Objekty, přetížení, dědičnost

dědičnost

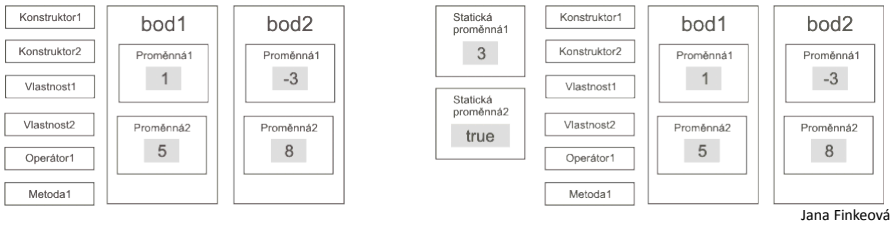
generalizace – specializace

Objekty

- **třída** (**class**) je předpis, pomocí něhož se tvoří **objekty** (**instance** dané třídy)
- každý objekt dané třídy má stejnou strukturu a chování, které je definováno ve třídě
- ještě jeden pohled:
  - třída je to, co navrhujeme a deklarujeme
  - objekt je to, co se z dané třídy vytváří za běhu aplikace
- objekty z jedné třídy mají sice stejnou strukturu a chování, ALE každý objekt má svůj vlastní **identifikátor** a svůj vlastní **stav**, tento stav je reprezentován množinou hodnot všech proměnných (atributů) daného objektu
- k identifikátoru se ze třídy přistoupí pomocí klíčového slova **this**

Atributy

- funkční členy jsou sdílené a datové složky, které reprezentují stav daného objektu, jsou specifické pro konkrétní instance, v terminologii OOP nazývají **atributy instance**
- mohou existovat také informace specifické pro danou třídu, které všechny instance dané třídy sdílejí, takové informace se obecně jmenují **atributy třídy** a v jazyce C# odpovídají **statickým proměnným**, které jsou deklarovány pomocí klíčového slova **static**
- statické proměnné představují data společná všem instancím



Jana Finkeová

Dědičnost

situace: potřebujeme novou třídu, která je však téměř stejná jako již existující třída, má pouze navíc jeden nebo několik datových a funkčních členů

- základní třída – předek / rodič
- odvozená třída – potomek

deklarace odvozené třídy:

```
class OdvozenaTrida : ZakladniTrida
 {// Implementace třídy
 }
```

jednoduchá dědičnost: každá třída může mít jen jednoho jediného předka

hierarchie dědění: lze vytvářet hierarchie odvozených tříd

```
class DalsiOdvozenaTrida : OdvozenaTrida
```

kořenovou třídou hierarchie všech tříd je tř. **`System.Object`**

```
(class ZakladniTrida : System.Object)
```

Jana Finkeová

Konstruktory

- při vytváření instance třídy se postupně volají konstruktory tříd v hierarchii dědičnosti od nejzákladnější třídy (System.Object) až po danou třídu
- konstruktor odvozené třídy volá konstruktor třídy základní
- toto volání se implementuje pomocí klíčového slova **base**

```
class OdvozenaTrida : ZakladniTrida
{// zápis konstruktoru, který volá konstruktor
 // bez parametrů implementovaný v základní třídě
   public OdvozenaTrida(): base()
   {
   }
// konstruktor s parametry
   public OdvozenaTrida(string nZT, double mOT) :
  base(nZT)
  {
    minB = mOT;
   }
```

Jana Finkeová

Dědičnost-metody

- vztah **generalizace-specializace**
- v odvozené třídě můžeme nejen **doplnit nové metody**
- můžeme i **přepsat metody** základní třídy, to znamená, změnit jejich definici (nové metody mají stejnou hlavičku jako metody předka)
  - např. tehdy je-li metoda v základní třídě (předkovi) definována jako **virtuální** (pomocí klíčového slova **virtual**), což znamená, že může být v odvozených třídách přepsána
  - v odvozené třídě (potomkovi) je přepsána pomocí klíčového slova **override**, čímž je pozměněna definice této metody
  - POZOR: nelze použít pro soukromé metody
- jestliže i v přepsané metodě potřebujeme zavolat metodu základní třídy, pak použijeme klíčové slovo **base**

```
public NazevMetodyVOdvozeneTride ()
  {  base.NazevMetodyZeZakladniTridy();}
```

- můžeme **přetěžovat metody** – v tomto případě není hlavička metody potomka stejná jako hlavička metody předka

Jana Finkeová

Dědičnost

- modifikátory přístupu
  - **public** - **veřejný člen** je přístupný uvnitř i vně třídy (obecně typu)
  - **private** - **soukromý člen** je přístupný pouze uvnitř třídy (obecně typu)
  - **protected** - **chráněný člen** je přístupný pouze uvnitř třídy (obecně typu) a třídách odvozených z této třídy

Přetěžování

- řada úkonů z našeho života má stejný název, avšak vlastní provedení záleží na předmětu (předmětech), se kterým(i) se daný úkon provádí
- konkrétní úkon však provedeme až po zjištění, s jakým předmětem jej máme provést
- funkční člen, jenž má více definic (verzí), které závisí na signatuře (tj. počtu parametrů a jejich datových typech), se nazývá **přetížený** (**overloaded**)
- **volání přetíženého funkčního členu tedy provede tu verzi, která souhlasí signaturou se seznamem skutečných parametrů (co do jejich počtu i datových typů)**
- přetěžování metod
- přetěžování konstruktoru
- přetěžování operátoru

Přetěžování operátoru

- pomocí statické metody nazvané **operator+**,

```
public struct KomplexniCislo
  {
        public double real;
        public double imag;

       public KomplexniCislo(double a, double b)
         {
        this.real = a;
        this.imag = b;
        }
       public static KomplexniCislo
  operator+(KomplexniCislo cislo1,KomplexniCislo cislo2)
        {
        return new KomplexniCislo(cislo1.real+
  cislo2.real, cislo1.imag + cislo2.imag);
        }
   }
```

Jana Finkeová

Přetěžování operátoru

```
KomplexniCislo z1 = new KomplexniCislo(-5,6);

KomplexniCislo z2 = new KomplexniCislo(-3,7);

// Součet komplexních čísel řešený pomocí
// přetíženého operátoru + je intuitivní:
KomplexniCislo z3 = z1 + z2;
```

Jana Finkeová