```
public class ZmenaHesovacihoKodu {
  public static void main(String[] args) {
    HashSet hs = new HashSet();
    Citac c = new Citac();
    Citac pom = new Citac();
    System.out.println("citace jsou si rovny: "
                                        + c.equals(pom));

    hs.add(c);
    System.out.println(hs);
    System.out.println("obsahuje 0: "
                           + hs.contains(pom));
    pom.plusJedna();
    System.out.println(hs);
    System.out.println("obsahuje 0: "
                           + hs.contains(pom));
  }
}
```

Vypíše:

```
citace jsou si rovny: true
[Citac@0]
obsahuje 0: true
[Citac@0]
obsahuje 0: false
```

14.3.4. Doporučení pro přípravu perfektní metody hashCode()

Na začátek je třeba opět zdůraznit, že zájemce o vyčerpávající výklad této problematiky jej nalezne v [JE].

Nejprve je nutno zmínit tři pravidla „obecného kontraktu“ pro `hashCode()`, tak jak jsou uvedena v dokumentaci k `java.lang.Object`.

1) Pro tentýž objekt musí `hashCode()` vracet vždy stejný **`int`**. Pravidlo lze porušit, pokud se změní stav objektu a tato změna bude detekována i jinou hodnotou vracenou `equals()`. Pozor ale na to, že když bude takový prvek v kolekci, mohou nastat problémy – viz předchozí příklad.
   Pravidlo lze dále porušit pro různé běhy programu, tj. spustíme-li několikrát tentýž program, mohou se v různých bězích hodnoty hešovacího kódu měnit (pozor při ukládání těchto informací na disk).

2) Rozhodla-li `equals()`, že jsou si dva objekty rovny, musí `hashCode()` vrátit stejný **`int`**.

3) Nejsou-li si objekty rovny podle `equals()`, mohou mít stejný hešovací kód. Je to ale nevhodná implementace `hashCode()`, která významně snižuje efektivitu programu.

Nejčastěji se setkáváme s porušením pravidla 2), kdy přektyjeme `equals()` a `hashCode()` necháme nepřekrytou, tj. zděděnou ze třídy `Object`. To se samozřejmě snadno vyřeší tím, že napíšeme vlastní `hashCode()`. Aby byla efektivní, měla by vytvářet nestejné hešovací kódy pro nestejné objekty. Ve zcela ideálním případě by tyto hešovací kódy měly mít navíc rovnoměrné rozložení, což je často úkol hodný kvalifikovaného matematika.

Naštěstí i v této oblasti byly provedeny četné výzkumy a s využitím jejich výsledků se lze tomuto ideálu dosti přiblížit. V [JE] je uveden následující postup, měřítkem jehož kvality je také to, že je bez úprav převzat i do [TIJ3]. Další měřítko kvality uvidíme v testu níže.

❶ Použijeme pomocnou proměnnou `vysledek` typu **`int`** inicializovanou magickým číslem `17`.

```
int vysledek = 17;
```

❷ Pro každou významnou stavovou proměnnou objektu, která byla použita pro porovnávání v metodě `equals()` (viz výše) vypočteme vlastní hešovací kód a uložíme jej do pomocné proměnné `pom`. Výpočet provádíme v závislosti na typu stavové proměnné (označené jako `sp`) takto

- **`boolean`**: `pom = sp ? 0 : 1;`
- **`byte`**, **`char`**, **`short`**, **`int`**: `pom = (int) sp;`
- **`long`**: `pom = (int) (sp ^ (sp >>> 32));`
- **`float`**: `pom = Float.floatToIntBits(sp);`
- **`double`**:
```
long l = Double.doubleToLongBits(sp);
pom = (int) (l ^ (l >>> 32));
```
- odkaz na objekt: `pom = (sp == null) ? 0 : sp.hashCode();`
- pole: vypočteme `pom` postupně pro každý prvek pole

Po vypočtení `pom` touto hodnotou (společně s další magickou konstantou `37`) ovlivníme proměnnou `vysledek`

```
vysledek = 37 * vysledek + pom;
```

a pokračujeme s další stavovou proměnnou.

❸ Poté, co bude `vysledek` ovlivněn postupně všemi významnými stavovými proměnnými objektu, vrátíme `vysledek` jako výslednou hodnotu metody `hashCode()`.

❹ Zkontrolujeme na příkladech, zda stejné objekty (z pohledu `equals()`) vracejí stejné hešovací kódy. Pokud ne, je třeba zjistit proč a chybu opravit.

Poznámka:

Tento postup vypadá poměrně složitě, ačkoliv ve skutečnosti složitý není. Je možné zpočátku použít jednodušší postup, kdy napíšete metodu `hashCode()` tak, že všechny atributy primitivních datových typů vynásobíte. Po odladění programu pak můžete provést pokusy s rychlostí s touto `hashCode()` a pak s `hashCode()` napsanou podle výše uvedených zásad. Podle výsledků zjistíte, zda je rozdíl výkonností významný.

Podle mých pokusů se ale vždy vyplatí rovnou napsat „perfektní" `hashCode()`. Není zase o tolik složitější a navíc nikdy nevíme, kolik objektů naší třídy někdo v budoucnu vytvoří. Jak uvidíte dále, je nárůst výkonnosti s perfektní `hashCode()` skutečně významný. Neplatí ani předpoklad, že pro malé objemy dat je rozdíl rychlosti výpočtu „jednoduché" a „ideální" `hashCode()` vyrovná případnou neefektivitu uložení v hešovací tabulce. □

14.3.5. Praktické použití perfektní hashCode()

Příklad:

Ukázka efektivnosti různých přístupů k hešování. Je použita třída `Osoba`, která má základní atributy typu **`boolean`**, **`int`**, **`double`** a `String`, aby bylo ukázáno, jak se s těmito typy pracuje. Tato třída má připravenu metodu `equals()` přesně podle doporučení, ale nemá překrytu metodu `hashCode()`. Je třeba zdůraznit, že použité atributy mají z povahy řešeného problému hodnoty ve velmi omezeném rozsahu, což je ale v případě tříd popisujících objekty „z reálného života" běžné.

Od třídy `Osoba` jsou zděděny tři další třídy, které teprve metodu `hashCode()` překrývají. Je třeba zdůraznit, že všechny tři jsou funkční a pokud je použijete pro malé objemy dat (řádu tisíců), nepoznáte při běhu programu výkonnostní rozdíl.

Testovací program vždy připraví pole objektů zvolené třídy. (Třída `Random` použitá v konstruktoru je vysvětlena v [17./195]). Pak (v měřené části) uloží všechny objekty z tohoto pole do `HashSet`. V následujícím měřeném úseku postupně v množině všechny prvky vyhledá.

Třída `NevhodnaOsoba` vrací jako hešovací kód pouze hodnotu atributu `vyska`. Protože však výška může být pouze v rozsahu `170` až `200`, je k dispozici pouze `31` různých hešovacích kódů. To způsobí výrazný nárůst doby běhu programu v závislosti na počtu vložených prvků. Je to z toho důvodu, že jednak skutečný rozdíl mezi prvky musí rozpoznat až metoda `equals()`, ale zejména proto, že hešovací tabulka se změnila na 31 lineárně zřetězených seznamů. To je ovšem implementační detail.

Mnohem lepší je třída `PrijatelnaOsoba`, kde jednoduchou úpravou, která představuje pouze vynásobení číselných atributů třídy (`vyska * vaha`), získáme znatelný nárůst výkonnosti.

S nárůstem výkonnosti bychom snad mohli být spokojeni do té doby, dokud nevyzkoušíme třídu `PerfektniOsoba`, která má metodu `hashCode()` připravenu přesně podle návodu. Při jejím použití zjistíme, že potřebný čas (zejména pro vyhledávání) narůstá s počtem prvků velmi málo.

Poznámka:

Při výpočtu hešovacího kódu je použito opětovného (tj. zbytečného) volání metody `Double.doubleToLongBits(this.vaha);` místo aby se jednoduše použil atribut třídy `longVaha`. Je to proto, aby metoda `hashCode()` rigidně dodržela pravidla stanovená pro její přípravu. V příkladu [14.4./177] metoda `hashCode()` již atribut `longVaha` využívá. □

V příkladu je použita navíc speciální třída `NemennaPerfektniOsoba`. Ta vychází z předpokladu, že hodnoty atributů se nebudou měnit a je tedy možné hešovací kód vypočítat jednou provždy v konstruktoru. Tato konstantní hodnota je pak vracena překrytou metodou `hashCode()`. Je třeba zdůraznit, že se nemění princip výpočtu hešovacího kódu – je stále „perfektní“.

```
import java.util.*;

class Osoba {
  // zakladní stavové atributy
  protected boolean muz;
  protected int vyska;
  protected double vaha;
  protected String jmeno;
  // bitový obraz vahy pro hashCode() a equals()
  // odvozený atribut
  protected long longVaha;

  private static Random r = new Random();
```

```
  Osoba() {
    this.muz = r.nextBoolean();
    // výška <170; 200>
    this.vyska = 170 + r.nextInt(31);
    // váha <50; 100)
    this.vaha = 50 + 50 * r.nextDouble();
    this.longVaha = Double.doubleToLongBits(this.vaha);
    byte[] b = new byte[5];
    for (int i = 0;  i < 5;  i++)
      b[i] = (byte) ((r.nextInt(26) + (byte) 'a'));
    jmeno = new String(b);
  }

  public String toString() {
    return jmeno + ", " + (muz ? "muz " : "zena") +
                              ", " + vyska + ", " + vaha;
  }

  public boolean equals(Object o) {
    if (o == this)
      return true;
    if (o instanceof Osoba == false)
      return false;
    Osoba os = (Osoba) o;
    boolean bMuz = this.muz == os.muz;
    boolean bVyska = this.vyska == os.vyska;
    boolean bVaha = this.longVaha == os.longVaha;
    boolean bJmeno = this.jmeno.equals(os.jmeno);
    return bMuz && bVyska && bVaha && bJmeno;
  }
}

class NevhodnaOsoba extends Osoba {
  public int hashCode() {
    return vyska;
  }
}

class PrijatelnaOsoba extends Osoba {
  public int hashCode() {
    return (int) (vyska * vaha);
  }
}
```

```
class PerfektniOsoba extends Osoba {
  public int hashCode() {
    int vysledek = 17;
    int pom;
    pom = this.muz ? 0 : 1;
    vysledek = 37 * vysledek + pom;
    pom = this.vyska;
    vysledek = 37 * vysledek + pom;
    long l = Double.doubleToLongBits(this.vaha);
    pom = (int) (l ^ (l >>> 32));
    vysledek = 37 * vysledek + pom;
    pom = this.jmeno.hashCode();
    vysledek = 37 * vysledek + pom;
    return vysledek;
  }
}

class NemennaPerfektniOsoba extends PerfektniOsoba {
  protected int hashKod;
  NemennaPerfektniOsoba() {
     super();
     hashKod = super.hashCode();
  }

  public int hashCode() {
    return hashKod;
  }
}

public class TypyHashCode {
  static int pocet;

  public static void main(String[] args) {
    if (args[0] != null)
      pocet = Integer.parseInt(args[0]);

    Osoba[] pole = new Osoba[pocet];

    for (int i = 0;  i < pocet;  i++) {
//      pole[i] = new NevhodnaOsoba();
//      pole[i] = new PrijatelnaOsoba();
      pole[i] = new PerfektniOsoba();
//      pole[i] = new NemennaPerfektniOsoba();
    }
    System.out.println(pole[0].getClass().getName());
```

```
    long zac = System.currentTimeMillis();
    HashSet mnOsob = new HashSet(pocet);
    for (int i = 0;  i < pocet;  i++) {
      mnOsob.add(pole[i]);
    }
    long kon = System.currentTimeMillis();
    System.out.print("Vlozeni: " + mnOsob.size() +
                                    " (" + pocet + ") ");
    System.out.println("cas = " + (kon - zac));

    zac = System.currentTimeMillis();
    int n = 0;
    for (int i = pocet - 1;  i >= 0;  i-) {
      if (mnOsob.contains(pole[i]) == true) {
        n++;
      }
    }
    kon = System.currentTimeMillis();
    System.out.print("Pristup: " + n +
                                    " (" + pocet + ") ");
    System.out.println("cas = " + (kon - zac));
  }
}
```

Po opakovaném spuštění a zpracování do tabulky dostaneme např.:

	počet prvků	1 000	10 000	20 000	30 000
NevhodnaOsoba	vložení	20	1 392	7 050	17 185
	přístup	10	1 351	7 000	16 904
	počet prvků	1 000	10 000	100 000	500 000
PrijatelnaOsoba	vložení	10	51	1 542	25 066
	přístup	10	20	771	18 106
PerfektniOsoba	vložení	10	40	961	8 051
	přístup	10	20	130	651
NemennaPerfektniOsoba	vložení	10	40	871	7 761
	přístup	0	10	80	411

Z výsledků je třeba zdůraznit tyto věci:

1) Hodnoty pro 1000 prvků v kolekci, kdy je čas přístupu k objektům libovolné z uvedených čtyř tříd prakticky neměřitelný, jsou důvodem, proč je správné přípravě metody `hashCode()` obecně věnována tak malá pozornost. Málokdo totiž ukládá do kolekcí desetitisíce objektů, zejména ne ve školních příkladech. Skutečný život je ale jiný a rozdíl mezi `NevhodnaOsoba` a `PerfektniOsoba` je pak rozdílem mezi nefunkčním a funkčním programem.

2) Na příkladu `NemennaPerfektniOsoba` je vidět, že pokud nechceme měnit stav vkládaného objektu, můžeme zvýšit rychlost až o jednu třetinu. To je jeden z mnoha dobrých důvodů, proč používat neměnné třídy, další jsou uvedeny v [JE].

3) Na porovnání `NemennaPerfektniOsoba`, `PrijatelnaOsoba` a `PerfektniOsoba` je dobře vidět, že není nutné mít obavy z přehnané časové náročnosti výpočtu „perfektního" hešovacího kódu. U `PerfektniOsoba` a `NemennaPerfektniOsoba` zůstává čas stále velmi malý. To se u `PrijatelnaOsoba` (s jednodušším výpočtem) zdaleka nedá říci.

4) Podle teorie uváděné dříve by ale časy pro `NemennaPerfektniOsoba` a `PerfektniOsoba` měly zůstávat stále stejné. To platí (uvažujeme-li nepřesnost měření) pouze pro počty prvků do 10 000 a pak se čas zvyšuje. Tento zvyšující se čas ale není způsoben chybou v hešovací funkci ani nesprávností teorie. Je zaviněn konstrukcí procesoru a operačního systému. Při použití 100 000 a více prvků jsou totiž sice všechny prvky uloženy v RAM paměti, ale ne ve stejné oblasti. Přístup k nejvzdálenějším prvkům pak představuje např. několik málo instrukcí strojového kódu navíc, což při tak velkých počtech přístupů způsobí měřitelné zpomalení.

Budeme-li na výše zmíněných třídách provádět pokusy, jak generované hešovací kódy splňují podmínku unikátnosti, zjistíme, že `NevhodnaOsoba` (v souladu s teorií) vygeneruje jen 31 rozdílných kódů a (víceméně) nezáleží na počtu objektů. `PrijatelnaOsoba` pro 100 000 různých objektů vygeneruje 11 147 rozdílných a 88 853 shodných kódů. `PerfektniOsoba` pro tentýž počet objektů poskytne 99 998 rozdílných a pouze 2 shodné. Program je v souboru `RozlozeniHashCode.java`.

14.4. Co všechno by měl mít prvek kolekce

Třída, jejíž objekty se budou vkládat do kolekce, může být libovolná. Chcete-li se však vyhnout mnoha nepříjemnostem, existuje několik málo doporučení (viz též [Tut2]), jaké vlastnosti by takováto třída měla mít.

Pozámka:

Doporučení uvedená v předchozí části se týkala jen metod `equals()` a `hashCode()`. Zde se díváme na třídu komplexně. □

Jako ilustrační příklad použijeme třídu `Jmeno` pro práci se jménem člověka. Pro zjednodušení neobsahuje třída podporu řazení akcentovaných znaků (viz též [14.4.1./177]).