Objekty

v jazyku C# a jejich využití

- Svět stále sestává z objektů,
- objekty konkrétní VS abstraktní,
- ponožka VS obraz (na slidu).
- Objekty patří do jednotlivých tříd, tedy
- uvedená jména objektů jsou názvy tříd.

Objekty, atributy, metody

- Objekty jsou nástupci struktur (recordů),
- struktury měly proměnné (prvky), u objektů tomu říkáme atributy,
- objekty mají i funkce zvané metody.
- V programu vytváříme různé objekty, nejen Program.

- Program žije v `namespacu`,
- v tomtéž namespacu definujeme pro použití další třídy podobně jako "hlavní" třídu.
- Definujeme ji úplně stejně (jen jí nevyrobíme statickou metodu Main).

Příklad

komplexní čísla

```
namespace Nic {
      class kompl {
          int re,im;
          public void nastav(int x,int y)
          {    re=x;
               im=y;
          }
      }
      class Program {
          static void Main(string[]x)
          {.........}
      } }
```

Instanciace

příklad, pořádně bude příště!

```
namespace Nic {
      ........
      class Program {
            public void Main(string[]x)
            {    kompl a=new kompl(),b=new kompl();
                 a.nastav(0,0);
                 b.nastav(10,5);
            }
      }
}
```

Komplexní čísla

další užitečné metody

```
class kompl {
      .........
      public void pricti(kompl co)
      {    re+=co.re;
           im+=co.im;
      }
}
```

Otázka: V jakém režimu je `re` a `im`? (public, private, protected...)

Komplexní čísla

předání parametru výsledkem

```
class kompl{
      ..............
      public void hodnota(out int x,out int y)
      {    x=re; y=im;}
}
```

- Co kdybychom předávali referencí?
- A neinicializovali proměnnou, protože ji funkce `hodnota` stejně hned přepíše?
- Pozor, modifikátory `out` a `ref` nutno uvést i při volání funkce (`x.hodnota(out a, out b);`)!
- Co nám připadá na příkladu jako největší opičárna?
- Nápověda: Že se nosič téže hodnoty chvíli jmenuje `re` a chvíli `x`. Jak se tomu ubránit?

Krok stranou

scope-resolution

- Hledáme-li proměnnou, překladač zkouší napřed parametry dotyčné funkce a lokální proměnné,
- pak atributy své třídy...
- Co když uděláme parametr jmenující se stejně jako atribut třídy?
- Dojde k zastínění a nedostali bychom se k atributu, proto objekt `this`.
- Lepší implementace:

```
public void hodnota(out int re,out int im)
{    re=this.re; im=this.im;}
```

Konstruktory

konstruují objekt

- Implementované přiřazování do čísel je nešikovné.
- Číslo chceme inicializovat typicky hned po vytvoření.
- K tomu můžeme použít konstruktor.
- Na konstruktor lze pohlížet jako na bezejmennou funkci vracející "svůj" typ.
- Nebo jako na funkci bez návratového typu jmenující se stejně jako dotyčná třída.
- Konstruktorů může být víc, musejí se lišit strukturou argumentů.
- Nedefinujeme-li žádný, vygeneruje se implicitní konstruktor (bez parametrů).
- Definujeme-li nějaký konstruktor, implicitní se negeneruje!

Příklad konstruktoru

```
class kompl {
      ..............
      public kompl (int re,int im)
      {     this.re=re; this.im=im;}
      public kompl()
      {     this.re=0; this.im=0;}
}
```

Není možné udělat konstruktor bez parametrů elegantněji?

Konstruktory

volání jiného konstruktoru

- Chceme-li zavolat jiný konstruktor, za parametry konstruktoru uvedeme operátor dvojtečky a řekneme, co se má zavolat:
- `public kompl():this(0,0){}`
- Ve složených závorkách můžeme definovat kód. Ten se provede až po dotyčném konstruktoru.
- Odbočka: Při dědičnosti lze použít objekt `base` pro rodiče. Funguje stejně jako `this`. Lze tedy podobně zavolat rodičovský konstruktor.

Dědičnost

- Nezřídka máme speciální případy obecné třídy a chceme, aby se chovaly uniformně.
- Například typ živá hmota má 7 metod (projevů života), každá živá hmota se ale chová úplně jinak.
- Chceme-li implementovat zvěřinec, hodilo by se implementovat typ zvíře a od něj odvodit typy jednotlivych živočišných druhů.
- Definovat "rodiče"je snadné, syna definujeme opět pomocí operátoru dvojtečky.

Příklad

rodič

```
class zvire {
      string jmeno;
      public void VydejZvuk()
      { Console.WriteLine("Nevydam, jsem zvire!");}
      public void NastavJmeno(string jmeno)
      { this.jmeno=jmeno;}
      public void KdoTam()
      { Console.WriteLine(jmeno);}
}
```

Příklad

syny

```
class tygr:zvire
{    public tygr(string jmeno)
     {    NastavJmeno(jmeno);}
     public void VydejZvuk()
     {    Console.Writeline("Vrrrrrr-rrum!");}
}
class slepice:zvire
{   public slepice() //slepice jmena nemaji
    { NastavJmeno("zadne"); }
    public void VydejZvuk()
    {    Console.WriteLine("Ko - ko - ko!");}
}
```

Použití dědičnosti

- Synovská třída zdědí všechno z rodičovské,
- pokud nějakou metodu předefinujeme, je předefinována,
- potřebujeme ale uniformní přístup ke všem synům (jinak by dědičnost byla k ničemu), proto
- do rodiče lze přiřadit syna:
- `zvire matysek=new tygr("Matysek");`
- Můžeme zavolat zděděné metody:
  `matysek.KdoTam();`
- Můžeme zkusit zavolat synovské metody (definované i v rodiči),
- ale spláčeme nad vejdělkem: `matysek.VydejZvuk(); =>`
  `"Nevydam, jsem zvire!''`
- Abychom mohli volat synovské metody, musí být metoda virtuální!