Поредак објеката (интерфејси Comparable<> и Comparator<>)

Имплементирајући интерфејс `Comparable`, класа указује да њене инстанце имају природни поредак. Сортирање низа `a` објеката чија класа имплементира `Comparable` је једноставно: `Arrays.sort(a)`
Такође, једноставна је претрага, израчунавање екстремних вредности, аутоматско одржавање сортираних колекција `Comparable` објеката.

Имплементирајући `Comparable` омогућујете се да Ваша класа може користити у свим многобројним генеричким алгоритмима и колекцијама које зависе од овог интерфејса. Уз мало труда добијате много.

Услови које једини метод интерфејса, `compareTo()`, мора да задовољава слични су условима за метод `equals()`: метод пореди текући и задати објекат и враћа негативни цео број, нулу или позитиван цео број када је текући објекат мањи од, једнак или већи од задатог објекта. Избацује `ClassCastException` ако тип задатог објекта онемогућује његово поређење са текућим објектом.

У наредној нотацији, sgn(izraz) означава математичку функцију знака која враћа -1, 0 или 1 када је вредност израза негативна, нула, односно позитивна, тим редом.

Мора важити:

- sgn(`x.compareTo(y)`) == -sgn(`y.compareTo(x)`) за све `x` и `y`
  (одавде следи да `x.compareTo(y)` мора избацити изузетак акко `y.compareTo(x)` избацује тај изузетак).
- релација је транзитивна: ако `x.compareTo(y)>0` && `y.compareTo(z)>0`, онда `x.compareTo(z)>0`
  (и исто за <).
- ако `x.compareTo(y)==0`, онда sgn(`x.compareTo(z)`)==sgn(`y.compareTo(z)`), за свако `z`.

Строго се препоручује, али се стриктно не захтева да важи:
`(x.compareTo(y)==0) == (x.equals(y))`. Свака класа која имплементира интерфејс `Comparable` и нарушава овај услов треба јасно да укаже на ту чињеницу. Препоручује се да се то учини напоменом: „Note: This class has a natural ordering that is incosistent with equals“.

Математичка природа услова не треба да Вас обесхрабри јер они уопште нису компликовани као што можда изгледају. Допуштено је избацити `ClassCastException` ако се ради о два објекта различитих класа.

Као што класа која нарушава услове за `hashCode()` може „покварити“ друге класе које зависе од хеширања, класа која нарушава услове за `compareTo()` може „покварити“ друге класе које зависе од поређења. Ту спадају сортиране колекције, `TreeSet` и `TreeMap`, као и класе `Collections` и `Arrays`, које садрже алгоритме за сортирање и претрагу.

Први услов каже да уколико се промени смер поређења два објекта, деси се очекивано: ако је први објекат мањи од другог, други мора бити већи од првог; ако је први објекат једнак другом, други мора бити једнак првом; ако је први објекат већи од другог, други мора бити мањи од првог.

Други услов каже да ако је један објекат већи од другог и други већи од трећег, онда први мора бити већи од трећег.

Последњи услов каже да сви једнаки објекти морају давати једнаке резултате приликом поређења са неким другим објектом.

Једна од последица ова три услова је да тест једнакости метода `compareTo()` мора задовољавати иста ограничења као `equals()`, што треба имати на уму ако се жели извести класа из неапстрактне базне класе са атрибутом који треба да учествује у поређењима.

Препорука каже да тест једнакости у `compareTo()` треба да врати исто што и `equals()`. Ако је то случај, каже се да је поредак дефинисан методом `compareTo()` **конзистентан** са `equals()`. Класа чији метод `compareTo()` дефинише поредак који није конзистентан са `equals()` ће и даље радити, али сортиране колекције које садрже елементе класе можда неће испуњавати опште услове одговарајућих колекцијских интерфејса (`Collection`, `Set`, `Map`). Ово стога што су ти услови дефинисани у терминима метода `equals()`, али сортиране колекције, уместо њега, користе тест једнакости из метода `compareTo()`.

Нпр. метод `compareTo()` класе `Float` није конзистентан са `equals()`. Уколико се креира `HashSet` и додају `new Float(-0.0f)` и `new Float(0.0f)`, скуп ће садржати два елемента, јер две додате инстанце нису једнаке поређене методом `equals()`. Ако се, међутим, исти поступак понови користећи `TreeSet` уместо `HashSet`, скуп ће садржати само један елемент, јер су горње две инстанце једнаке поређене методом `compareTo()`.
Метод `compareTo()` класе `BigDecimal` такође није конзистентан са `equals()`. Нпр. за
```
BigDecimal b1 = new BigDecimal("2.0");
BigDecimal b2 = new BigDecimal("2.00");
```
`b1.equals(b2)` враћа 0, док `b1.compareTo(b2)` враћа 0.

Писање метода `compareTo()` слично је писању метода `equals()`, уз неколико кључних разлика.
Није неопходна провера типа. Ако аргумент није одговарајућег типа, метод треба да избаци `ClassCastException`.
Ако је аргумент `null`, метод треба да избаци `NullPointerException`. То је управо понашање које се добије ако се само кастује аргумент у исправан тип и затим покуша приступ његовим члановима.

Поређења атрибута типа класа врши се рекурзивним позивима метода `compareTo()`. Ако одговарајућа класа не имплементира интерфејс `Comparable` или је потребно користити нестандардни поредак, може се користити експлицитни `Comparator`. Може се написати сопствени компаратор или користити предефинисани (нпр. `String.CASE_INSENSITIVE_ORDER`).
`x.equals(null)` треба да врати `false`, док `x.compareTo(null)` треба да избаци `NullPointerException`.

Енумерација: `compareTo()`, као и за сваку другу класу.

Нумерички примитивни типови се пореде релационим операторима < и >, осим `float` и `double`, које треба поредити помоћу `Float.compare(float, float)` и `Double.compare(double, double)`.
`boolean` примитивни тип: користити тестове облика `(x && !y)`.
Напомена: од **Јава 1.7** постоји метод **`compare()` у свим омотач-класама примитивних типова** (претходно је постојао само у `Float` и `Double`, од Јава 1.4).

Низ: применом ових упутстава елемент по елемент.

Колекција: неке колекције немају дефинисан редослед итерирања, па поређење елемент по елемент у тим случајевима није смислено.

Уколико класа поседује већи број значајних атрибута, редослед којим се они пореде је критичан. Мора се започети од најзначајнијег атрибута и настављати са све мање значајним. Ако резултат поређења није нула (нула представља једнакост), завршено је: само се врати тај резултат. Ако су најзначајнији атрибути једнаки, пореде се следећи по значајности итд. Ако су сви атрибути једнаки, објекти су једнаки: врати се 0.

Ако је задатак сортирати уносе релационе базе података, обично је много пожељније допустити да база изврши сортирање користећи клаузу ORDER BY.

Трик са коришћењем разлике атрибута типа `int` би требало избегавати или га користити уз екстреман опрез. Не примењивати га осим ако смо сигурни да атрибут не може имати негативну вредност или, општије, да је разлика између најмање и највеће могуће вредности атрибута мања или једнака од `Integer.MAX_VALUE` (што је: 2^{31}-1) . Разлог због ког трик не функционише у општем случају је тај што означени 32-битни цео број није довољно велик да представи разлику произвољна два означена 32-битна броја. Ако је `i` велики позитиван, а `j` велики негативан `int`, `(i-j)` ће изаћи из опсега типа `int` и биће враћена негативна вредност. Резултујући метод `compareTo()` неће радити. Враћаће бесмислене резултате за неке аргументе и нарушаваће први и други услов. Ово није проблем чисто теоретске природе: узроковао је грешке у стварним системима. Ове грешке је тешко открити јер метод ради исправно за многе вредности.

Компаратори

Уколико је потребно сортирати објекте на начин који се разликује од њиховог природног поретка или се желе сортирати објекти класе која не имплементира интерфејс `Comparable`, неопходно је обезбедити `Comparator` – објекат који енкапсулира поредак. Интерфејс поседује само један метод:

```
public interface Comparator<T>{
      int compare(T o1, T o2);
}
```

Метод пореди своја два аргумента, враћајући негативан цео број, 0 или позитиван цео број у зависности од тога да ли је први аргумент мањи од, једнак или већи од другог. Ако је било који од аргумената неодговарајућег типа, метод избацује `ClassCastException`.

Већина онога што је речено за `Comparable` важи и за `Comparator`. Писање метода `compare()` је скоро идентично писању метода `compareTo()`, осим што први добија оба објекта као аргументе.

Претпоставимо да имамо класу `Zaposleni`:

```
public class Zaposleni implements Comparable<Zaposleni> {
      public Ime getIme() {…}
      public int getBroj() {…}
      public Date getDatumZaposlenja() {…}
}
```

и да је природни поредак запослених по имену. Међутим, шеф тражи листу запослених сортирану по дужини стажа. Следећи програм формира такву листу:

```
import java.util.*;
public class ZaposlSort {
      static final Comparator<Zaposleni> STAZ_POREDAK = new Comparator<Zaposleni>(){
            public int compare(Zaposleni z1, Zaposleni z2){
                  return z2.getDatumZaposlenja().compareTo(z1.getDatumZaposlenja());
            }
      };


      // baza zaposlenih
      static final Collection<Zaposleni> radnici = …;

      public static void main(String[] args) {
            List<Zaposleni> z = new ArrayList<Zaposleni>(radnici);
            Collections.sort(z, STAZ_POREDAK);
            System.out.println(z);
      }
}
```

Програм је праволинијски. Заснива се на природном уређењу класе `Date` примењеном на вредности враћене приступним методом `getDatumZaposlenja()`.`Comparator` пореди датум запослења другог запосленог са датумом запослења првог, јер запослени који се последњи запослио има најкраћи стаж.

Друга техника коју људи повремено користе да постигну овај ефекат је да одрже редослед аргумената, али да негирају резултат поређења:

```
// Ne raditi ovo!
return -z1.getDatumZaposlenja().compareTo(z2.getDatumZaposlenja());
```

Увек треба користити прву технику јер за другу се не може гарантовати да ради. Разлог је што метод `compareTo()` може вратити произвољан негативни `int` ако је његов аргумент мањи од објекта за који је позван. Постоји један негативни `int` који остаје негативан када се негира:

```
-Integer.MIN_VALUE == Integer.MIN_VALUE
```

`Comparator` из претходног програма добро ради за сортирање листе (`List`), али има један недостатак: не може се користити за уређење сортиране колекције, попут `TreeSet`, јер генерише уређење које није конзистентно са `equals()`. Ово значи да `Comparator` изједначава објекте које метод `equals()` не изједначава. Посебно, било која два запослена који су почели да раде истог дана биће упоређени као једнаки. Приликом сортирања листе, то није од значаја; међутим, када се `Comparator` користи за уређење сортиране колекције, то је фатално. Ако се `Comparator` користи за убацивање већег броја запослених на исти дан у `TreeSet`, само први од њих ће бити додат у скуп; остали ће бити виђени као дупликати првог и биће игнорисани.

Како би се проблем решио, просто се `Comparator` напише тако да производи поредак конзистентан са `equals()`. Другим речима, напише се тако да су једини елементи који су једнаки коришћењем метода `compareTo()` такође једнаки и када се пореде методом `equals()`. Начин да се ово изведе је дводелно поређење, где је први део онај за који смо заинтересовани, а други поређење атрибута који јединствено идентификује (идентификују) објекат. У овом случају, то је очигледно број запосленог:

```
static final Comparator<Zaposleni> STAZ_POREDAK = new Comparator<Zaposleni>(){
	public int compare(Zaposleni z1, Zaposleni z2){
		int rez = z2.getDatumZaposlenja().compareTo(z1.getDatumZaposlenja());
		if(rez != 0)
			return rez;

		return Integer.compare(z1.getBroj(), z2.getBroj());
	}
} ;
```