Prednáška 3: Polymorfizmus, rozhrania a vhniezdené typy

Objektovo-orientované programovanie 2012/13

Valentino Vranić

Ústav informatiky a softvérového inžinierstva
Fakulta informatiky a informačných technológií
Slovenská technická univerzita v Bratislave

6. marec 2013

Obsah prednášky

Polymorfizmus

Dedenie a typy

- Trieda definuje typ
- Typ podtriedy je podtypom nadtriedy
- *Upcasting*: implicitná zmena typu referencie na objekt z typu podtriedy na typ nadtriedy
  - *up* – hore: smerom k nadtriede, ktorá je v hierarchii tried postavená *vyššie*
- Príklad: metóda bude akceptovať objekt podtriedy

```
void zarad(Utvar u) {. . .}
. . .

Kruh k = new Kruh();
zarad(k);
```

- Ide o prejav *polymorfizmu*

Polymorfizmus

- Objekt sa môže správať akoby bol objektom hociktorého zo svojich nadtypov
- Každý objekt však „pozná“ svoj skutočný typ
- Výber tela metódy sa uskutoční až v čase vykonávania programu
  - okrem pri finálnych metódach, pri ktorých evidentne nemôže dôjsť k prekonávaniu (**private** metódy sú implicitne finálne)
  - *late binding* – neskoré viazanie
- Pozor pri zdanlivom prekonávaní **private** metód (TiJ, Ch. 7, `PrivateOverride.java`)

Príklad: grafické útvary

```
public class Utvar {
    ...
    public void nakresli() { }
}

public class Kruh extends Utvar {
    ...
    public void nakresli() { System.out.print("Kruh "); }
}

public class Trojuholnik extends Utvar {
    ...
    public void nakresli() { System.out.print("Trojuholnik "); }
}
```

- Použitie:

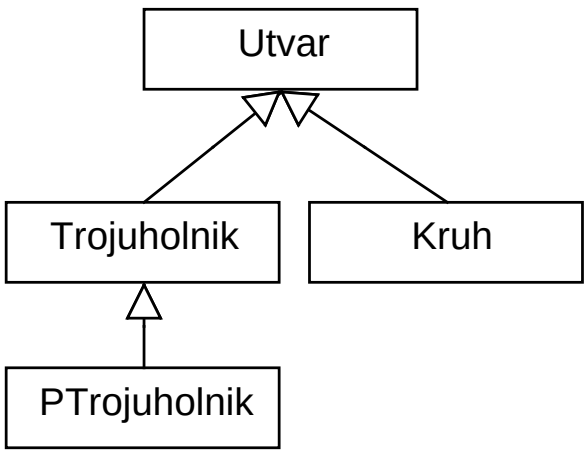
```
Utvar[] z = new Utvar[] { new Kruh(a, 3.0),
                          new Trojuholnik(a, b, c),
                          new Kruh(b, 1.0) };
for (int i = 0; i < z.length; i++)
    z[i].nakresli(); // "Kruh Trojuholnik Kruh"
```

Ďalšie útvary

- Možno pridať ďalšie druhy útvarov
- Možno pridať odvodené druhy útvarov – napr. pravouhlý trojuholník:

```
public class PTrojuholnik extends Trojuholnik {
    ...
    public void nakresli() {
        System.out.print("PTrojuholnik ");
    }
}
```

- Hierarchia útvarov v UML:



Flexibilita polymorfizmu

- Polymorfizmus bude fungovať aj po pridaní ďalších tried do hierarchie:

```
Utvar[] z = new Utvar[] { new Kruh(a, 3.0),
                          new Trojuholnik(a, b, c),
                          new PTrojuholnik(a, b, c) };

for (int i = 0; i < z.length; i++)
    z[i].nakresli(); // "Kruh Trojuholnik PTrojuholnik"
```

Abstraktné triedy a metódy

- Abstrakcia – uberanie detailov
  - abstrahovať od niečoho = vynechať niečo
  - lat. *abs* (od) + *trahere* (ťahať) – vyťahovať (traktor)
  - opačný proces: konkretizácia
- Abstraktná trieda nemôže mať inštancie – určené sú len na dedenie
- Abstraktné metódy nemajú telo – určené sú len na prekonávanie
  - Jedine abstraktná trieda môže (ale nemusí) obsahovať abstraktné metódy

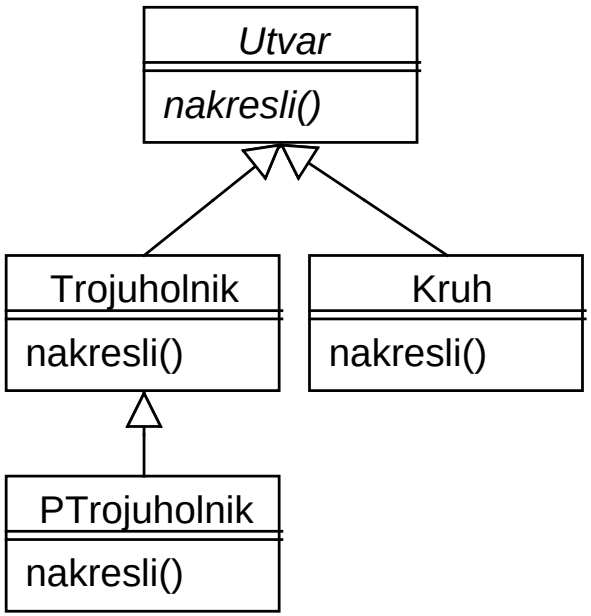
Príklad: útvar ako abstraktná trieda (1)

- Aby sa zabezpečilo rozpoznanie volania `nakresli()` v podtriedach, trieda `Utvar` musí obsahovať metódu `nakresli()`
- Nie je však žiaduce, aby metódu `nakresli()` v triede `Utvar` niekto zavolal
- Zabezpečíme to deklarovaním tejto metódy ako abstraktnej

```
abstract class Utvar {
    ...
    abstract void nakresli();
}
```

Príklad: útvar ako abstraktná trieda (2)

- Abstraktné prvky sa v UML označujú kurzívou



- Diagram nemusí obsahovať všetky detaily kódu – znázorníme to na čo chceme poukázať

Polymorfizmus a statické metódy

- Pri statických metódach nedochádza k prekonávaniu – len k skrývaniu

```
class A {
    static void sf() { System.out.println("A"); }
}

class B extends A {
    static void sf() { System.out.println("B"); }
}
```

- Výber statickej metódy sa uskutočňuje už pri preklade na základe typu referencie

```
A a = new B();
a.sf(); // A
```

Downcasting

- Čo keď bol vykonaný upcasting a potrebujeme zavolať metódu podtriedy?
- Urobíme *downcasting*
- Využíva sa pritom *run-time type identification* (RTTI)
- Downcasting nie vždy končí úspešne (na rozdiel od upcastingu)

Príklad downcastingu

```
class A {
    public void f() {}
}

class B extends A {
    public void f() {}
    public void g() {}
}
```

- Použitie:

```
A o1 = new A();
A o2 = new B(); // upcasting
// o1.g(); // chyba pri preklade
// o2.g(); // chyba pri preklade
((B)o2).g() // downcasting – OK
((B)o1).g() // downcasting – chyba pri vykonávaní
```

Rozhrania

Rozhrania – dedenie správania

- Rozhranie (**interface**) umožňuje definovať správanie bez implementácie

```
interface Kresleny {
    void nakresli();
    void nakresli(int f);
}
```

- Dedenie rozhrania – trieda implementuje rozhranie

```
class Kruh implements Kresleny {
    ...
}
```

Príklad: kruh ako kreslený útvar

```
abstract class Utvar {
  int farba;
}

class Kruh extends Utvar implements Kresleny {
   ...
   public void nakresli() {
      System.out.println("Kruh (" + c.x + ", " + c.y + ") r = " + r);
   }
   public void nakresli(int f) {
      System.out.println("Kruh (" + c.x + ", " + c.y + ") r = " + r + "farba " + f);
   }
   ...
}
```

- Použitie:

```
Kresleny k = new Kruh();
```

Rozhrania a abstraktné triedy

- Všetky triedy odvodené od triedy, ktorá implementuje nejaké rozhranie tiež dedia toto rozhranie
- Ak chceme dosiahnuť, že každý útvar musí implementovať metódy na kreslenie, stači to uviesť pri triede `Utvar`:

```
abstract class Utvar implements Kresleny {
    private int farba;
    ...
}
```

Vlastnosti rozhraní

- Konkrétne triedy musia implementovať všetky metódy rozhrania, ale abstraktné triedy nemusia
- Všetky metódy rozhrania sú abstraktné a majú prístup `public`
  - Aj všetky implementácie týchto metód v triedach musia byť **public**
- Všetky atribúty rozhrania sú **static** a **final**
- Trieda môže implementovať viac rozhraní
  - Môžu vznikať kolízie názvov metód

Príklad: viac rozhraní jednej triedy (1)

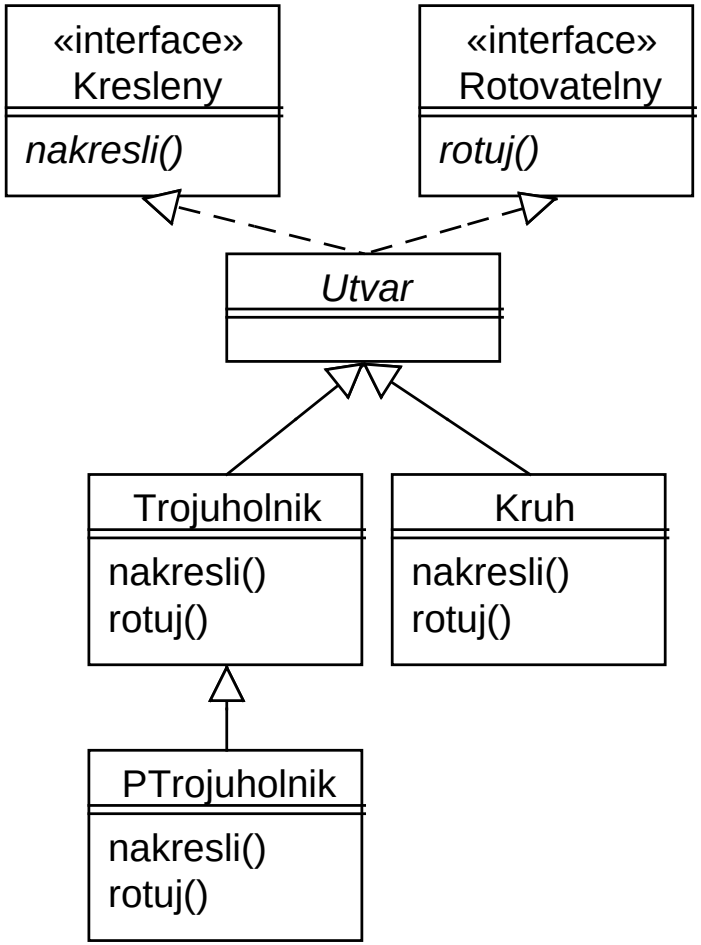
```
interface Kresleny {
    void nakresli();
    void nakresli(int f);
}

interface Rotovatelny {
    void rotuj(double uhol);
}

abstract class Utvar implements Kresleny, Rotovatelny {
    private int farba;
    ...
}
```

Príklad: viac rozhraní jednej triedy (2)

- Situácia sa komplikuje – vhodné je nakresliť diagram



Dedenie medzi rozhraniami

- Rozhranie môže dediť od iného rozhrania pomocou klauzuly `extends`
- Takéto dedenie môže byť aj od viacerých rozhraní

```
interface M {
    void m1();
    void m2();
}
interface N {
    void n();
}
interface I extends M, N {
    void i();
    void n(int i); // preťažená metóda
}
```

Vhniezdené typy

Vhniezdené typy

- Termín *typ* budeme používať ako spoločné označenie pre triedy a rozhrania
- Vhniezdené typy (nested types): typy deklarované v inej triede alebo rozhraní, vrátane metód a inicializačných blokov príkazov

```
class A {
    interface I { }
    class B { }
}

interface J {
    class X {
        class Y {
        }
    }
}
```

Preklad vhniezdených typov

- Prekladom vznikajú súbory s príponou **class** ako aj pri typoch na vrchnej úrovni
- Názov takého súboru obsahuje zreťazené názvy typov, v ktorých je tento typ vnorený:

```
class A {
    class B {
      class C { }
    }
}
```

```
A.class
A$B.class
A$B$C.class
```

Vhniezdené rozhrania

- Len v statickom kontexte

```
interface M {
    interface N {
        void n();
    }
    void m();
}

class C implements M, M.N {
    public void n() { . . . };
    public void m() { . . . };
}
```

- Rozsiahlejší príklad v TiJ (Ch. 8, *Interfaces*, *Nesting interfaces*)

Vnútorné triedy

- Vhniezdené triedy, ktoré nie sú explicitne alebo implicitne deklarované ako statické, sa označujú ako vnútorné triedy (*inner classes*):[1]
  - nestatické členské triedy
  - lokálne triedy (v hocijakom bloku vrátane tiel podmienených príkazov a cyklov)
  - anonymné triedy
- Termín *vhniezdené triedy* v užšom zmysle označuje statické členské triedy (použité v TiJ)

[1] J. Gosling, B. Joy, G. Steele, and G. Bracha. The Java Language Specification, 2nd edition. Addison-Wesley, 2000. `http://java.sun.com/docs/books/jls/`

Príklad: vnútorné triedy

```
class A {
    class B { } // trieda v triede

    void f(int i) {
        class C { } // trieda v metode
        if(i > 0) {
            class D { // trieda v podmienenom prikaze
                C c = new C();
            }
        }
    }
// D d; // D neznáme -- mimo rozsahu!
    void m() {
        B b = new B(); // objekt vnútornej triedy
    // C c; // C neznáme - mimo rozsahu!
    }
}
```

Zmysel vnútorných tried

- Vnútorné triedy prenášajú OO prostriedky do nižších úrovní programu
- Umožňujú viacnásobné dedenie (*multiple inheritance*) štruktúry (implementácie)
- Predstavujú veľmi flexibilný prostriedok často používaný v Java API, napríklad v grafickom rámci Swing

Vlastnosti vnútorných tried

- Vnútorná trieda je súčasťou triedy, v ktorej je deklarovaná – všetky časti vonkajšej triedy sú dostupné vnútornej triede
- Inštancie vnútornej triedy sa nedajú vytvárať bez vytvorenia objektu vonkajšej triedy
  - Na objekt vonkajšej triedy sa môže vzťahovať viac objektov tej istej vnútornej triedy
  - Ale nemusí ani jeden
- Vnútorná trieda nemôže obsahovať statické prvky
- Lokálna trieda je viditeľná len v rámci bloku, v ktorom bola deklarovaná

Príklad: viditeľnosť vnútorných tried

```
class A {
    private int i = 0;
    class B {
        void m() {
            i = 1;
        }
    }
    private class P { }
}

class C {
    void f() {
// A.P p; // private!
      A.B x = (new A()).new B();
      x.m();
    }
}
```

Preklad lokálnych tried

- Java nepodporuje podmienený preklad ako v jazyku C
- Deklarácia lokálnej triedy v podmienenom bloku neznamená, že jej **class** súbor vznikne podmienečne
- Príklad:

```
class A {
    void f(int i) {
        if(i > 0) {
            class D { }
        }
    }
}
```

- Prekladom vzniknú súbory:

  `A.class`

  `A$1D.class`

Objekty vnútorných tried

- Objekty vnútornej triedy môžu byť použité aj mimo triedy alebo bloku, v ktorom vnútorná trieda bola deklarovaná
- Ale deklarácia vnútorných tried okrem členských tried s iným prístupom než je **private** nedostupná
- To znamená, že je autohtónne rozhranie objektu vnútornej triedy neznáme (okrem rozhrania triedy `Object`)
- Preto vnútorná trieda najčastejšie implementuje verejne dostupné rozhranie

Príklad: vnútorná trieda a rozhranie

```
interface I {
    void m();
}

class A {
    I f() { // metoda f() nemoze mat navratovu hodnotu typu B!
        class B implements I {
            public void m() { . . . }
        }
        return new B();
    }
}

class C {
    void f() {
        I b = (new A()).f();
        b.m();
    }
}
```

Anonymné triedy

- Priamo pri inštanciácii triedy možno deklarovať anonymnú triedu (*anonymous class*)

```
new [Typ]([parametre]) {
  [telo]
};
```

  - [Typ] – názov triedy alebo rozhrania, od ktorého anonymná trieda dedí (**extends**, resp. **implements**)
  - [parametre] – parametre konštruktora triedy, od ktorej anonymná trieda dedí
  - [telo] – atribúty a metódy anonymnej triedy
- Pri takejto inštanciácii vzniká objekt anonymnej triedy
- Ale už nikdy nebude môcť vzniknúť ďalší – lebo táto trieda nemá meno

Príklad anonymných tried

```
class A { . . . }

class C {
  Object o = new Object() {
      float x;
      void f() { . . . }
  };
  void f() {
      A o = new A() {
          int a;
      };
  }
}
```

- Tieto dve anonymné triedy sú málo použiteľné – nepoznáme ich rozhranie

Anonymné triedy a implementácia rozhrania

- Rozhranie sprístupni metódy objektu anonymnej triedy

```
interface I {
    void m();
}
class A {
    I f() {
        class B implements I {
            public void m() { . . . }
        }
        return new B();
    }
    I f2() { // ako f(), ale s anonymnou triedou
        return new I() {
            public void m() { . . . }
        };
    }
}
```

Vnútorné triedy a parametre metód

- Lokálne a anonymné triedy môžu pristupovať len k finálnym parametrom metódy, v ktorej sa nachádzajú

```
interface I {
  float m();
}
class A {
    void f(final float n) {
        class B implements I {
            public float m() {
                return n * n;
            }
        }
    }
}
```

Statické vhniezdené triedy

- Statické vhniezdené triedy môžu byť súčasťou rozhraní
- Vhodné na ladenie jednotlivých tried:
  - Do každej triedy pridať statickú vhniezdenú triedu s metódou `main()`
  - Z finálneho programu možno odstrániť **class** súbory, ktoré vzniknú prekladom týchto tried – žiaden kód navyše

Statické vhniezdené triedy (2)

- Ladenie pomocou statických vhniezdených tried:

```
class T {
    int f(int i) {
        ...
    }
    static class Test {
        public static void main(String[] args) {
            T t = new T();
            System.out.println(t.f(0));
            System.out.println(t.f(-1));
            ...
        }
    }
}
```

Viacnásobné dedenie štruktúry

- C++ priamo podporuje viacnásobné dedenie
  - Viacnásobné dedenie aj štruktúry (implementácie), aj správania (typu)
  - Ako keby v Jave bolo možné za **extends** uviesť viac tried
- Viacnásobnému dedeniu štruktúry sa často dá vyhnúť zmenou štruktúry tried a použitím rozhraní
- Niekedy to nejde a niekedy to ani nie je vhodné...
- Viacnásobné dedenie štruktúry pomocou vnútorných tried
  - TiJ, Chapter 8, *Inner classes & control frameworks* (len odporúčané čítanie)

Sumarizácia

Sumarizácia

- Polymorfizmus – kľúčový mechanizmus OOP
- Prekonávanie metód – nie pri statických metódach
- Abstraktné triedy a rozhrania obsahujú abstraktné metódy
- Downcasting
- Typy (triedy a rozhrania) môžu byť vhniezdené:
  - vhniezdené rozhrania a statické vhniezdené triedy
  - vnútorné triedy – môžu byť lokálne (v metódach) a dokonca aj anonymné (bez mena)

Čítanie

- Dnešná prednáška: OJA, kapitoly 5 a 7
- Na ďalšej prednáške sa vrátime k princípom OO programovania
- Z dostupných materiálov môžete pozrieť (na ďalšiu prednášku):
  - R. C. Martin. The Liskov Substitution Principle. C++ Report, 1996. `http://www.objectmentor.com/resources/articles/lsp.pdf`
  - R. C. Martin. The Open-Closed Principle. C++ Report, 1996. `http://www.objectmentor.com/resources/articles/ocp.pdf`
  - The Ellipse-Circle Dilemma. `http://ootips.org/ellipse-circle.html`