Кључеви у мапи: equals() hashCode()

Класа `Object` дефинише `public` метод `hashCode()`, па се објекат сваке класе може користити као кључ у мапи.
Овај метод није универзално применљив. Пошто за добијање хеш-вредности користи меморијску адресу на којој је објекат смештен, различити објекти увек производе различите хеш-вредности. То је повољно, јер ће операције над хеш-мапом бити ефикасније, међутим, оно што није добро је то да различите инстанце које садрже идентичне податке производе различите хеш-вредности и, у том случају, објекту смештеном у хеш-мапу никада се не може приступити коришћењем друге инстанце кључа, чак и ако је тај кључ идентичан кључу коришћеном за смештање објекта у мапу у свим осталим аспектима. А управо је то случај у већини ситуација: објекат кључ који се користи за претрагу мапе неизбежно је различит од оног коришћеног за смештање одговарајућег пара кључ/вредност у мапу.

Да би објекти наше класе могли да се користе као кључеви у хеш-табели, у класи се морају предефинисати методи `hashCode()` и `equals()` наслеђени од класе `Object`.

Метод
```
public boolean equals(Object o)
```
се користи у методима класе `HashMap<>` за утврђивање да ли су два кључа једнака, па наша верзија овог метода треба да врати `true` када два различита објекта садрже идентичне вредности одговарајућих атрибута.

Метод
```
public int hashCode()
```
враћа хеш-вредност за објекат, типа `int`.

Генерисање хеш-кôдова је огромна тема. Метод `hashCode()` мора испуњавати неке услове да би био од користи.
Треба да се трудимо да вратимо хеш-кôд који има велику вероватноћу да буде јединствен и хеш-кôдови које генеришемо за различите објекте са којима ћемо радити треба да буду што је могуће равномерније распоређени по опсегу вредности типа `int`.
Да бисмо постигли јединственост, типично ћемо желети да комбинујемо вредности значајних атрибута у објекту. Први корак је добити `int` за сваки значајни атрибут. Затим треба искомбиновати те `int`-ове за добијање вредности која ће бити хеш-кôд објекта.

1. начин: Хортонов (није много присутан у пракси)

Један од начина да се ово уради јесте да се сваки `int` који одговара неком атрибуту помножи различитим простим бројем и саберу тако добијени резултати. То би требало да дâ разумну дистрибуцију вредности са великом вероватноћом да вредности буду различите за различите објекте.
Није битно које просте бројеве ћемо користити све док:

- нису тако велики да резултат изађе изван опсега за тип `int`
- користимо различит прост број за сваки атрибут.

Како од атрибута добити `int`?
Генерисање `int`-а за атрибуте типа `String` је просто: само позовемо метод `hashCode()` за атрибут, који је у класи `String` имплементиран тако да произведе добре хеш-кôдове који ће бити једнаки за идентичне стрингове.
`int` атрибуте можемо користити такве какви су, док атрибути реалних типова захтевају нешто обраде.

Нпр. желимо да користимо објекте класе `Osoba` као кључеве у хеш-табели, а атрибути су `ime` и `prezime`, типа `String`, и `godine`, типа `int`. Могли бисмо метод `hashCode()` те класе да имплементирамо овако:

```
public int hashCode(){
	return 13 * ime.hashCode() + 17 * prezime.hashCode() + 19 * godine;
}
```

Ако је атрибут типа неке класе, а не променљива примитивног типа, треба имплементирати метод `hashCode()` у тој класи и користити га при рачунању хеш-кôда за објекат у класи кључа.

2. начин: Eclipse може аутоматски да генерише методе hashCode() и equals():

```
Source > Generate hashCode() and equals()...
```

Појави се прозор у коме се изаберу атрибути које ће методи користити.
Генерисани методи су (у великој мери) у складу са материјалом преведеним из поглавља 3 књиге "Effective Java" (ниже у тексту).

3. начин: Коришћењем метода нове класе Objects (java.util.Objects, уведене у Java 1.7)

Класа се састоји од статичких utility метода за оперисање објектима.
Садржи null-safe и null-tolerant методе за рачунање хеш-кôда објекта, враћање Стринг-репрезентације објекта и поређење објеката:

```
static <T> int      compare(T a, T b, Comparator<? super T> c)
static boolean      deepEquals(Object a, Object b)
static boolean      equals(Object a, Object b)
static int          hash(Object... values)
static int          hashCode(Object o)
static <T> T        requireNonNull(T obj)
static <T> T        requireNonNull(T obj, String message)
static String       toString(Object o)
static String       toString(Object o, String nullDefault)
```

Нама је, у овом тренутку, занимљив метод `hash(Object... values)` који генерише хеш-кôд за секвенцу улазних вредности. Хеш-кôд се генерише као да су све улазне вредности смештене у низ и тај низ хеширан позивом метода `Arrays.hashCode(Object[])`.

Метод је користан за имплементирање `hashCode()` за класе које садрже већи број атрибута.
Нпр. ако објекат има 3 атрибута: x, y и z, може се писати:

```
@Override
public int hashCode() {
	return Objects.hash(x, y, z);
}
```

Упозорење: Када се, као аргумент, проследи само једна референца, повратна вредност није једнака хеш-кôду одговарајућег објекта. Таква вредност може се добити позивом метода `Objects.hashCode(Object)`.

Метод `Objects.equals()` се може употребити на следећи начин:

```
@Override
public boolean equals(Object obj){
	if(this == obj)
		return true;
	if(!(obj instanceof OvaKlasa))
		return false;
	OvaKlasa ok = (OvaKlasa)obj;
	 return Objects.equals(x, ok.x) && Objects.equals(y, ok.y) && Objects.equals(z, ok.z);
}
```

Метод `Objects.equals()` враћа `true` ако су аргументи једнаки, а `false` иначе.
Ако су оба аргумента `null`, враћа се `true`, а ако је тачно један `null`, враћа се `false`.
Иначе, једнакост се утврђује коришћењем метода `equals()` за први аргумент (метод мора бити исправно дефинисан у одговарајућој класи).

Метод `Objects.compare()` враћа 0 ако су аргументи једнаки, а `c.compare(a, b)` иначе. Ако су оба аргумента `null`, враћа се 0.

Теоретски:

`equals()` мора дефинисати релацију еквиваленције (рефлексивна, симетрична и транзитивна) и мора бити конзистентан (ако се објекат не мења, мора враћати исту вредност). Такође, за сваку не-null референцу `o`, `o.equals(null)` мора увек вратити `false`.

`hashCode()` такође мора бити конзистентан (ако се објекат не мења у терминима које користи `equals()`, метод мора враћати исту вредност).

Релација између ова два метода је:
	када је `a.equals(b)`, мора бити `a.hashCode()==b.hashCode()`.

У пракси:

Ако класа предефинише један од ова два метода, мора предефинисати и други.
У том случају, оба метода морају користити исти скуп атрибута.
Ако су два објекта једнака (у складу са методом `equals`), њихови хеш-кôдови такође морају бити једнаки.

Такође, битно је: приликом коришћења хеш-заснованих колекција или мапа (`HashSet`, `LinkedHashSet`, `HashMap`, `Hashtable`, `WeekHashMap`) обезбедити да се хеш-кôдови кључева смештених у колекцију не мењају док су ти објекти у колекцији. Безбедан начин да се то постигне јесте учинити кључеве непроменљивим.

Превод делова поглавља 3 књиге "Effective Java"

equals()

Не постоји начин да се из неапстрактне класе изведе класа чији атрибут треба да учествује у `equals()` поређењима тако да се очувају сва својства која `equals()` треба да има (ту се, пре свега, мисли на симетричност и транзитивност). Међутим, постоји заобилазни начин: дати предност композицији уместо наслеђивању (уместо да се класа изведе из постојеће, има атрибут типа постојеће класе и јавни `get()` метод за тај атрибут). Могуће је додати атрибут поткласи апстрактне класе без нарушавања својстава метода `equals()`.

Рецепти за писање квалитетних `equals()` метода:

1. користити оператор == како би се проверило да ли је аргумент референца на текући објекат.
Ако јесте, вратити `true`. Ово је само оптимизација перформанси, али вреди ако је поређење потенцијално скупо.

2. користити оператор `instanceof` за проверу да ли је аргумент исправног типа.
Ако није, вратити `false`. Типично, исправан тип је класа у којој се метод налази. Повремено, то је неки од интерфејса који та класа имплементира (нпр. `Set`, `List`, `Map`, `Map.Entry`).

3. кастовати аргумент у исправан тип. Како кастовању претходи тест са `instanceof`, оно ће гарантовано успети.

4. за сваки значајни атрибут класе, проверити да ли атрибут аргумента одговара атрибуту текућег објекта.
Ако сви ти тестови успеју, вратити `true`, а иначе `false`. Уколико је тип у кораку 2 интерфејс, атрибутима аргумента мора се приступати одговарајућим методима интерфејса.
За атрибуте примитивног типа, осим `float` и `double`, за поређења користити оператор ==,
за атрибуте референтног типа, позвати рекурзивно метод `equals()`.
За атрибуте типа `float` користи се поређење на једнакост (оператором ==) `int`-ова добијених позивима метода `Float.floatToIntBits()`, а за атрибуте типа `double`, поређење на једнакост (оператором ==) `long`-ова добијених позивима метода `Double.doubleToLongBits()`.
За низове је потребно применити ова упутства на сваки елемент.
Како би се избегла појава `NullPointerException`, користити следећи идиом за поређење атрибута референтног типа:

```
(field == o.field || (field != null && field.equals(o.field)))
```

За неке класе, поређење атрибута је комплексније од простих тестова једнакости. Требало би да је из спецификације класе јасно да то јесте случај. Ако је тако, може се чувати канонička форма сваког објекта, тако да метод `equals()` врши јевтина поређења канонических форми уместо скупљих поређења. Ова техника је најпогоднија за непроменљиве класе јер се канонička форма ажурира када се објекат промени.

Перформансе метода `equals()` могу зависити од редоследа којим се атрибути пореде. За најбоље перформансе, најпре треба поредити атрибуте за које је вероватније да су различити, јевтиније их је поредити или, идеално, важе оба претходна услова. Не смеју се поредити атрибути који нису део логичког стања објекта (попут атрибута класе `Object` који се користе за синхронизацију). Није неопходно поредити редундантне атрибуте, који се могу израчунати из "значајних", али то може допринети побољшању перформанси метода `equals()` (нпр. ако поређење редундантних атрибута не успе, штеди се на поређењу стварних података).

5. Када завршите писање метода `equals()` запитајте се 3 питања: да ли је симетричан, транзитиван и конзистентан? (преостала два својства се постарају сама за себе). Ако није, откријте због чега није и у складу са тим измените метод.

hashCode()

Увек предефинишите `hashCode()` када предефинишете `equals()`.
Уколико се то не учини, нарушава се услов да једнаки објекти морају имати једнаке хеш-кôдове.

Добра хеш-функција тежи да производи неједнаке хеш-кôдове за неједнаке објекте.
Идеално, хеш-функција треба да дистрибуира сваку колекцију неједнаких инстанци униформно по свим могућим хеш-вредностима.
Достизање овог идеала може бити екстремно тешко. На срећу, није тешко достићи добру апроксимацију. Следи једноставан рецепт:

1. У променљиву `result` типа `int` сместите константну не-нула вредност, нпр. 17.
2. За сваки значајни атрибут `f` објекта (тј. сваки атрибут узет у обзир методом `equals`)
   урадите следеће:
   а. израчунајте хеш-кôд `c`, типа `int`, за атрибут:
      i. ако је атрибут типа `boolean`, израчунајте `(f ? 0 : 1)`.
      ii. ако је атрибут типа `byte`, `char`, `short` или `int`, израчунајте `(int)f`.
      iii. ако је атрибут типа `long`, израчунајте `(int)(f ^ (f >>> 32))`.
      iv. ако је атрибут типа `float`, израчунајте `Float.floatToIntBits(f)`.
      v. ако је атрибут типа `double`, израчунајте `Double.doubleToLongBits(f)` и затим хеширајте резултујући `long` као у кораку 2.а.iii.
      vi. ако је атрибут референца на објекат и метод `equals()` ове класе га пореди рекурзивно позивајући `equals()`, рекурзивно позовите `hashCode()` за атрибут. Ако је потребно комплексније поређење, израчунајте "каноничку репрезентацију" атрибута и позовите `hashCode()` за њу. Ако је вредност атрибута `null`, вратите 0 (или неку другу константу, али традиционално се враћа 0).
      vii. ако је атрибут низ, третирајте га као да је сваки елемент посебан атрибут. То значи, израчунајте хеш-вредност сваког значајног елемента примењујући ова правила рекурзивно и комбинујте ове вредности као што је описано у кораку 2.б.

   б. комбинујте хеш-вредност `c` израчунату у кораку а. са `result` на следећи начин:

```
result = 37*result + c;
```

3. Вратите `result`.
4. Када завршите писање метода `hashCode()`, запитајте се да ли једнаке инстанце имају једнаке хеш-вредности.
   Ако немају, откријте због чега и поправите метод.

Не-нула иницијална вредност се користи у кораку 1. како би иницијални атрибути, чија је хеш-вредност израчуната у кораку 2.а. нула утицали на хеш-вредност текућег објекта. Да је у кораку 1. коришћена иницијална вредност 0, то не би био случај, што би повећало број колизија. Вредност 17 је произвољна.

Множење у кораку 2.б. чини хеш-вредност зависном од редоследа атрибута, што резултује много бољом хеш-функцијом ако класа садржи већи број сличних атрибута. Нпр. да је множење изостављено из хеш-функције класе `String`, сви анаграми би имали идентичне хеш-кôдове. Фактор 37 је изабран јер је непаран и прост. Да је паран и деси се прекорачење при множењу, информације би биле изгубљене јер је множење са 2 еквивалентно шифтовању. Предности коришћења простих бројева су мање јасне, али они се традиционално користе на овом месту.

Ако је класа непроменљива, а цена израчунавања хеш-кôдова значајна, може се размотрити кеширање хеш-кôда у објекту, уместо његовог израчунавања сваки пут када је потребан. Ако верујете да ће већина објеката класе бити коришћена као кључеви, хеш-кôд треба израчунати приликом креирања објекта, а иначе се може лењо иницијализовати први пут када се позове метод `hashCode()`.
Више о лењој иницијализацији у наставку документа.

Не дођите у искушење да изоставите значајне делове објекта из израчунавања хеш-кôда како бисте побољшали перформансе. Иако резултујућа хеш-функција може бити бржа, квалитет може опасти толико да хеш-табела постане неупотребљиво спора. Посебно, хеш-функција може у пракси бити суочена са великом колекцијом инстанци које се разликују у регионима које смо одлучили да занемаримо. Ако се то деси, хеш-функција мапира све те инстанце у веома мали број хеш-кôдова и хеш-засноване колекције имају квадратне уместо линеарних перформанси. То није само теоретски проблем. Хеш-функција класе `String` у верзијама пре Java 1.2 радила је са највише 16 карактера, равномерно распоређених у стрингу, почев од првог карактера. За велике колекције хијерархијских имена, попут URL-ова, ова функција показивала је управо описано патолошко понашање.

НЕОБАВЕЗНО:

Ако је објекат непроменљив, `hashCode()` је кандидат за кеширање и лењу иницијализацију:
рачуна се само једном и само ако је то потребно (то је могуће једино када је објекат непроменљив)

Лења иницијализација има два циља:
- одлагање скупе операције све док она није апсолутно неопходна
- чување резултата те скупе операције, тако да није неопходно њено понављање.

Као и обично, побољшање перформанси веома зависи од проблема који се решава и у многим случајевима није значајно.
Као и сваку оптимизацију, ову технику треба користити само ако је јасно да води значајном побољшању.

Како би се избегао `NullPointerException`, класа и сâма мора приступати својим атрибутима који се лењо иницијализују помоћу метода, а не директно (self-encapsulation). У доњем примеру, атрибуту `fHashCode` унутар класе приступало би се коришћењем метода `hashCode()`.

пример:

```
private int fHashCode;

public int hashCode() {
  if ( fHashCode == 0 ) {
    fHashCode = //... racunanje hes-koda po gornjem receptu
  }
  return fHashCode;
}
```

Лења иницијализација је посебно корисна за графичке корисничке интерфејсе који се дуго креирају.
Постоји неколико политика које се могу користити:
- конструиши увек - прозор се креира много пута, кад год је то потребно и резултат се не кешира
- конструиши на први захтев - прозор се креира једном, први пут када се то захтева.
  Резултат се кешира, за евентуалне будуће захтеве
- конструиши у позадини - прозор се креира једном, у позадинској нити ниског приоритета, приликом иницијализације система.
  Резултат се кешира, за евентуалне будуће захтеве.

Непроменљиви (енг. immutable) објекти

Непроменљиви објекти су просто објекти чији подаци не могу бити промењени након њиховог креирања.
Примери из Јавине библиотеке су објекти класа `String` и `Integer`.

Непроменљиви објекти знатно упрошћавају програм јер:
- једноставно се креирају, тестирају и користе
- аутоматски су thread-safe и немају проблеме са синхронизацијом
- није им потребан копи-конструктор
- није потребна имплементација метода `clone()`
- допуштају да `hashCode()` користи лењу иницијализацију и кешира повратну вредност
- није неопходно њихово копирање када су атрибути класе
- погодни су као кључеви у мапи и елементи у скупу (ови објекти не смеју мењати стање док су у колекцији)
- када непроменљиви објекат избаци изузетак, никада не остаје у нежељеном и неодређеном стању

Упутства за писање непроменљиве класе:
- обезбедити да се из класе не може извести нова - `final` класа или статички factories методи и `private` конструктори
- `private` и `final` атрибути
- комплетно креирање објекта у једном кораку (уместо коришћења подразумеваног конструктора у комбинацији са позивима `set*()` метода)
- не обезбеђивати методе који на било који начин могу променити стање објекта - не само методе `set*()`, већ било које методе који могу променити стање објекта
- ако класа поседује променљиве атрибуте, они се морају копирати када се прослеђују између класе и њеног позиваоца

У својој књизи "Effective Java", Joshua Bloch препоручује:
"Класе треба да буду непроменљиве, осим ако постоји веома добар разлог за супротно...
Ако класа не може бити непроменљива, ограничите променљивост колико је то могуће."

Пример непромењиве класе:

```
/** Planeta je nepromenljivi objekat, jer nema nacina za menjanje njenog stanja nakon kreiranja */
public final class Planeta {

  public Planeta (double masa, String ime, Date datumOtkrica) {
     this.masa = masa;
     this.ime = ime;
     // kreiranje privatne kopije za datumOtkrica
     // to je jedini nacin da se this.datumOtkrica ucini privatnim i zastiti
     // od promena koje pozivalac moze napraviti na originalnom objektu datumOtkrica
     this.datumOtkrica = new Date(datumOtkrica.getTime());
  }

  /**
   * Vraca vrednost primitivnog tipa.
   * Pozivalac moze raditi sta god hoce sa povratnom vrednoscu,
   * bez uticaja na unutrasnjost ovog objekta. On vidi svoj sopstveni double
   * koji prosto ima istu vrednost kao masa
   */
  public double getMasa() {
    return masa;
  }

  /**
   * Vraca nepromenljivi objekat.
   * Pozivalac dobija direktnu referencu na atribut. Medjutim, to
   * ne predstavlja opasnost, jer je String nepromenljiv.
   */
  public String getIme() {
    return ime;
  }

//  /**
//   * Vraca promenljivi objekat - LOSE!.
//   * Pozivalac dobija direktnu referencu na atribut. To je obicno opasno,
//   * jer stanje objekta tipa Date se moze promeniti i ovom klasom i pozivajucom klasom.
//   * Prema tome, ova klasa vise nema potpunu kontrolu nad datumOtkrica.
//   */
//  public Date getDatumOtkrica() {
//    return datumOtkrica;
//  }

  /**
   * Vraca promenljivi objekat - DOBRO.
   * Vraca kopiju atributa.
   * Pozivalac ovog metoda moze raditi sta god zeli sa vracenim objektom tipa Date,
   * bez ikakvog uticaja na ovaj objekat, jer ne poseduje referencu na datumOtkrica.
   * Umesto toga, on ima drugi Date objekat koji inicijalno ima iste podatke kao datumOtkrica.
   */
  public Date getDatumOtkrica() {
    return new Date(datumOtkrica.getTime());
  }

  // PRIVATE //

  /** Final podatak primitivnog tipa je uvek nepromenljiv. */
  private final double masa;

  /** Nepromenljivi atribut. (String objekti nikada ne menjaju stanje.) */
  private final String ime;

  /**
   * Promenljivi atribut. U ovom slucaju, stanje ovog promenljivog atributa
   * menja se samo ovom klasom.
   */
  private final Date datumOtkrica;
}
```