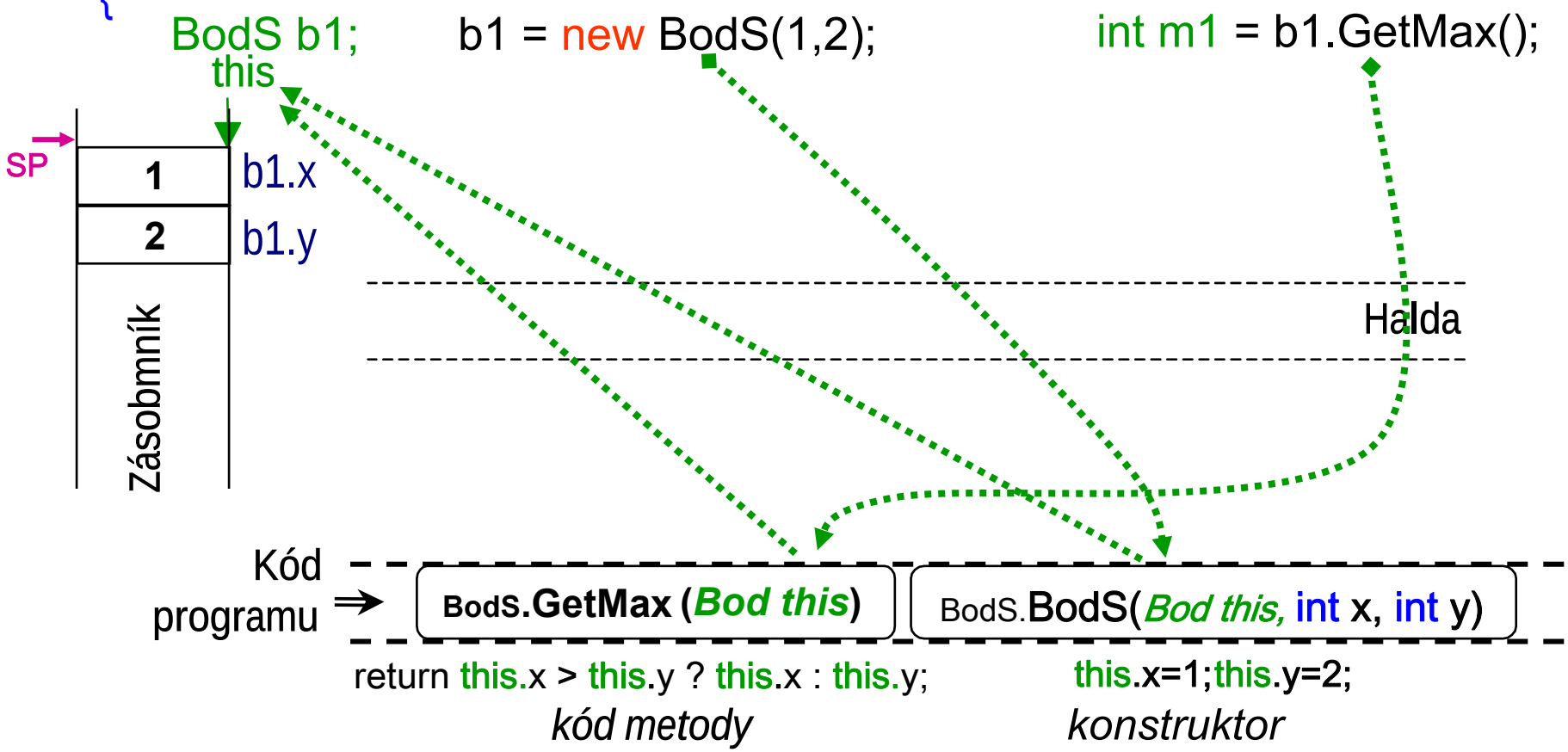
Volání konstruktoru

```
struct BodS {  public int x, y; public int GetMax() { return x > y ? x : y; }
               public BodS(int x, int y) { this.x=x; this.y=y; }
            }
static void Main(string[] args)
 {
```



Volání konstruktoru

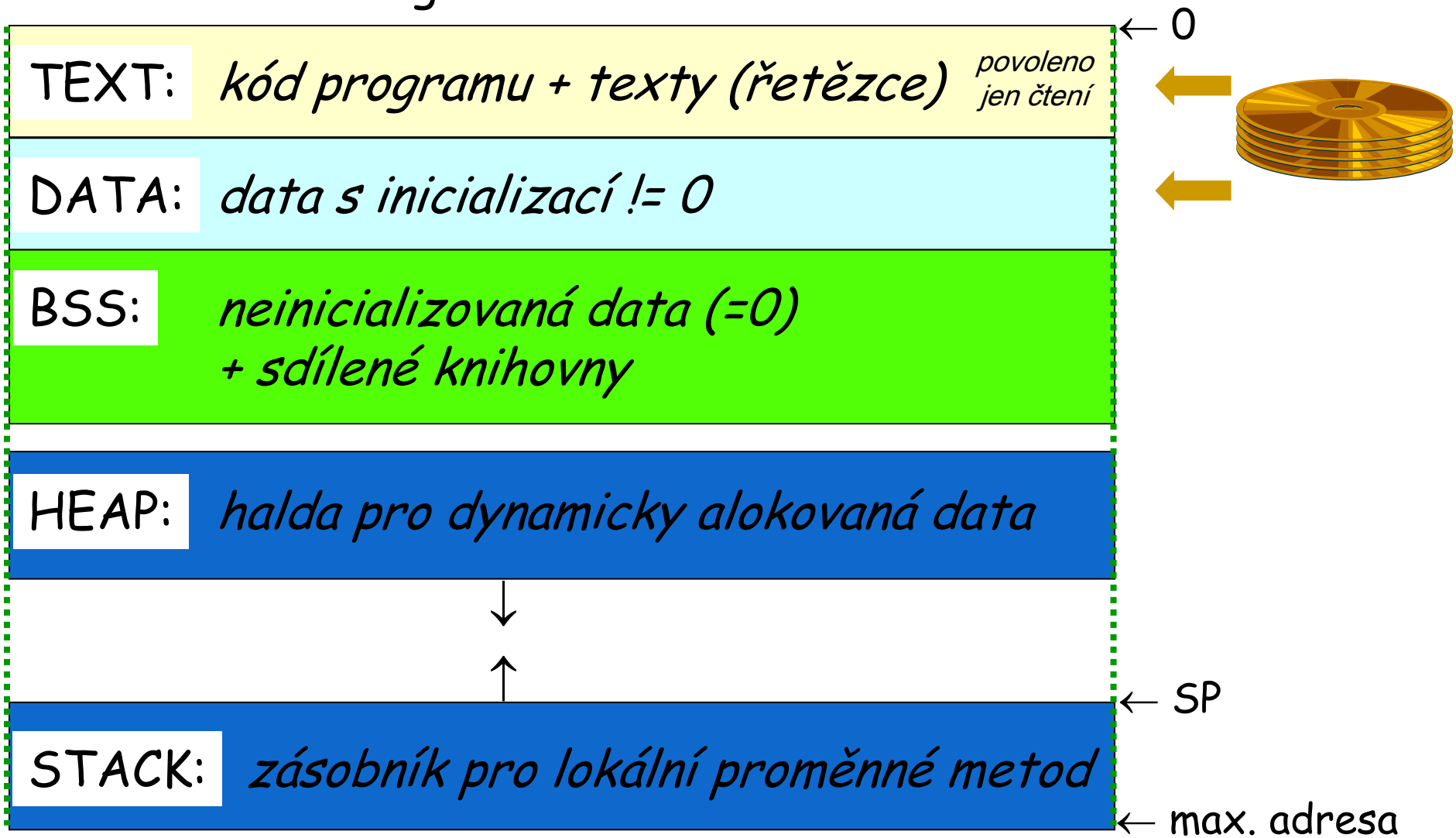
```
struct BodS {  public int x, y; public int GetMax() { return x > y ? x : y; }
               public BodS(int x, int y) { this.x=x; this.y=y; }
            }
static void Main(string[] args)
 {
```



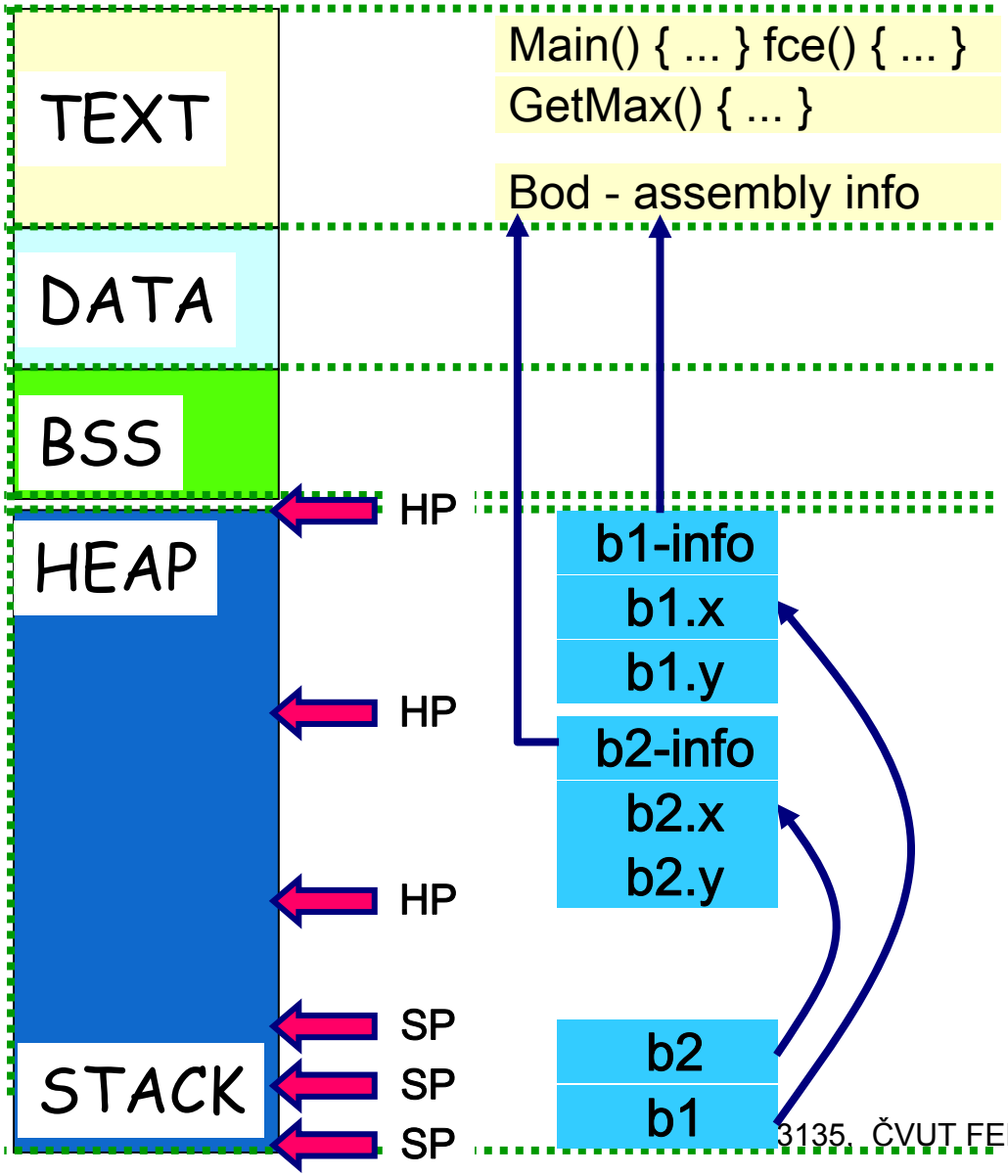
Statické prvky tříd

Segmenty obecného programu v paměti

NÁZEV: obsah segmentu



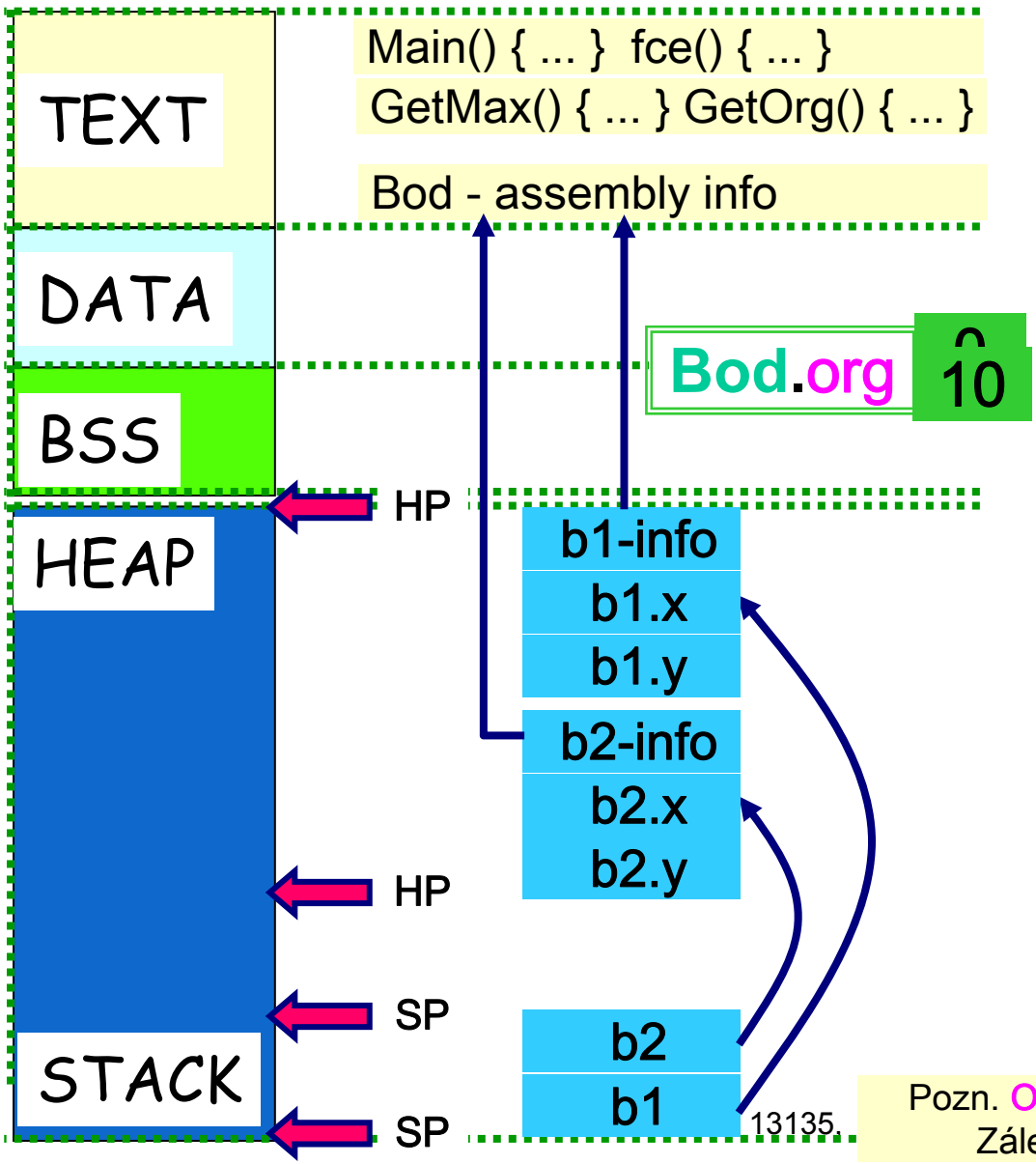
Třída bez statických prvků




```
class Bod
 { public int x, y;
   public int GetMax()
    { return x>y ? x : y; }
 }
```

```
class Program
{
static void Main()
{  fce();
 }
static void fce()
{ Bod b1 = new Bod();
   Bod b2 = new Bod();
 }
}
```

Třída se statickým prvkem




```
class Bod
 { public int x, y;
   public int GetMax() {...}
   public static int org;
   public static int GetOrg()
       { return org; }
 }
```

```
class Program
{
static void Main()
{  fce();
 }
static void fce()
{  Bod.org=10;
   Bod b1 = new Bod();
   Bod b2 = new Bod(); }
}
```

13135

Pozn. Org může být v jak DATA, tak v BSS.
Záleží to na konkrétním překladači.

Statické metody a data

- *Na statické čteny tříd musíme mimo deklarační prostor jejich třídy odkazovat jménem třídy, příklad* Bod.**org** .
- *Statické členy patří k typu, nikoliv k instanci třídy. Jsou analogií globálních dat a funkcí používaných v jazyce C.*
- *JIT inicializuje statická data při prvním nahrání modulu třídy do paměti, což nejčastěji bývá při spuštění programu.*
- *Statické metody mohou přistupovat pouze ke statickým členům tříd.*

Statický konstruktor

- Pro inicializaci statických členů tříd si můžeme definovat statický konstruktor.
- Třída může mít nejvýše jeden statický konstruktor.
- U statického konstruktoru se nesmějí použít modifikátory přístupu, tedy public, apod.
- Statický konstruktor se volá automaticky při prvním nahrání modulu třídy do paměti.
- Statický konstruktor se nedědí !!!

možno zapsat i přímo u členu

```
class Bod  {
   public int x, y;
   public static int org;
   static Bod() { org=5; }
        /*....*/
        }
```

```
class Bod {
      public int x, y;
      public static int org=5;
            /*...*/
             }
```

Konstanty

Patří také k typu jako statické členy

Odkazujeme na ně jménem třídy

Konstanty zpřehledňují program

```
class Bod
 { public int x, y;
   public int GetMax() {...}
   public static int org;
   public int XYORG_VYCHOZI=10;
   public static int GetOrg()  { return org; }
  }
```

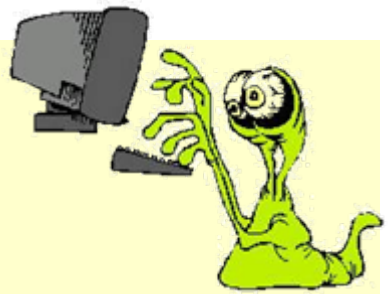
```
class Program
{
  static void Main() {  fce();  }
  static void fce()
  {  Bod.org=10;   // číslo 10 neříká nic
     Bod.org=Bod. XYORG_VYCHOZI; // mnohem jasnější
     Bod b1 = new Bod();  Bod b2 = new Bod();
   }
}
```

Prvky readonly

```
public class Mereni
   {    public const int MERITKO=10;
        public readonly int startTime;
        public Mereni()
        {   startTime
            = Environment.TickCount / MERITKO;
        } }
```

run-time inicializace v konstruktoru

- *Hodnota konstanty se zná v době překladu → lepší optimalizace kódu*
- *Konstanty nedovoluji run-time inicializaci → tu umožní jen readonly*
- *Členy readonly jsou na rozdíl od konstant prvky instancí třídy → více paměti*



K čemu slouží statické prvky?

- Jako přímé náhrady globálních proměnných a funkcí, tedy pro prvky přístupné v celém programu
- Můžeme jimi vytvořit třídu s jednou instancí, k níž přístup odkudkoliv, tzv. Singleton.

Singleton: vzor vytvoření

```
public sealed class Singleton
{  // pozdržení vytvoření
private static readonly Singleton instance = new Singleton();

 // blokace default konstruktoru
 private Singleton() { }

 // jediná cesta k instanci
 public static Singleton Instance  { get { return instance; } }

 /* plus moje případné další členy */
}
```

Singleton: chybová hlášení

```
public sealed class Chyby
{ private static readonly Chyby instance = new Chyby();
  private Chyby() { }
  public static Chyby Instance {
    get   {  if (!souborNacten) instance.CtiSoubor();
             return instance;   }    }

 private static bool nacteno = false;
 private string[ ] textyChyb;
 private void CtiSoubor()  { textyChyb = /* ... */ ; nacteno = true; }
 public string Chyba(string kod) { /* ... */  return nalezeny_text; }
}
```

Příklad použití chybových hlášení

```
static void Main(string[ ] args)
 {
  if(args.Length<1)
  Console.WiteLine(
        Chyby.Instance.Chyba("ArgLen0")
                        );
 }
```

ke všem public prvkům třídy Chyby mám přístup odkudkoliv v celém programu

Dědičnost a přetypování

"Copy-Edit" tvorba nové třídy

Kopíruji chybu, místo jedné mám teď dvě!

```
class Bod
{    protected int x, y;
     public int GetMax() { return x<y ? x : y; }
     static public int GetOrg ()  { return 0; }
     public Bod(int x, int y) { this.x =x; this.y=y; }
}
```

Vznikne kód metod
1. Bod.GetMax()
2. Bod.GetOrg()
3. Bod.Bod(int, int)

```
class BodN
{    protected int x, y;
     protected double frekvence;
     public int GetMax() { return x<y ? x : y; }
     static public int GetOrg () { return 0; }
     public BodN(int x, int y, double frekvence)
          { this.x =x; this.y=y; this.frekvence=frekvence; }
     public double GetOmega() { return 2*Math.PI*frekvence; }
}
```

Vznikne kód metod
4. BodN.GetMax()
5. BodN.GetOrg()
6. BodN(int, int, double)
7. BodN.GetOmega()

Tvorba nové třídy děděním

Chyba zůstává jen zde

```
class Bod
{    protected int x, y;
     public int GetMax() { return x<y ? x : y; }
     static public int GetOrg ()  { return 0; }
     public Bod(int x, int y) { this.x =x; this.y=y; }
}
```

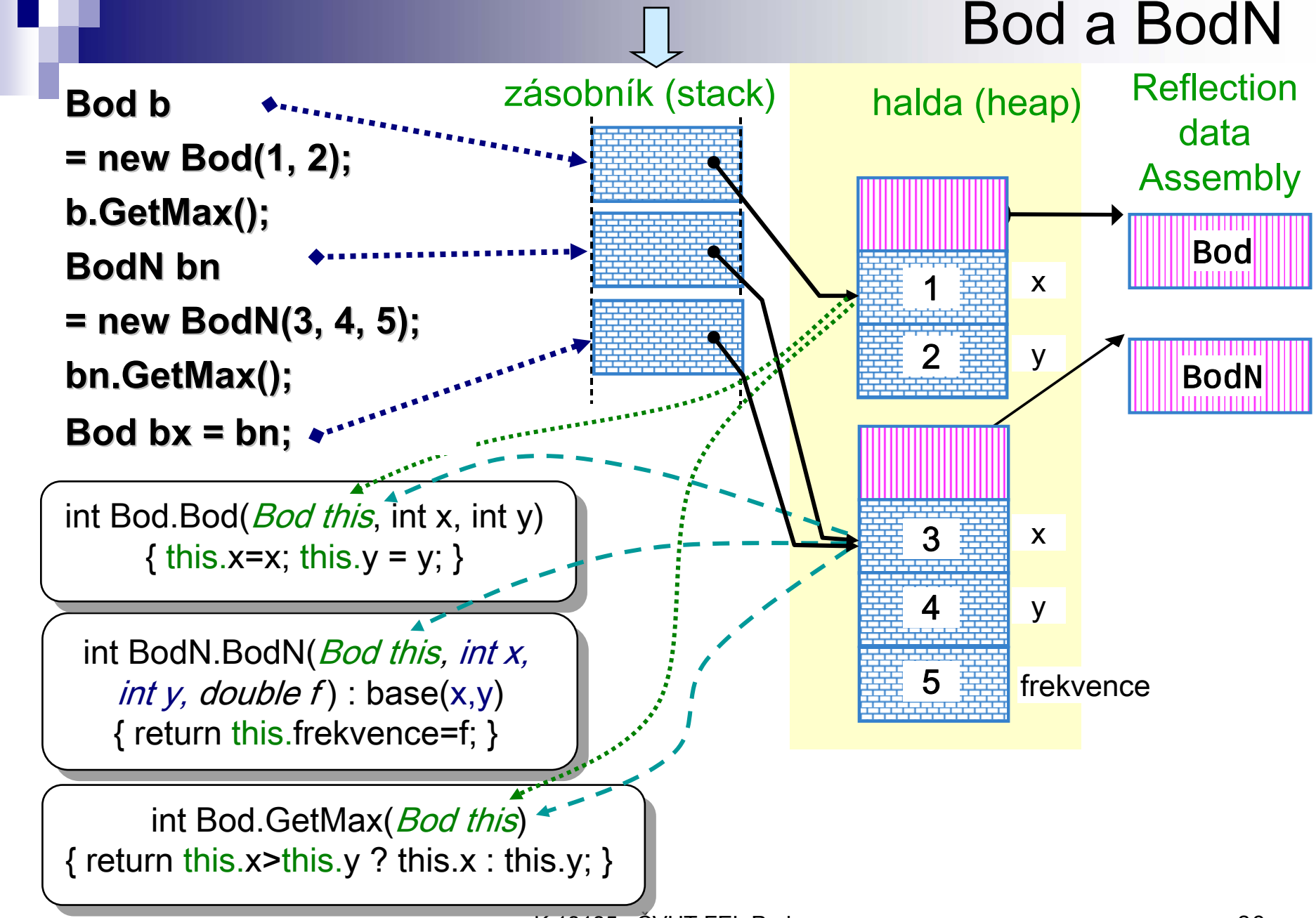
Vznikne kód metod
1. Bod.GetMax()
2. Bod.GetOrg()
3. Bod.Bod(int, int)

```
class BodN : Bod
{    protected double frekvence;
     public BodN(int x, int y, double frekvence) : base(x,y)
        { this.frekvence=frekvence; }
     public double GetOmega() { return 2*Math.PI*frekvence; }
}
```

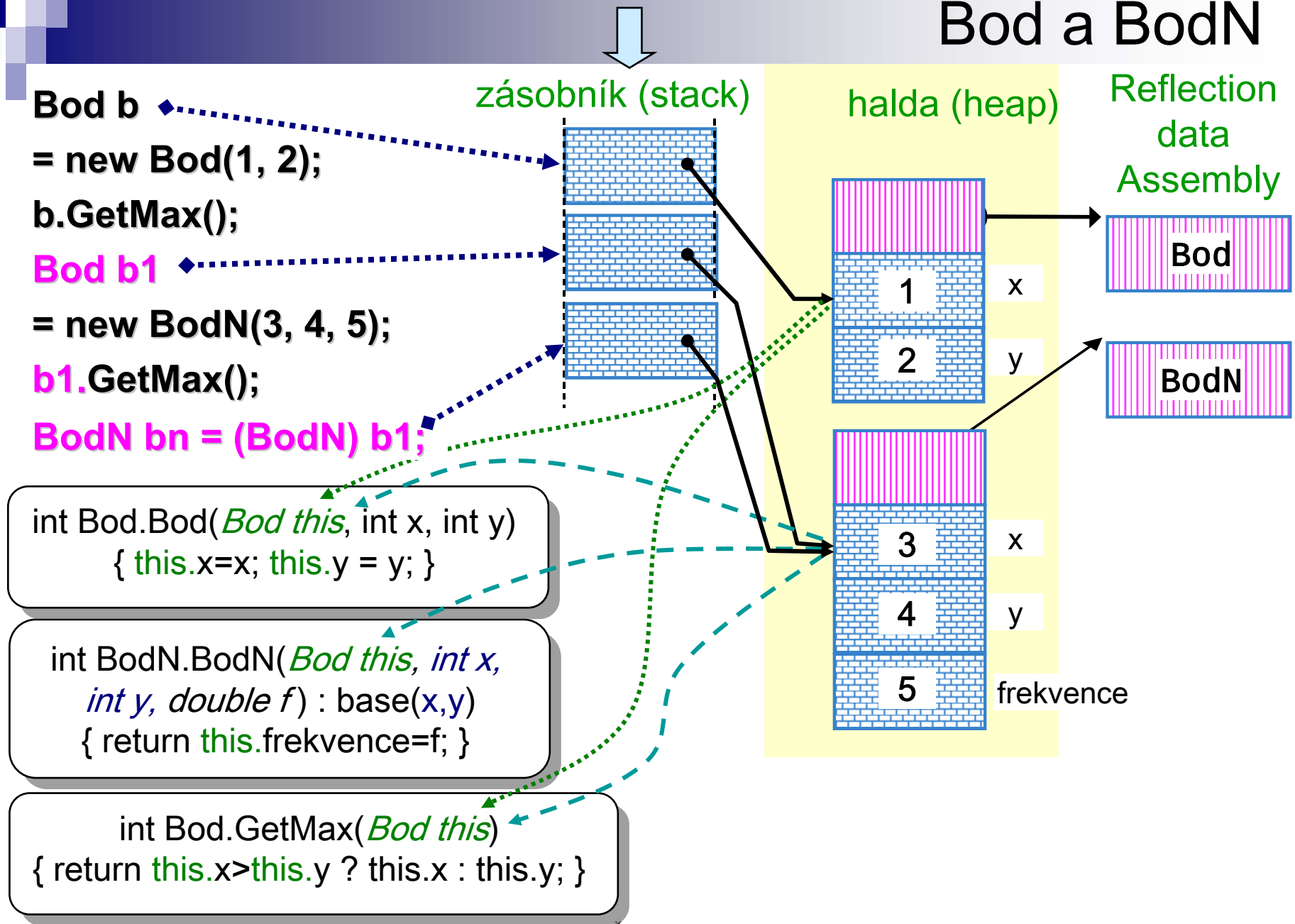
Vznikne kód metod
4. BodN(int, int, double)
5. BodN.GetOmega()

Moje chyba se nešíří do dalšího kódu a
výsledný program je navíc optimálnější o dvě metody a kus konstruktoru.

Bod a BodN


Bod b
= new Bod(1, 2);
b.GetMax();
BodN bn
= new BodN(3, 4, 5);
bn.GetMax();
Bod bx = bn;
zásobník (stack)
halda (heap)
Reflection data Assembly
Bod
BodN
1
2
x
y
3
4
5
x
y
frekvence
int Bod.Bod(Bod this, int x, int y)
{ this.x=x; this.y = y; }
int BodN.BodN(Bod this, int x,
int y, double f) : base(x,y)
{ return this.frekvence=f; }
int Bod.GetMax(Bod this)
{ return this.x>this.y ? this.x : this.y; }

Bod a BodN


Bod b
= new Bod(1, 2);
b.GetMax();
Bod b1
= new BodN(3, 4, 5);
b1.GetMax();
BodN bn = (BodN) b1;
zásobník (stack)
halda (heap)
Reflection data Assembly
Bod
BodN
1
2
x
y
3
4
5
x
y
frekvence
int Bod.Bod(Bod this, int x, int y)
{ this.x=x; this.y = y; }
int BodN.BodN(Bod this, int x,
int y, double f) : base(x,y)
{ return this.frekvence=f; }
int Bod.GetMax(Bod this)
{ return this.x>this.y ? this.x : this.y; }

Přehled objektových rozdílů class a struct v C#

	class	struct
typ objektu	**referenční**	**hodnotový**
dědičnost	**ano**	**ne**
rozhraní (interface)	**ano**	**ano**
bezparametrový konstruktor	**lze deklarovat**	**nesmí se deklarovat**
destruktor	**ano**	**ne**

Přetypování

Přetypování

- Implicitní – provádí se automaticky,
  např. int i=12345; double d=i;
- Explicitní – lze provést operátorem (*nový-typ*)
  např. i = **(**int**)** d;
- Run-time výjimka – neexistuje implicitní/explicitní konverze, nebo je mimo rozsah;
- Pro jednoduché hodnotových typy se kontrola přetečení při přetypování řídí pomocí **unchecked/checked**
  např. **byte b = checked((byte)i);**

Výchozí nastavení ->Properties->Build->Advanced

Check for Arithmetic Overflow/Underflow

Přetypování referenčních typů

- **U referenčních typů měním přetypování jen deklarační prostor, s nímž mohu pracovat, hodnota se nemění.**
- **Přetypování se provádí automaticky, pokud existuje implicitní konverze.**
  - Implicitní konverze existuje třeba při přetypování z odvozené třídy na základní.
  - Někdy lze definovat implicitní operátor
- **Explicitní přetypování lze provést, pokud existuje explicitní konverze.**
  - Explicitní konverze existuje třeba při přetypování ze základní třídy na odvozenou třídy.
  - Někdy lze definovat i explicitní operátor.

Přetypování referenčních typů

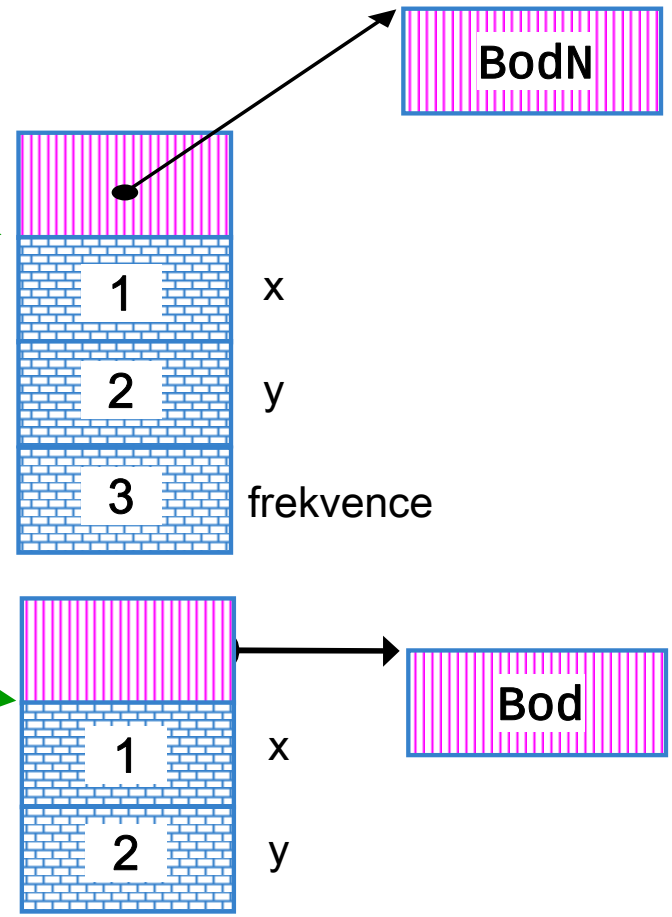
- Všechny C# objekty jsou odvozené od třídy Object – a proto na Object lze vždy implicitně přetypovat

```
Bod b1= new BodN(3, 4, 5);
Console.WriteLine(b1.GetType()); //→ "Prog1.BodN"
Object o = b1;
Console.WriteLine(o.GetType());   //→ "Prog1.BodN"
```

- Při explicitním přetypování přetypování ze základní na odvozenou třídu se automaticky provádí kontrola správnosti pomocí dat uložených v assembly.

Přetypování

```
static void Main()
    {     BodN bn = new BodN(1,2,3);
          Bod  b  = bn;
          Object o = bn;   o=b;
          Bod f = (Bod) o;
          BodN fn = (BodN) b;
          Bod bx = new Bod(10,20);
/*!!*/     BodN fx = (BodN) bx;
        // run-time chyba
    }
```



Rozšíření přetypování

- Můžeme definovat přetypovací operátory

```
class Bod
  { public int x, y; public Bod(int x, int y) { this.x = x; this.y = y; }
    public static implicit operator Bod(int i)
        {  return new Bod(i,0);    }
    public static explicit operator Bod(string s)
        { return new Bod(int.Parse(s), 0);   }
  }
```

```
static void Main()
{  Bod bb = new Bod(1,2);
   int i=12345;  bb = i; bb = (Bod)"54321";
}
```

Operátory nelze definovat pro přetypování mezi základními a odvozenými třídami...

Boxing - unboxing

Umožňuje přetypovat hodnotové typy na Object

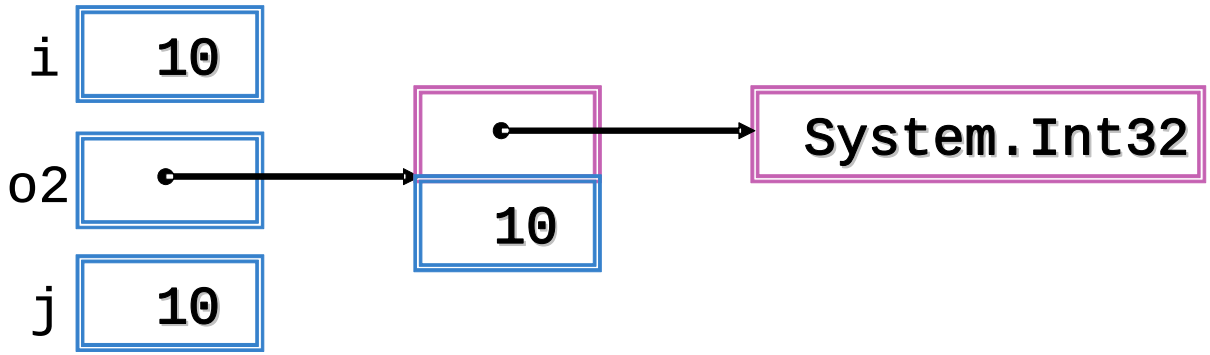
Boxing

- **alokace "boxu" a kopírování hodnoty do něho**

Unboxing

- **kontrola typu v boxu a kopírování hodnoty z něho**

```
int i=10;
Object o2 = i;
int j = (int) o2;
```



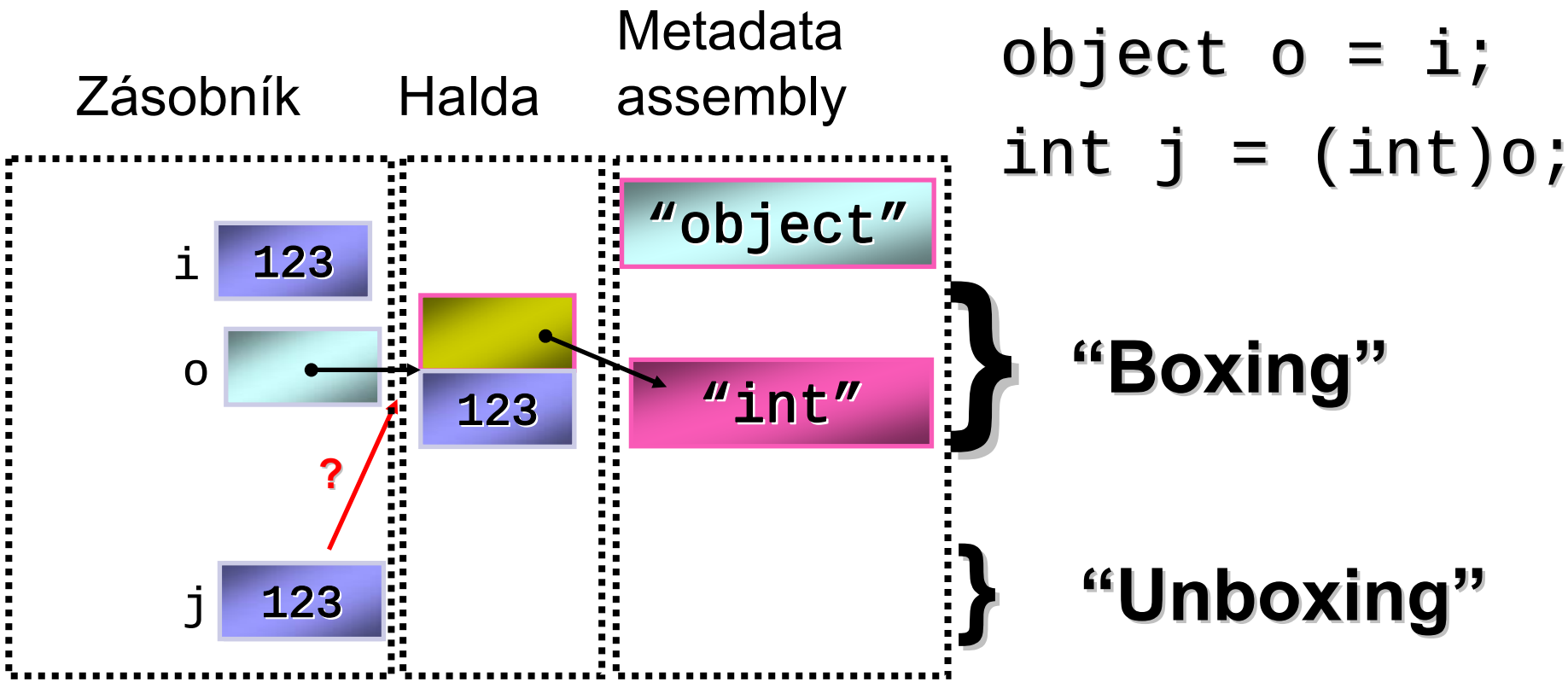
// BodN f = (BodN) o2; // run time chyba <undefined value>

Boxing / Unboxing, aneb vše je objekt

Boxing - alokace "boxu" a kopírování hodnoty do něho

Unboxing - kontrola typu v boxu a kopírování hodnoty z něho

```
int i = 123;
object o = i;
int j = (int)o;
```



// BodN f = (BodN) o; // run time chyba <undefined value>

Property

Prodej bažinaté nemovitosti (property) o velikosti 1 akr

[Z www.grinningplanet.com/]

Java versus C#

Java

```
private int radius;
public int getRadius()
  { return radius; }
public void setRadius(int value)
 {  if (value < 0)
        radius = 0;
      else
        radius = value;
    }
```

```
int s = kruh.getRadius();
kruh.setRadius(s+1);
```

C#

```
private int radius;
public int Radius
   { get { return radius; }
     set {
       if (value < 0)
         radius = 0;
       else
         radius = value;
     } }
```

```
kruh.Radius++;
```

Property - náhražka get/set metod

uložená hodnota

typ property

jméno property

```
class Data {
    FileStream s;
    public string FileName {
        set { s = new FileStream(value, FileMode.Create);
              }
        get {  return s.Name;  }
    }
}
```

klíčové slovo: **hodnota přiřazená do property**

```
Data d = new Data();
d.FileName = "myFile.txt";     // → set("myFile.txt")
string s = d.FileName;         // → get()
```

JIT překladače často přeloží jako inline get/set methody
➔ není ztráta výkonu

Property jako náhrada datového členu

```
class C
   {
      private static int size;
      public static int Size
      {
         get { return size; }
         set { size = value; }
      }
   }
```

```
C.Size = 3;
C.Size += 2;  // Size = Size + 2;
```

Můžeme si zde dát breakpoint na manipulace s daty

Vynechání get nebo set

```
class Account
  {   long balance;
      public long Balance {    // pouze ke čtení
            get { return balance; }
                        }
  }
```

```
x = account.Balance;        // ok
account.Balance = ...;      // chyba
```

Indexery = *speciální typ properties*

```
public class Clovek
 { private string jmeno;  // datové členy properties private
   public Clovek(string text) { jmeno = text; }
   public string Jmeno { get {return jmeno; } }
   /* … */
 }
```

Pozn.: Pokud by se 'jmeno' nastavovalo jen v konstruktoru, dal by se použít read only element

```
public class Seznam
 {private Clovek[ ] osoby = new Clovek[4];
   public Clovek this [int index]
   { get { return osoby[index]; }
     set { osoby[index] = value; }
   } }
```

Přístup na indexery

```
static void Main()
   {
      Seznam lide = new Seznam();
      lide[0] = new Clovek("Pepa");
      lide[1] = new Clovek("Honza");
      lide[2] = new Clovek("Karel");
      lide[3] = new Clovek("Lucie");

      for (int i=0; i<4; i++)
            Console.WriteLine(lide[i].Jmeno);
}
```

Kdy napsat property?

- *používejte property, když se jedná logickou náhradu datového členu*

```
private string name;
public string Name
    { get { return name; }
      set { name = value.Trim(); }
                        // úprava hodnoty před zápisem
    }
```

Použijte metodu

- pro náročné operace - tím zdůrazníme vhodnost uchování jejich výsledku.
- pro konverze jako třeba **Object.ToString()**.
- když **get** člen má vedlejší efekt.
- když volání téhož členu dvakrát za sebou dává odlišné výsledky.
- když pořadí operací je důležité pro výsledek, u property se get a set mohou volat v libovolném pořadí při výpočtu výrazu.
- člen je statický, ale vrací hodnotu, která může být změněna.
- při vracení pole - property vracející pole jsou velice matoucí !

Vytvořte indexované property

- jen pro datové členy mající charakter pole.
- používejte pouze jeden index.
- jako index volte jedině typ integer nebo string.
- nevytvářejte indexované property a metody, které jsou semanticky ekvivalentní.
- pojmenujte Item ty elementy třídy, které jsou typu třída nebo struktura a mají definovaný indexer.

```
class Skupina
{  public read only string Jmeno;
   public class Seznam  Item;
/* */
}
```

Delegate

Rozšířený ukazatel na funkci,

známý z jazyka C++

Hlavní prvky tříd

```
class Class1 {
    ... fields, constants...          // pole, konstanty
    ... methods...                    // metody
    ... constructors...               // konstruktory,
    ... destructors...                // destruktory
    ... properties ...                // vlastnosti
    ... indexers ...                  // indexery
    …delegates…                       // cca ukazatelé na metody
    ... events ...                    // události
    ... overloaded operators ...      // přetížené operátory
     …enums…                          // výčtové typy
    …class, struct…                   // vnořené deklarace
}
```

C++ pointer na funkci

1. *Metoda pro obsluhu události*

```
double Mocnina(double d) { return d*d; }
```

2. *Deklarace typu pointer na funkci*

```
typedef double ( *LPMocnina)(double);
```

3. *Definice proměnné typu pointer na funkci*

```
LPMocnina lpMocnina;
```

Testujeme

```
lpMocnina = Mocnina;   // přiřadíme
d = (*lpMocnina) (5);      // voláme
```

Stejný program pomocí C# delegate

1. *Metoda pro obsluhu události*

```
double Mocnina(double d) { return d*d; }
```

2. *Deklarace proměnné typu delegate*

```
delegate double LPMocnina(double d);
```

3. *Definice proměnné typu delegate*

```
LPMocnina lpMocnina;
```

Testujeme

```
lpMocnina = new LPMocnina(Mocnina);
d = lpMocnina(5);
```

Vlastnosti proměnné delegate

- Proměnná typu delegate může být i null, ale pak se nesmí volat, jinak dojde k výjimce.
- Proměnné typu delegate jsou objekty, takže se smí předávat jako parametry a použít jako členy tříd.
- Pokud se volá několik metod a typ delegate vrací hodnotu, tou bude výsledek posledního volání, což platí i pro případný parametr out.
- Parametr předaný delegate se použije pro všechny metody.

Delegáty lze propojit třídy

```
class Prekladac                    // deklarační prostor Funkce
{  private StringBuilder error = new StringBuilder();
   int errorCounter=0;
   private void VypisChybu(string text)
     { errorCounter++; error.AppendLine(text); }
   public delegate void Chyba(string text);
   public void Analyza(string code)
     { Parser p = new Parser(VypisChybu); p.Zpracuj(code); }
}
                                   // deklarační prostor Parser
class Parser
 {  Prekladac.Chyba vypisChyby;
    public void Zpracuj(string code)  {    if(code==null) vypisChyby("No code.");
                                           /* další operace */
                                      }
    public Parser(Prekladac.Chyba vypisChyby) {this.vypisChyby = vypisChyby;}
 }
```

privátní metoda druhé třídy předaná jako klíč k přístupu

Delegát jako předpis operace

```
class Program
{  delegate void DoWithFile(string filename);
    static void ProcessDir(string directory,
                           DoWithFile doWithFile)
    {
      foreach (string file in Directory.GetFiles(directory))
                doWithFile(file);
      foreach (string dir in Directory.GetDirectories(directory))
                ProcessDir(dir, doWithFile);
    }
 /* pokračování na dalším snímku */
```

Prostý tisk

```
static void Main(string[] args)
        {
            ProcessDir(@"C:\Temp", Print);
        }

static void Print(string file)
                { Console.WriteLine(file); }
```

Anonymní metoda

```
static void Main(string[] args)
{
  ProcessDir(@"C:\Temp",
  delegate(string file){ Console.WriteLine(file); }
                  );
}
```

Anonymous Methods

```
formN1.FormClosed
    += delegate(type1 name1, type2 name2...)
    { /*.....*/ };
```

formální parametry

kód metody

- zapíšeme přímo kód metody, ušetříme tvorbu pojmenované metody
- anonymní metoda má přístup na privátní členy svého objektu, podobně jako každý delegate
- return v anonymní metodě, znamená jen ukončení anonymní metody

Omezení

- anonymní metoda nesmí být přiřazená objektu
- anonymní metoda nesmí přistupovat na ref a out parametry metody, která ji vytváří

Možnost vynechat formální parametry

```
static void Main(string[] args)
 {
    int count=0;
    ProcessDir(@"C:\Temp",
                       delegate(string file){ count++; } );
    count=0;
    ProcessDir(@"C:\Temp", delegate{ count++; } );
 }
```

Omezení

- formální parametry lze vynechat jen tehdy, pokud mezi nimi není nějaký parametr s modifikátorem **out**

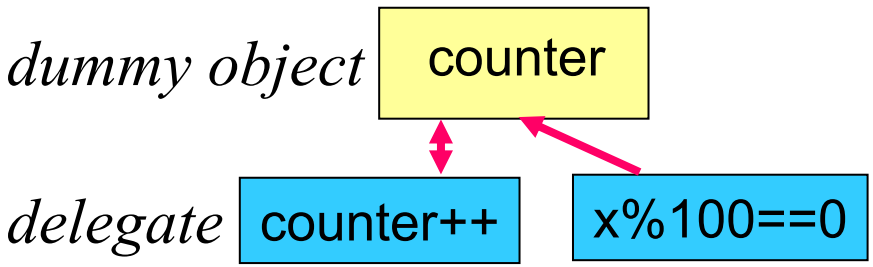
Delegate - možný seznam operací

```
static void Main(string[] args)
 {
   DoWithFile operace = new DoWithFile(Print);
   int count = 0;
   operace += delegate { count++; };
   ProcessDir(@"C:\Temp", operace);
 }
static void Print(string file) { Console.WriteLine(file); }
```

Dummy proměnné

```
static void Main(string[] args)
    { ProcessDir(@"C:\Temp", CreateDemo());
    }
static DoWithFile CreateDemo()
   {   int counter = 0;
       DoWithFile operace
             = new DoWithFile( delegate(string file)
                                    { Console.WriteLine(file); counter++;  });
       operace += delegate {
                        if (counter % 100==0) Console.WriteLine("/**100**/");
                                    };
       return operace;
    }
```

dummy – *n.* atrapa, maketa, *adj.* fiktivní



Proměnná counter *je sdílena všemi delegáty, kteří ji používají, a existuje do konce životnosti posledního z nich.*

Events - události

- *Speciální případy typu delegate. Jejich metody musí ale vracet void.*
- *Pokud se používají pro obsluhu událostí ve Windows, mají navíc povinné dva argumenty*
  - **object sender** – *odesílatel zprávy*
  - **EventArgs třídu** *nebo třídu od ní odvozenou, např.* AlarmEventArgs, MouseEventArgs, PaintEventArgs
- *Deklarace event vytváří člen třídy určený pro uložení odkazu odkaz na event-handler*

```
public delegate void MouseEventHandler (
    Object sender, MouseEventArgs e );

public event MouseEventHandler MouseMove;
```

Jméno členu typu event

Jaký je rozdíl mezi event a delegate?

- *delegate dovoluje operace* **= + = a - =**
- event *má jen operace* **+ = a - =**
- event *je systémově blízký property, pro tu překladač interně vygeneruje private datový člen typu delegate a skryje přitom operaci* **=**
- event *se může stát součástí deklarace interface (ta bude později), datový člen delegate nikoliv*
- public delegate se může volat odkudkoliv.
- public event lze evokovat jen z jeho deklaračního prostoru třídy/struktury.

Použití event

- event je cílený na GUI nebo jiné grafické aplikace, ač není apriori omezen jen tam.
  - Deklarací event dáváme najevo svůj úmysl – obsluha události.
- event omezuje manipulaci na přidání a ubrání metod, a proto z kódu odvozené třídy nelze zrušit již přidané private metody základní třídy, nutné pro správnou obsluhu událostí.
  - Event chrání funkčnost základní třídy.

Příklady na event uvedeme hlavně v následující přednášce o oknech

…KONEC…

nashledanou příště