Хеширане, колекции и компаратори

Любомир Чорбаджиев

Технологично училище "Електронни системи"
Технически университет, София

15 март 2017 г.

Забележка: Тази лекция е адаптация на:

- Scott Ostler: *Hashing, Collections, and Comparators* from *6.092: Java for 6.170* (MIT OpenCourseWare: Massachusetts Institute of Technology)
  Лиценз: Creative commons BY-NC-SA

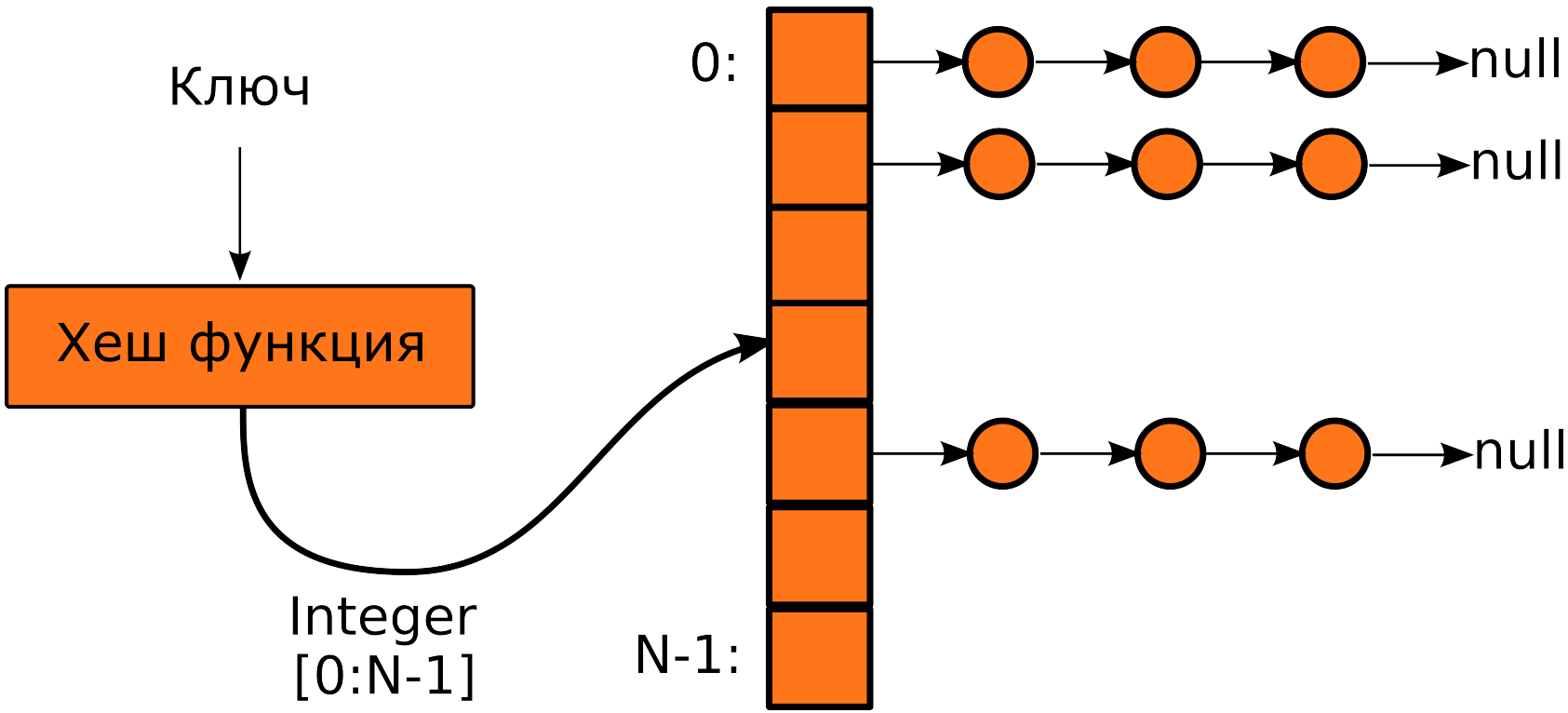
Съдържание

1. Хеширане
2. Колекции
3. Сравняване и сортиране

Хеширане

- Миналият път разгледахме как се предефинира (override) метода `equals` на класа `java.lang.Object`
- Днес ще разгледаме как се предефинира (override) метода `hashCode`.
- Целта е да разберем защо това е необходимо и как се прави.

Хеширане



Хеш-функция

- Хеш-функцията на обект от даден клас съпоставя цяло число, което се наричат хеш-код
- За обектите, които са равни (в смисъла на `Object.equals()`), хеш-функцията трябва да дава еднакви хеш-кодове.
- Когато хеш-функцията връща различни числа за дадени обекти, то тези обекти задължително трябва да са различни (в смисъл на `!Object.equals()`).

Методът `hashCode`

- Метод на класа `java.lang.Object`, който връща цяло число
- Дефиницията на `hashCode()` в класа `java.lang.Object` връща цяло число, което е практически уникално за дадения екземпляр (на основата на разположението на обекта в паметта).
- Всеки клас притежава такъв метод — или метода, който е наследен от `java.lang.Object`, или го предефинира (override)

Изисквания към `hashCode`

- Хеш кодът на обекта не трябва да се променя, докато обектът не се променя.
- Два обекта, които са равни (в смисъл на `equal`), трябва да имат еднакви хеш кодове
- Когато два обекта не са равни би било много добре хеш кодове им да не са еднакви

Пример: hashCode

```
1 String scott = "Scotty";
2 String scott2 = "Scotty";
3 String corey = "Corey";
4 System.out.println(scott.hashCode());
5 System.out.println(scott2.hashCode());
6 System.out.println(corey.hashCode());
```

-1823897190, -1823897190, 65295514

```
1 Integer int1 = 123456789;
2 Integer int2 = 123456789;
3 System.out.println(int1.hashCode());
4 System.out.println(int2.hashCode());
```

123456789, 123456789

Пример: class Name

```
1  public  class Name {
2    public String first;
3    public String last;
4    public Name(String first, String last) {
5      this.first = first;
6      this.last = last;
7    }
8    public String toString() {
9      return first + "␣" + last;
10   }
11   public boolean equals(Object o) {
12     return (o instanceof Name &&
13         ((Name) o).first.equals(this.first) &&
14         ((Name) o).last.equals(this.last));
15   }
16 }
```

Пример: class Name и hashCode

```
1  Name kyle = new Name("Kyle", "MacLaughlin");
2  Name jack = new Name("Jack", "Nance");
3  Name jack2 = new Name("Jack", "Nance");
4
5  System.out.println(kyle.equals(jack));
6  System.out.println(jack.equals(jack2));
7
8  System.out.println(kyle.hashCode());
9  System.out.println(jack.hashCode());
10 System.out.println(jack2.hashCode());
```

false, true, 6718604, 7122755, 14718739

- Обектите jack и jack2 са равни, но хеш кодовете им са различни.

Наистина ли `hashCode` е толкова важна?

- Кодът на класът **class** `Name` изглежда работоспособен
- Наистина ли е толкова важно функцията `hashCode` да спазва изискванията?
- Ако не планираме да използваме функцията `hashCode` защо трябва да я предефинираме?

Наистина ли `hashCode` е толкова важна?

- Ако не предефинираме метода `hashCode` в **class** `Name`, то дефиницията която наследяваме от класа `Object` ще противоречи на изискванията към хеш код на обект
- Това може да доведе до неочаквани и странни резултати

Неочаквани странни резултати

```
1 Set<String> strings = new HashSet<String>();
2 strings.add("jack");
3 System.out.println(strings.contains("jack"));
4
5 Set<Name> names = new HashSet<Name>();
6 names.add(new Name("Jack", "Nance"));
7 System.out.println(names.contains(
8         new Name("Jack", "Nance")));
```

true, false

Предефиниране на `hashCode`

За да се избегнат неочаквани и странни ефетки е необходимо методът `hashCode` да изпълнява следните изисквания:

- `hashCode` трябва да е еднакъв за обекти, които са равни
- `hashCode` трябва да не се променя между извиквания на метода при положение, че обектът не се променя.
- `hashCode` трябва да връща цели числа (**int**)
- препоръчително е `hashCode` да връща различни стойности за обекти, които не са равни

Пример: Предефиниране на `hashCode`

```
1 public class Name {
2   ...
3   public int hashCode() {
4     return 1;
5   }
6 }
```

Пример: Предефиниране на hashCode

```
1 public class Name {
2   ...
3   public int hashCode() {
4     return first.hashCode() + last.hashCode();
5   }
6 }
```

```
1 Set<Name> names = new HashSet<Name>();
2 names.add(new Name("Jack", "Nance"));
3 System.out.println(names.contains(
4         new Name("Jack", "Nance")));
```

true

Пример: Подобрена реализация на `hashCode`

```
1 public class Name {
2   ...
3   public int hashCode() {
4     return first.hashCode() + 37*last.hashCode();
5   }
6 }
```

- Защо тази реализация е по-добра?

Изисквания към `hashCode`

- `hashCode` трябва да е еднакъв за обекти, които са равни
- `hashCode` трябва да не се променя между извиквания на метода при положение, че обектът не се променя.
- `hashCode` трябва да връща цели числа (**int**)
- препоръчително е `hashCode` да връща различни стойности за обекти, които не са равни

```
1 Name name1=new Name("Jack", "Nance");
2 Name name2=new Name("Nance","Jack");
```

```
name1.hashCode()!=name2.hashCode()
```

Колекции

- Служат за съхранение и обработка на данни
- Съвкупност от класове и интерфейси служещи за:
  - Добавяне на обекти
  - Съхранение на обекти
  - Сортиране на обекти
  - Обхождане на обекти
  - Извличане на обекти
- Предоставят сходен интерфей за достъп до различни реализации на колекции

Пример: Използване на колекции

- Класовете и интерфейсите са дефинирани в групата пакети java.util.*

```
1  package lab2;
2
3  import java.util.*;
4
5  public class CollectionUser {
6    List<String> list = new ArrayList<String>();
7    // ...
8    // ... останалата част от класа
9    // ...
10 }
```

Основни методи на `Collection<Foo>`

- **boolean** `add(Foo o)`
- **boolean** `contains(Foo o)`
- **boolean** `remove(Foo o)`
- **int** `size()`

Пример: Използване на колекции

```
1  List<Name> nameList = new ArrayList<Name>();
2
3  nameList.add(new Name("Laura", "Dern"));
4  nameList.add(new Name("Toby", "Keeler"));
5  System.out.println(nameList.size());        // => 2
6
7  nameList.remove(new Name("Toby", "Keeler"));
8  System.out.println(nameList.size());        // => 1
9
10 List<Name> nameList2 = new ArrayList<Name>();
11 nameList2.add(new Name("Scott", "Ostler"));
12 nameList.addAll(nameList2);
13
14 System.out.println(nameList.size());        // => 2
```

Типизирани колекции

- Типизираните колекции предоставят възможност да се укаже явно типът на обектите, които ще бъдат съхранявани в колекцията

```
1 List<String> stringList;
```

- По този начин се гарантира, че в колекцията може да има само обекти от даден тип
- Не е задължително да се указва типът, но е препоръчително и много удобно

Пример: Използване на типизирани колекции

```
1 List untyped = new ArrayList();
2
3 Object obj = untyped.get(0);
4 String sillyString = (String) obj;
```

```
1 List<String> typed = new ArrayList<String>();
2
3 String smartString = typed.get(0);
```

Пример: Обхождане на елементите на колекция

```
1 Collection<Foo> coll;
2 ...
3 Iterator<Foo> it = coll.iterator();
4 while (it.hasNext()) {
5   Foo obj = it.next();
6   // действия с обекта obj
7 }
```

```
1 Collection<Foo> coll;
2 ...
3 for (Foo obj : coll) {
4   // действия с обекта obj
5 }
```

Изтриване на елементи от колекция

- Елементи не могат да бъдат изваждани от колекция докато тя се обхожда. Методът `remove()` на колекцията ще генерира изключение `ConcurrentModificationException`

```
1 for (Foo obj : coll) {
2   coll.remove(obj); // throws
3 }
```

Изтриване на елементи от колекция

- Елементите могат да бъдат изтривани от колекция по време на обхождане посредством итератора:

```
1 Iterator<Foo> it = coll.iterator();
2 while (it.hasNext()) {
3   Foo obj = it.next();
4   it.remove(); // ВМЕСТО coll.remove(obj);
5 }
```

- Методът remove() не е задължителен и не всеки итератор го поддържа. Ако итераторът не притежава този метод, опита да го извикате генерира UnsupportedOperationException

Основни типове колекция

- Списък – `List`, `ArrayList`
- Множество – `Set`, `HashSet`, `TreeSet`
- Асоциативен контейнер – `Map`, `HashMap`

Списък

- Контейнер за данни (съхранява данни)
- Подредено, линейно множество от елементи
- Списъкът е динамична колекция – размерът на списъка е променлива
- Редът на елементите съвпада с реда на вмъкването им

```
1 List<String> strings = new ArrayList<String>();
2 strings.add("one");
3 strings.add("two");
4 strings.add("three");
5
6 // strings = [ “one”, “two”, “three”]
```

Списък: други операции

```
1  List<String> strings = new ArrayList<String>();
2
3  // Вмъкване след последния елемент
4  strings.add("one");
5  strings.add("three");
6
7  // Вмъкване на определена позиция
8  strings.add(1, "two");
9
10 // strings = [ “one”, “two”, “three”]
11
12 // Достигане до елемент на определена позиция:
13 System.out.println(strings.get(0));        // => “one”
14 System.out.println(strings.indexOf("one")); // => 0
```

Множество

- Колекция от данни
- Реализира математическата абстракция множество (set)
- Редът на вмъкване на елементите не влияе на редът им в множеството
- Не позволява вмъкването на два еднакви елемента (add() връща false):

```
1 Set<Name> names = new HashSet<Name>();
2 names.add(new Name("Jack", "Nance"));
3 names.add(new Name("Jack", "Nance"));
4
5 System.out.println(names.size()); // => 1
```

Изисквания към елементите на множеството

- След като даден елемент се вмъкне в множеството той не би следвало да бъде ползван по никакъв начин, който би го променило (неговия хеш код и данните за equal())
- Промяната на елемент от множеството би могло да доведе до неочаквани резултати:

```
1  Set<Name> names = new HashSet<Name>();
2  Name jack = new Name("Jack", "Nance");
3  names.add(jack);
4  System.out.println(names.size());          // => 1
5  System.out.println(names.contains(jack));// => true;
6
7  jack.last = "Vance";
8
9  System.out.println(names.contains(jack));// => false
10 System.out.println(names.size());          // => 1
```

Какво е решението на този проблем?

- Няма решение!
- Единственият начин да се спасите от този проблем е да не променяте състоянието на обекта след като го добавите в някое множество
- Най-добрият подход е обектите, които се съхраняват в множества или служат за ключове в асоциативни контейнери да бъдат непроменяеми (immutable)
- Ако все пак се налага да промените състоянието на обект, който е елемент на множество, първо трябва да го изтриете от множеството (за да се избегнат неочаквани резултати) и след като го промените да го добавите отново

Асоциативни контейнери (Map)

- Служи за съхранение на връзка между ключ и стойност
- На всеки ключ съответства стойност
- Удобно, когато трябва да се осъществи връзка между два обекта и по единия от тях да може бързо да се намери другия
- Ключовете трябва да са уникални
- Стойностите не е задължително да са уникални

Асоциативните контейнери не са колекции

- Асоциативният контейнер не поддържа методите на колекцията — **boolean** `add(Foo o)`, **boolean** `contains(Object obj);`
- Вместо тях асоциатвният контейнер разполага със следните операции:

```
1 boolean put(Foo key, Bar value);
2 boolean containsKey(Foo key);
3 boolean containsValue(Bar value);
```

Използване на асоциативни контейнери

```
Map<String, String> dns =
    new HashMap<String, String>();
```

- Вмъкване на ключ "lubo.elsys-bg.org" със стойност "82.147.133.129"

```
dns.put("lubo.elsys-bg.org",
    "82.147.133.129");

System.out.println(
    dns.get("lubo.elsys-bg.org"));
// => "82.147.133.129"
System.out.println(
    dns.containsKey("lubo.elsys-bg.org"));
// => true
System.out.println(
    dns.containsValue("82.147.133.129"));
// => true
```

Използване на асоциативни контейнери

- Изтриване на елемент по ключ "lubo.elsys-bg.org"

```
1 dns.remove("lubo.elsys-bg.org");
2 System.out.println(
3     dns.containsValue("82.147.133.129"));
4 // => false
```

Полезни методи на асоциативен контейнер

- `keySet()` — Връща множество (set, т.е. не може да има повторения) от всички ключове
- `values()` — Връща колекция (може да има повторения) от всички стойности
- `entrySet()` — Връща множество (set, т.е. не може да има повторения) от двойките ключ-стойност на асоциативния контейнер. Всяка двойка е обект от тип Map.Entry, който поддържа методи за достъп до ключа и стойността getKey(), getValue(), setValue()

Проблеми при промяна на ключа

- Не трябва да се променя състоянието на обект, който се използва като ключ в асоциативен контейнер
- Проблемът е напълно аналогичен на разглежданият при множествата
- Ако ключът бъде променен, ключът и съответстващата му стойност могат да се загубят

Проблеми при промяна на ключа

```
1  Name isabella = new Name("Isabella", "Rosellini");
2  Map<Name, String> directory =
3      new HashMap<Name, String>();
4  directory.put(isabella, "123-456-7890"); // добавяме
5  System.out.println(directory.get(isabella));
6
7  isabella.first = "Dennis"; // променяме ключа
8  System.out.println(directory.get(isabella));
9
10 // добавяме още един обект със стойност като оригиналния
11 directory.put(new Name("Isabella", "Rosellini"),
12     "555-555-1234");
13 // връщаме стойността на оригиналния
14 isabella.first = "Isabella";
15
16 System.out.println(directory.get(isabella));
```

Решение чрез копиране на ключа

```
1  Name dennis = new Name("Dennis", "Hopper");
2
3  // копиране на ключа:
4  Name copy = new Name(dennis.first, dennis.last);
5  map.put(copy, "555-555-1234"); // използва се копието
6
7  // промяната на dennis не би попречила
8
9  // но въпреки това остава опасност от промяна на ключа:
10 for (Name name : map.keySet()) {
11   name.first = "u␣r␣wrecked"; // ГРОЗНО!
12 }
```

Решение чрез непроменяем ключ

```
1  public  class Name {
2    public final String first; // т.е. не може да се променя
3    public final String last;
4
5    public Name(String first, String last) {
6      this.first = first;
7      this.last = last;
8    }
9
10   public boolean equals(Object o) {
11     return (o instanceof Name &&
12         ((Name) o).first.equals(this.first) &&
13         ((Name) o).last.equals(this.last));
14   }
15 }
```

Решение чрез непроменяем делегат

```
1 Map<String, String> dir =
2     new HashMap<String, String>();
3 Name naomi = new Name("Naomi", "Watts");
4
5 String key = naomi.first + "," + naomi.last;
6 dir.put(key, "888-444-1212");
```

- Обектите от типа String не могат да бъдат променяни.

Решение чрез замразяване на ключа

```
1  public  class Name {
2    private String first;
3    private String last;
4    private boolean frozen = false;
5
6    public void setFirst(String s) {
7      if (!frozen)
8        first = s;
9    }
10   // аналогично за setLast()
11
12   // „замразяваме“ обекта
13   public void freeze() {
14     frozen = true;
15   }
16 }
```

Променяеми ключове

- Всеки подход за решавне на проблемът с променяемите ключове има своите предимства и недостатъци
- Винаги използвайте възможно най-простото решение, което е подходящо за ситуацията
- Ако ключът не може да се променя, то няма да имате никакви проблеми

Чести грешки при работа с колекции

- Изтриване на елемент от списък докато го обхождаме
- Промяна на елемент в множество
- Промяна на ключ в асоциативен контейнер

Сравняване и сортиране

- Използва се за да се прецени кой от два обекта е по-голям или двата обекта са равни
- Изразът `a.compareTo(b)` връща следните стойности:
  - `<0` ако `a<b`
  - `==0` ако `a==b`
  - `>0` ако `a>b`

Пример: Сравнения

```
1 Integer one = 1;
2 System.out.println(one.compareTo(3));      // => -1
3 System.out.println(one.compareTo(-50));    // => 1
4
5 String frank = "Frank";
6 System.out.println(frank.compareTo("Booth"));  // => 4
7 System.out.println(frank.compareTo("Hopper")); // => -2
```

Пример: Азбучно сортиране на списък

```
1  List<String> names = new ArrayList<String>();
2  // добавяне на елементи към списъка
3  names.add("Sailor");
4  names.add("Lula");
5  names.add("Bobby");
6  names.add("Santos");
7  names.add("Dell");
8
9  // сортиране на списъка
10 Collections.sort(names);
11
12 // names => [ "Bobby "Dell "Lula "Sailor "Santos"]
```

Интерфейсът `Comparable`

- За да се сортират елементите на списъка трябва да имплементират интерфейса Comparable
- Методът, който трябва да бъде реализиран е:

```
1 int compareTo(Object obj);
```

- Класът `String` имплементира интерфейсът `Comparable` и поради това списък от низове може да бъде сортиран

Пример: Сортиране на класът Name

```
1  public  class Name implements Comparable<Name> {
2    // ...
3
4    // сортиране по фамилия
5    public int compareTo(Name o) {
6      int compare = this.last.compareTo(o.last);
7      if (compare != 0)
8        return compare;
9      else
10       return this.first.compareTo(o.first);
11   }
12 }
```

Пример: Сортиране на класът Name

```
1  List<Name> names = new ArrayList<Name>();
2  names.add(new Name("Nicolas", "Cage"));
3  names.add(new Name("Laura", "Dern"));
4  names.add(new Name("Harry", "Stanton"));
5  names.add(new Name("Diane", "Ladd"));
6  names.add(new Name("William", "Morgan"));
7  names.add(new Name("Dirty", "Glover"));
8  names.add(new Name("Johnny", "Cage"));
9  names.add(new Name("Metal", "Cage"));
10
11 System.out.println(names);
12 Collections.sort(names);
13 System.out.println(names);
14
15 // => [Johnny Cage, Metal Cage, Nicolas Cage, Laura Dern,
16 // Crispin Glover,Diane Ladd, William Morgan, Harry Stanton]
```

Интерфейсът `Comparator`

- Позволява едни и същи данни да бъдат сортирани по различни критерии
- Сравнява два обекта
- Методът, който трябва да се реализира е:

```
1 int compare(Object o1, Object o2);
```

Пример: Сортиране на класът Name по първо име

```
1  import java.util.Comparator;
2
3  public class FirstNameFirst
4        implements Comparator<Name> {
5    //сортиране по първо име
6    public int compare(Name n1, Name n2) {
7      int ret = n1.first.compareTo(n2.first);
8      if (ret != 0)
9        return ret;
10     else
11       return n1.last.compareTo(n2.last);
12   }
13 }
```

Пример: Сортиране на класът `Name` по първо име

```
1  List<Name> names = new ArrayList<Name>();
2  // ...
3
4  // създаваме обект, чрез който ще
5  // извършваме сравненията по време на сортиране
6  Comparator<Name> first = new FirstNameFirst();
7  Collections.sort(names, first);
8
9  System.out.println(names);
10
11 // => [Crispin Glover, Diane Ladd, Harry Stanton, JohnnyCage,
12 // Laura Dern, Metal Cage, Nicolas Cage, WilliamMorgan]
```

Изисквания към сравняване

- Резултатът при сравнението на два обекта винаги не трябва да се променя (ако обектите не се променят)!
- Особено внимание трябва да се обръща на ситуациите `(compare(e1, e2)==0) != e1.equals(e2)`. Такива ситуации трябва да се избягват.
- Такива случаи биха довели до неопределени резултати при сортирането на колекции като `SortedSet`, `SortedMap` и т.н.

Сортирани множества

- Ако искаме елементите в множеството да бъдат сортирани, можем да използваме имплементацията `TreeSet`.
- Елементите на множеството `TreeSet` трябва да имплементират интрефейсът `Comparable` или при създаване на множеството `TreeSet` трябва да подадете допълнителен обект, който имплементира интерфейсът `Comparator`

```
1 SortedSet<Name> names = new TreeSet<Name>(
2         new FirstNameFirst());
3 names.add(new Name("Laura", "Dern"));
4 names.add(new Name("Harry", "Stanton"));
5 names.add(new Name("Diane", "Ladd"));
6
7 System.out.println(names);
8 // => [Diane Ladd, Harry Stanton, Laura Dern]
```