Elementárne programovanie v Jave

Objektovo-orientované programovanie 2012/13

Valentino Vranić

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

27. február 2013

Obsah prednášky

Preťaženie metód

Preťaženie metód (1)

- Overloading
- Dve metódy tej istej triedy môžu niesť rovnaký názov, ak sa líšia v zozname parametrov

```
class Student {
    String meno;
    boolean zapisany;
    int rocnik;

    void nastavenie(String m) {
        meno = m;
    }
    void nastavenie(String m, boolean z) {
        meno = m;
        zapisany = z;
    }
}
```

Preťaženie metód (2)

- Použitie preťažených metód:

```
Student a = new Student();
Student b = new Student();
a.nastavenie("Jozef Kohut", true);
b.nastavenie("Jana Petrova");
```

- Presnejšie povedané preťažené sú *názvy metód*[1]
- Analógia v prirodzených jazykoch: homonymá
- V širšom zmysle preťaženie je druh polymorfizmu[2]

---

[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/

[2] Krzysztof Czarnecki. Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models. Ph.D. Thesis, Computer Science Department, Technical University of Ilmenau, Ilmanau, Germany, 1998. http://www.issi.uned.es/doctorado/generative/Bibliografia/TesisCzarnecki.pdf

Preťaženie metód (3)

- Návratová hodnota sa nedá použiť na rozlíšenie medzi preťaženými metódami
- Metóda println() ako príklad preťaženia
  - vypíše čokoľvek
  - dosiahnuté pomocou preťaženia
  - dokumentácia J2SE API:

```
println()
println(boolean)
println(char)
...
println(java.lang.Object)
println(java.lang.String)
```

Promócia typov

- Typy môžu byť automaticky zmenené na „vyššie“
- V Jave dochádza k tzv. promócii primitívnych typov:
  - výsledok matematickej operácie alebo operácií na bitoch je aspoň **int** alebo najväčšieho zo zúčastnených typov
  - pri priradení do pôvodného typu, treba explicitne zmeniť typ (*casting*)

```
byte b1 = 1, b2 = 2;
b1 = b1 + b2; // chyba pri preklade!
b1 = (byte)(b1 + b2); // OK
```

  - pri použití skráteného zápisu op=, zmena je implicitná

```
b1 += b2; // OK
```

- Promócia typov platí aj pri volaní metód

Preťaženie a promócia typov (1)

- Pri preťažených metódach sa vyberie metóda, ktorej veľkosť typu formálneho parametra je najbližšia skutočnému
- Ak je len jeden parameter, situácia je jasná
  - Príklad v TiJ (*Overloading with primitives*)
- Čo keď je parametrov viac?
  - Zodpovedajúca metóda sa nájde len ak je priradenie formálnych k skutočným parametrom jednoznačné

Preťaženie a promócia typov (1)

```
class Pretazenie {
    static void m(int i, long l) { System.out.println("A"); }
    static void m(long l, int i) { System.out.println("B"); }

    public static void main(String[] args) {
        byte b = 8; short s = 16; int i = 32; long l = 64;

        m(b, l); m(s, l); m(i, l); // "A"
        m(l, b); m(l, s); m(l, i); // "B"
    // m(b, b); m(b, s); m(b, i); // ambiguous reference
    // m(s, b); m(s, s); m(s, i); // ambiguous reference
    // m(i, b); m(i, s); m(i, i); // ambiguous reference
    // m(l, l); // cannot resolve
    }
}
```

Inicializácia, konštruktory a finalizácia

Inicializácia

- Správna inicializácia je v programoch nevyhnutná
- Časť inicializácie sa odohráva prostredníctvom tzv. *konštruktorov* pri vytváraní objektov
- Ale možné sú aj ďalšie spôsoby inicializácie

Vynechanie inicializácie

- Nechcené vynechanie inicializácie sa často ťažko odhaľuje (napr. v jazyku C)
- Java poskytuje mechanizmy na predchádzanie tomuto problému
- Premenné v metódach musia byť explicitne inicializované

```
void f() {
    int i;
    i++; // chyba pri preklade!
}
```

Inicializácia atribútov tried

- Atribúty tried sa inicializujú priamo pri ich definícii

```
class CInit {
    int i = 1;
    int j = 2;
}
```

- Ak sa neinicializujú, nastavia sa na implicitné hodnoty
- Inicializácia statických atribútov prebehne pri načítaní triedy, inak pri vytvorení objektu

Inicializácia atribútov tried (2)

- Inicializácia nemusí byť vykonaná konštantou

```
class CInit {
    int i = f();
    int j = g(i);
    ...
}
```

- Poradie je významné

```
class CInit {
    int j = g(i); // chyba pri preklade!
    int i = f();
    ...
}
```

Konštruktory (1)

- Konštruktor je špeciálna operácia na *inicializáciu* objektu volaná automaticky pri jeho vytváraní
- Konštruktor nesie názov triedy

```
String s = new String("asdf");
```

- Ak programátor konštruktor neposkytne, prekladač poskytne implicitný (prázdny) konštruktor

```
class Student {
    String meno;
    boolean zapisany;
    int rocnik;
}
```

- Taký konštruktor možno použiť rovnako, ako keby bol súčasťou triedy

```
Student s = new Student();
```

Konštruktory (2)

- Poznámka k použitiu konštruktora triedy String:[3]
  - Konštruktor vytvára vždy nový objekt
  - Vytvorenie reťazca znakov priradením prináša optimalizáciu: rovnaké reťazce sú zdieľané

```
String x = new String("a");
String y = new String("a");
String w = "a";
String z = "a";
System.out.println(x == y); // false
System.out.println(w == z); // true
```

[3] J. Gosling et al. The Java Language Specification, Third Edition. Addison-Wesley, 2005. – Section 3.10.5 (citované v: J. Bloch. Effective Java, 2nd edition. Prentice Hall, 2008. – Item 5)

Konštruktory (3)

- Konštruktor nemôže mať návratovú hodnotu
- Konštruktor môže mať parametre:

```
class Student {
   ...
   public Student(int r) {
      rocnik = r;
   }
}
```

- Ak sa takto poskytne hocijaký konštruktor (aj keď má parametre), implicitný konštruktor sa nezachová:

```
Student s = new Student(); // chyba pri preklade!
Student s = new Student(1); // "Student 1. rocnika"
```

Preťaženie konštruktorov (1)

- Konštruktory môžu byť preťažené, čo umožňuje poskytnúť viac koštruktorov jednej triedy
- Napr. implicitný konštruktor triedy `Student` možno zachovať:

```
class Student {
    String meno;
    int rocnik;
    public Student() {}
    public Student(int r) {
        rocnik = r;
    }
}
```

Preťaženie konštruktorov (2)

- Konštruktor sa nedá volať priamo – okrem v iných konštruktoroch tej istej triedy
  - pomocou kľúčového slova **this**
  - dá sa volať len jeden ďalší konštruktor a len na začiatku

```
class Student {
    private String meno;
    private int rocnik;
    public Student() {
        this(1); // predpokladá sa prvý ročník
    }
    public Student(int r) {
        rocnik = r;
    }
}
```

Kľúčové slovo **this**

- Referencia na aktuálny objekt
- Použitie:
  - vrátenie referencie na aktuálny objekt

```
class C {
    ...
    C m() {
        if (...)
            return new C();
        else
            return this;
    }
}
```

  - volanie konštruktora v inom konštruktore
  - rozlíšenie medzi formálnym parametrom a atribútom objektu

```
public Student(int rocnik) {
    this.rocnik = rocnik;
}
```

- V statických metódach **this** nie je

Inicializácia atribútov tried v konštruktore

- . . . vlastne ani nie je inicializácia, lebo inicializácia (aspoň implicitnými hodnotami) už prebehla
- Väčšia flexibilita pri výbere iniciálnej hodnoty pre daný objekt
  - vzhľadom na parametre konštruktora
  - vzhľadom na iné podmienky (aj keď sa aj pri inicializácii atribútov dá použiť ternárny **if-else** operátor)
  - možnosť použitia slučiek (vhodné pre polia)

Inicializácia blokom príkazov

- Inicializácia blokom príkazov prebieha tiež až po skutočnej inicializácii
- Blok príkazov v triede

```
class C {
   {
      System.out.println("Novy objekt");
   }
}
```

- Umožňuje definovať veci, ktoré sa majú vykonať pri každom vytvorení nového objektu

```
new C(); // "Novy objekt"
new C(); // "Novy objekt"
```

Statická inicializácia blokom príkazov

- Bloku príkazov v triede môže predchádzať kľúčové slovo **static**

```
class C {
    static {
        System.out.println("Trieda nacitana");
    }
}
```

- Umožňuje definovať veci, ktoré sa majú vykonať pred prvým použitím triedy (pri jej načítaní)

```
new C(); // "Trieda nacitana"
new C();
```

Finalizácia

- Proces definovaný pre objekty danej triedy, ktorý sa uskutoční pri uvoľňovaní pamäte (*garbage collection*)
  - Ak k tomu vôbec dôjde!
- Poskytuje sa ako špeciálna metóda **public void** `finalize()`
- Finalizáciu asi nebudete potrebovať
- Možno ju využiť pre uvoľnenie pamäte obsadenej inak ako pomocou **new**
  - Java umožňuje volať kód v C a C++, takže `malloc()`...
- Nebudeme sa zaoberať podrobnosťami garbage collection

Polia

Polia v Jave

„Keď sú slová nevhodné, reč je neprispôsobená a činy sú neúspešné.“
– Konfucius

- Terminologická poznámka:
  - *field* – atribút; v Jave *pole* (priestor, slot) v triede
  - *array* – pole – rad prvkov elementárneho typu alebo objektov; lepší termín by bol *reťazec*
  - *string* – reťazec znakov; v Jave trieda `String`
- V Jave je pole objekt – vytvára sa dynamicky
- Referencia a (presnejšie povedané, *premenná*, ktorá obsahuje referenciu) na pole celých čísel:

  ```
  int[] a; // alebo int a[];
  ```

- Vytvorenie poľa a celých čísel dĺžky 5

  ```
  a = new int[5];
  ```

- Inicializácia polí je implicitná

Explicitná inicializácia polí

- Pole celých čísel inicializované explicitne:

  int[] c1 = **new int**[] { 1, 2, 3 };

- Skrátená forma:

  int[] c2 = { 1, 2, 3 };

- Rovnako sa postupuje aj pri polí referencií:

```
Student[] r1 = new Student[] { new Student(), new Student(), };
Student[] r2 = { new Student(), new Student(), };
```

  - Čiarka na konci nemusí byť, ale nie je ani chybou

Veľkosť poľa

- Veľkosť poľa musí byť daná
- Ale nemusí byť známa v čase prekladu – korektné je napríklad toto:

  `a` = **new int**[`C.f`(...)];

- Pritom `f()` je metóda:

```
class C {
    public static int f(...) { . . . }
    . . .
}
```

- Veľkosť poľa sa uchováva v atribúte `length`
  - pre uvedené pole `a` je to `a.length`

Prístup k prvkom poľa

- K prvkom poľa sa pristupuje prostredníctvom indexu

```
for (int i = 0; i < a.length; i++)
    System.out.println(a[i]);
```

- Index prvého prvku je 0
- Nedá sa siahnuť mimo pamäte rezervovanej pre pole

```
a[a.length]; // chyba pri vykonávaní programu
```

Polia ako parametre metód

- Polia sa prenášajú referenciou – ako hociktorý iný objekt
- Použiteľné na simuláciu metód s premenlivým počtom parametrov

```
class Zapis {
    ...
    static void zapisStudentov(int r, Student[] s) {
        for (int i = 0; i < s.length; i++) {
            s[i].zapisany = true;
            s[i].rocnik = r;
        }
    }
}
```

- Zápis viacerých študentov naraz (napr. zo zoznamu):

```
Zapis.zapisStudentov(1, new Student[] { s1, s2, s3, s4 });
```

Metódy s premenlivým počtom parametrov (1)

- Od Javy 5 možno použiť metódy s premenlivým počtom parametrov (vararg/variadic[4] function/method)
- Podobnú vlastnosť podporuje jazyk C, ale jej použitie je zložité
- V Jave sa posledný parameter označený tromi bodkami jednoducho správa ako pole

```
class Zapis {
    ...
    static void zapisStudentov(int r, Student... s) {
        for (int i = 0; i < s.length; i++) {
            s[i].zapisany = true;
            s[i].rocnik = r;
        }
    }
}
```

[4] http://en.wikipedia.org/wiki/Variadic_function

Metódy s premenlivým počtom parametrov (2)

- Pri volaní už nemusíme vytvárať pole – ale môžeme:[5]

```
Zapis.zapisStudentov(2, s1, s2, s3, s4);
Zapis.zapisStudentov(2, new Student[] {s1, s2, s3, s4});
```

- Takýto zápis možno používať všade, kde aj starý (syntactic sugar):

```
public static void main(String... args)
```

- Zrovna pri metóde `main()` to nie je obvyklé

[5] http://www.agiledeveloper.com/articles/Java5FeaturesPartII.pdf

Viacrozmerné polia

- Podobne ako v jazyku C
- Májme takéto pole:

```
char[][] abc = {
        { 'a', 'b', 'c', },
        { 'x', 'y', 'z', },
};
```

- Po riadkoch ho vypíšeme takto:

```
for (int i = 0; i < abc.length; i++) {
    for (int j = 0; j < abc[i].length; j++)
        System.out.print(abc[i][j] + " ");
    System.out.println();
}
```

Podpora práce s poliami

- Java API poskytuje podporu práce s poliami prostredníctvom statických metód tried `Array` a `Arrays`
- `Array` poskytuje podporu pre dynamické vytváranie a prístup k poliam
- `Arrays` obsahuje rady preťažených metód pre prácu s poliami:
  - `equals()` – porovnávanie polí
  - `fill()` – naplnenie poľa zadanou hodnotou
  - `sort()` – triedenie prvkov poľa
  - `binarySearch()` – binárne vyhľadávanie v utriedenom poli
- Trieda `System` poskytuje statickú metódu `arraycopy()`, ktorá slúži na kopírovanie polí

Reťazce znakov

- Reťazce znakov sú v Jave objekty – môžu byť dvoch typov
  - `String` – konštantné reťazce znakov
  - `StringBuffer` – premenlivé reťazce znakov
- Java API poskytuje rozsiahlu podporu práce s reťazcami znakov: vytváranie, kopírovanie, spájanie, vyhľadávanie podreťazcov, nahrádzanie podreťazcov a pod.

Balíky a riadenie prístupu

Zdrojové súbory a ich preklad v Jave

- Každý zdrojový súbor (`.java`) obsahuje jednu alebo viac tried
- Každý zdrojový súbor sa prekladá zvlášť
  - pri preklade je potrebný bajtkód tried, ktoré sa v ňom používajú
  - javac ho vytvorí zo zdrojových súborov ak sú dostupné
- Pri preklade pre každú triedu vznikne súbor s bajtkódom (`.class`)
- Ak trieda obsahuje metódu `main()`, takýto súbor sa dá spustiť
  - tzn. že program môže mať viac vstupných bodov

Organizácia zdrojových súborov v Jave

- Triedy, ktoré súvisia, možno zoskupiť do balíka (*package*)

  package `nazov_balika`;

  - Ak je uvedená, deklarácia balíka musí byť prvým príkazom
  - Všetky triedy mimo explicitne definovaných balíkov patria do jedného, implicitného balíka
- Balíky sú organizované hierarchicky
  - umiestnenie balíka v hierarchii sa vyznačuje bodkami:

    package `nadnadbalik.nadbalik.balik`;
- Konvencia pre pomenúvanie balíkov – vysvetlená v TiJ

Organizácia preložených súborov v Jave

- Preložené súbory musia byť zaradené do adresárovej štruktúry, ktorá zodpovedá hierarchii balíka
- Súbory balíka

  package `nadnadbalik.nadbalik.balik`;

  musia byť v adresári `nadnadbalik/nadbalik/balik` (inak sa nebudú dať spustiť)
- Začiatok cesty je daný premenou prostredia **classpath**
  - obsahuje všetky adresáre, v ktorých JVM má hľadať triedy
  - definuje sa ako premenná prostredia (rovnako ako premenná **path**)
  - alebo sa zadáva priamo pri spúšťaní (za prepínačom **-classpath**)

Zavedenie balíka

- Zavedením balíka sa prekladaču len sprístupní priestor názvov

```
import nadnadbalik.nadbalik.balik.*;
```

- Inak sa musia používať plne vymedzené názvy

```
nadnadbalik.nadbalik.balik.Student s =
    new nadnadbalik.nadbalik.balik.Student();
```

- Plne vymedzené názvy sa musia používať pri kolíziách názvov (prekladač hlási chybu)

Zavedenie statického prvku

- Od Javy 5 prekladaču sa môže sprístupniť statický prvok daného typu

  import static `java.lang.System.out;`

- Potom možno k nemu pristupovať v skrátenej forme:

  `out.println("...");`

- Výhodné pre matematické funkcie:[6]

  `import` **static** `java.lang.Math.*;`

- Použitie:

  `out.println(abs(cos(2*PI/3)));`

- Pôvodný, neskrátený zápis sa ťažie sleduje:

  `out.println(Math.abs(Math.cos(2*Math.PI/3)));`

[6] `http://www.agiledeveloper.com/articles/Java5FeaturesPartII.pdf`

Riadenie prístupu

- Uvedením jedného z modifikátorov prístupu (*access modifiers* alebo *access specifiers*): **public**, **protected** a **private** definujeme prístup k prvku
- Neuvedenie žiadneho modifikátora znamená implicitný prístup v rámci balíka (*package access*)
- Vhodné je rozlišovať medzi definovaním:
  - prístupu k prvkom balíka – k triedam a *rozhraniam*
  - prístupu k prvkom tried (*class members*) – k atribútom a metódam, ale aj k *triedam a rozhraniam*
- Modifikátor sa uvádza pred názvom prvku

Prístup k prvkom balíka

- Môže byť len
  - **public** – prvok je prístupný všetkým
  - alebo v rámci balíka – prvok je prístupný len prvkom toho istého balíka
- Triedy a ich prvky definované v úplne nezávislých zdrojových súboroch, pre ktoré nie je uvedený balík sú jedna druhej prístupné
  - príklad **c05:Cake.java** / **c05:Pie.java** v TiJ
- Najviac jedna **public** trieda v rámci jedného zdrojového súboru
  - Súbor a trieda musia mať rovnaký názov

Prístup k prvkom tried

- **public**: prvok je dostupný všade kde trieda
- package access – prístup v rámci balíka: prvok je dostupný len triedam v rámci balíka
  - napr. používateľ knižnice im nemôže pristúpiť
- **private**: prvok je dostupný len v rámci triedy samotnej
  - niekedy je potrebné zabrániť priamemu prístupu k niektorým atribútom alebo volaniu niektorých metód zvonku
- **protected**: prvok je dostupný v rámci balíka a v rámci *hierarchie tried*, do ktorej trieda patrí
  - bude jasnejšie keď preberieme dedenie (*inheritance*)

Prístup k prvkom tried – príklad

```
class Student {
    private String meno;
    ...
    public void nastavMeno(String m) {
        meno = m;
    }
    ...
}

class Zapis {
    public static void main(String[] args) {
        Student s = new Student();
        s.nastavMeno("Peter Petrovič");
        s.meno = "Milan Petrovič"; // chyba pri preklade!
}
```

Agregácia

Agregácia

- Objekt môže obsahovať iné objekty:
  - hodnotou (containment by value)
  - referenciou (containment by reference)
- V Jave len referenciou – s výnimkou inštancií primitívnych typov (ktoré však nie sú objekty v OO zmysle)
- Dosahuje sa prostredníctvom atribútov tried
- Jeden zo spôsobov tvorenia hierarchie (spomeňte si na príklad s obrami)
- Agregáciu sme používali v doterajších príkladoch
  - Preštudujte a vyskúšajte ďalšie v TiJ (Ch. 6, *Composition syntax*)

Agregácia alebo kompozícia?

- Z jazykového hľadiska obidva termíny znamenajú to isté (zloženie, zoskupenie)
- Jazykovo nepodložené, ale žiaľ rozšírené použitie (UML) je:
  - containment by value = composition
  - containment by reference = aggregation
- Radšej:
  - aggregation by value / aggregation by reference
  - composition by value / composition by reference

Dedenie

Dedenie

- Dedenie (*inheritance*) vo vývoji softvéru založené na princípoch *zovšeobecňovania* (generalizácie) a *abstrakcie*
- Odvodená trieda je *špeciálnym prípadom* (všeobecnejšej) pôvodnej triedy, ale typicky ju aj rozširuje, t.j. *konkretizuje*
- Dedenie umožňuje využiť raz definovanú triedu ako základ pre ďalšie triedy od nej odvodené (*derived classes*)

Štruktúra a správanie

- Dá sa dediť:
  - štruktúra, t.j. implementácia
  - správanie (behavior), t.j. rozhranie (*interface*)
- Štruktúra je daná atribútmi a implementáciou metód
- Správanie je dané *typom*, tzn. poskytnutými metódami
- Ťažko ich úplne oddeliť

Rozširovanie tried

- Syntax:

```
class Nadtrieda {
}

class Podtrieda1 extends Nadtrieda {
}
```

- Podtrieda (*subclass*) rozširuje (*extends*) nadtriedu (*superclass*)
- Podtrieda získava štruktúru nadtriedy
  - Môže ju rozšíriť o ďalšie atribúty a metódy
  - Môže zmeniť jej metódy

Príklad: geometrické útvary

- Predpokladajme, že vyvíjame malý grafický systém
- Na chvíľku zabudneme na riadenie prístupu

```
class Utvar {
    int farba;
    void nakresli() {} // neznámy útvar
    void vypln(int f) {
        farba = f; // viac sa pre neznámy útvar nedá urobiť
    }
    void zrus() {} // neznámy útvar
}

class Bod {
    double x, y;
    public Bod(double x, double y) {
        this.x = x;
        this.y = y;
    }
}
```

Príklad: geometrické útvary (2)

```
class Kruh extends Utvar {
    Bod c;
    double r;
    public Kruh(Bod c, double r) { . . . }
    void nakresli() {
        System.out.println("Kruh (" + c.x + ", " + c.y +") r = " + r);
    }
    void vypln(int f) { . . . }
    void zrus() { . . . }

    public static void main(String[] args) {
        Bod b = new Bod(25.0, 50.0);
        Kruh k = new Kruh(b, 10.0);
        k.nakresli(); // "Kruh (25.0, 50.0) r = 10.0"
    }
}
```

Prekonávanie metód (overriding)

- Deklarácia nestatickej metódy rovnakej signatúry v podtriede *prekonáva* (overrides) pôvodnú metódu nadtriedy
  - Pre prekonávanie metód sa používajú tiež termíny predefinovanie, prekrývanie (nevhodné kvôli *skrývaniu*), prípadne potláčanie
- Prekonaná metóda sa z prekonávajúcej dá zavolať pomocou kľúčového slova **super**:

```
class Kruh extends Utvar {
    ...
    void vypln(int f) {
        super.vypln(f);
    }
    ...
}
```

- Takto sa dá zavolať len bezprostredne prekonaná metóda (nie napr. metóda nadnadtypu)

Prekonávanie metód a preťaženie

- Preťaženie metód platí aj pri dedení
- Pozor!
  - ak sa metódy nelíšia v parametroch, dôjde k prekonaniu
  - ak sa metódy líšia v parametroch, dôjde k preťaženiu

Príklad: preťaženie pri dedení

```
class Utvar {
   int farba;
   void nakresli() {}
   ...
}

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

- Použitie:

```
Kruh k = new Kruh(new Bod(25.0, 50.0), 10.0);
k.nakresli(); // ""
```

Skrývanie atribútov tried

- Atribút podtriedy skryje rovnomenný atribút nadtriedy
- K pôvodnému atribútu sa dá pristúpiť pomocou kľúčového slova **super**:

```
class A {
    int p = 0;
}

class B extends A {
    int p = 1;
    void f() {
        System.out.println(p + " " + super.p); // "1 0"
    }
}
```

- Takto sa dá pristúpiť len k bezprostredne skrytému atribútu (ako pri prekonávaní metód)

Riadenie prístupu v dedení

- Čím viac z nadtriedy treba skryť – prístup **private**
- Atribúty nadtriedy s prístupom **private** sú v podtriedach nedostupné
- Často však treba umožniť prístup aj v podtriedach – vhodný je prístup **protected**:
  - prvok je dostupný každej podtriede (v celej hierarchii dedenia)
  - prvok je dostupný triedam v tom istom balíku

```
class Utvar {
    private int farba; // v triede Kruh už nedostupné
    protected void nastavFarbu(int f) {. . .}
    protected int ziskajFarbu() { return farba; }
}
```

Inicializácia pri dedení

- Treba dbať o správnu inicializáciu nadtriedy
- Implicitne sa volajú len konštruktory bez parametrov
- Ostatné konštruktory musia byť vyvolané pomocou kľúčového slova **super**
- Musí to byť prvý príkaz
- Názorný príklad v TiJ (Ch. 6, *Initialization and class loading*)

Príklad: volanie konštruktora nadtriedy

```
class Utvar {
    int farba;
    Utvar(int f) { // implicitný konštruktor bez parametrov už nie je
        ...
    }
    ...
}

class Kruh extends Utvar {
    ...
    Kruh() { super(0); } // inak prekladač hlási chybu
    ...
}
```

Trieda `Object`

- Každá trieda, pre ktorú nie je definovaná klauzula `extends`, implicitne dedí od triedy `Object`
- Trieda Object je súčasťou základného balíka `java.lang`
- Poskytuje užitočné metódy ako napr. `toString()`
  - Tým je táto metóda definovaná pre každú triedu
  - Implicitne ju volá operátor + pre `String`
  - Jej prekonaním možno poskytnúť iný výpis (príklad v TiJ, Ch. 6, *Composition syntax*)

Kľúčové slovo **final**

- Kľúčové slovo **final** zabraňuje zmene prvku, na ktorý sa vzťahuje
- Viac použití:
  - finálne atribúty tried
  - finálne parametre metód
  - finálne metódy
  - finálne triedy

Finálne atribúty tried

- Finálne atribúty tried môžu reprezentovať konštanty inicializované pri preklade

  public final float `PI = 3.14f;`

- Ale môžu byť inicializované aj až pri vykonávaní programu

  private final int `K = f();`

- Môžu byť statické ako aj iné atribúty
- Musia byť explicitne inicializované pri definícii alebo v konštruktore
- Finálna referencia nezabraní zmene samotného objektu!

Finálne parametre

- Metóda môže používať parametre ako hocijaké iné lokálne premenné

```
void m(int i, Utvar u) {
    i = 0;
    u = new Utvar();
}
```

- Zmena platí len v metóde
- Kľúčovým slovom **final** sa tomu dá zabrániť
- Nezabraní sa tým zmene samotného objektu

```
void m(final int i, final Utvar u) {
    u.farba = 1;
}
```

Finálne metódy

- Finálne metódy sa nedajú prekonávať
  - Pokus o prekonanie vyvolá chybu pri preklade
- **private** metódy sú implicitne **final**
  - Pokus o „prekonanie“ prejde – lebo vlastne nejde o prekonanie (pre podtriedu **private** prvky nie sú viditeľné)

Finálne triedy

- Celá trieda môže byť definovaná ako finálna
  - Od finálnej triedy sa nedá dediť
  - Všetky metódy preto budú implicitne finálne (bez dedenia niet prekonávania)
- Atribúty finálnej triedy sa správajú rovnako ako pri triede, ktorá nie je finálna
- Príklady v TiJ, Ch. 6, *The final keyword*
- Použitie kľúčového slova **final** môže viesť k optimalizácii programu
- Ale netreba ho používať za týmto účelom
  „*Premature optimization is the root of all evil.*“
  – *D. Knuth*[7]

[7] http://shreevatsa.wordpress.com/2008/05/16/premature-optimization-is-the-root-of-all-evil/

Dedenie a typy

- Trieda definuje typ
- Typ podtriedy je podtypom nadtriedy
- *Upcasting*: implicitná zmena typu objektu z typu podtriedy na typ nadtriedy
  - *up* – hore: smerom k nadtriede, ktorá je v hierarchii tried postavená *vyššie*
- Majme napríklad takúto metódu:

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

- Táto metóda bude akceptovať objekt podtriedy

```
Kruh k = new Kruh(5, new Bod(10, 10));
zarad(k);
```

- Toto je príprava pre *polymorfizmus* – tému nasledujúcej prednášky

Sumarizácia

Sumarizácia

- Rozmanitosť konštrukcií v Jave
- Preťaženie metód: poskytovanie rôznych funkcionalít pod jedným názvom na základe typu parametrov
- Implicitná inicializácia pomáha predísť mnohým nepríjemnostiam
- Riadenie inicializácie: konštruktorom a blokom príkazov
- Polia ako objekty
- Preklad v Jave
- Balíky: zavedenie balíka je len sprístupnenie priestoru názvov
- Riadenie prístupu: prístup k prvkom balíka a prístup k prvkom tried
- Agregácia: objekt môže obsahovať iný objekt – jeho referencia sa uchováva v atribúte
- Dedenie: trieda môže zdediť štruktúru, ale rozhodujúce je dedenie správania – téma nasledujúcej prednášky

Čítanie

- Dnešná prednáška:
  - OJA, časti 3.8–3.10 a 3.10–3.15 a kapitola 4
  - OJA, kapitola 4
- Na ďalšiu prednášku: OJA, kapitoly 5 a 7