Пишите компараторы правильно

для Примитивов особенно float u double

<https://www.toptal.com/developers/sorting-algorithms> - ссылка на страницу, где представлено визуальное определение скорости работы разных алгоритмов сортировки в зависимости от начального состояния отсортированности списка.

**Comparable** используется только для сравнения объектов класса, в котором данный интерфейс реализован. Т.е. interface Comparable определяет логику сравнения объекта определенного ссылочного типа внутри своей реализации (по правилам разработчика). Пишется в классе, объекты которого будут создаваться и перебираться в другом классе.

**Comparator** представляет отдельную реализацию и ее можно использовать многократно и с различными классами.

Писать Comparable в своих классах, я так понимаю, не нужно. Т.к. в случае необходимости той или иной сортировки твоего класса-сущности, пишется отдельный класс для Comparator-а и при создании той или иной коллекции просто вставляется (в скобках) при ее создании.

Два примера.

Для **Comparable**:

Customer.java

**package** org.o7planning.beans;

**public** **class** **Customer** **implements** **Comparable**<Customer> {

**private** String fullName;

**private** int loyaltyPoints;

**public** **Customer**(String fullName, int pointOfPurchase) { //

this.fullName = fullName;

this.loyaltyPoints = pointOfPurchase;

}

**public** String **getFullName**() {

**return** fullName;

}

**public** int **getLoyaltyPoints**() {

**return** loyaltyPoints;

}

@Override

**public** int **compareTo**(Customer other) {

**if** (other == null) {

**return** -1; // this < other

}

int delta = this.loyaltyPoints - other.loyaltyPoints;

**if** (delta != 0) {

**return** - delta;

}

**return** this.fullName.compareTo(other.fullName);

}

}

PriorityQueueEx2.java

**package** org.o7planning.priorityqueue.ex;

**import** java.util.PriorityQueue;

**import** org.o7planning.beans.Customer;

**public** **class** **PriorityQueueEx2** {

**public** **static** **void** **main**(String[] args) {

Customer tom = **new** **Customer**("Tom", 200);

Customer jerry = **new** **Customer**("Jerry", 50);

Customer donald = **new** **Customer**("Donald", 300);

Customer mickey = **new** **Customer**("Mickey", 30);

Customer daffy = **new** **Customer**("Daffy", 500);

PriorityQueue<Customer> queue = **new** **PriorityQueue**<>();

queue.add(tom);

queue.add(jerry);

queue.add(donald);

queue.add(mickey);

queue.add(daffy);

Customer currentCustomer = null;

**while**((currentCustomer = queue.poll())!= null) { // Retrieves and removes

System.out.println("--- Serving customer: " + currentCustomer.getFullName() + " ---");

System.out.println(" >> Loyalty Points: " + currentCustomer.getLoyaltyPoints());

System.out.println();

}

}

}

Output:

--- Serving customer: Daffy ---

>> Loyalty Points: 500

--- Serving customer: Donald ---

>> Loyalty Points: 300

--- Serving customer: Tom ---

>> Loyalty Points: 200

--- Serving customer: Jerry ---

>> Loyalty Points: 50

--- Serving customer: Mickey ---

>> Loyalty Points: 30

Для **Comparator**: Для элементов, которые не являются самосопоставимыми, необходимо предоставить ***Comparator*** для **PriorityQueue**.

* [Сравнение и Сортировка в Java](https://betacode.net/10213/comparing-and-sorting-in-java)

PriorityQueueEx3.java

**package** org.o7planning.priorityqueue.ex;

**import** java.util.Comparator;

**import** java.util.PriorityQueue;

**public** **class** **PriorityQueueEx3** {

**public** **static** **void** **main**(String[] args) {

Employee tom = **new** **Employee**("Tom", 2000);

Employee jerry = **new** **Employee**("Jerry", 500);

Employee donald = **new** **Employee**("Donald", 3000);

Employee mickey = **new** **Employee**("Mickey", 2000);

Employee daffy = **new** **Employee**("Daffy", 5000);

PriorityQueue<Employee> queue = **new** **PriorityQueue**<>(**new** **EmployeeComparator**());

queue.add(tom);

queue.add(jerry);

queue.add(donald);

queue.add(mickey);

queue.add(daffy);

Employee currentEmployee = null;

**while** ((currentEmployee = queue.poll()) != null) { // Retrieves and removes

System.out.println("--- Serving employee: " + currentEmployee.getFullName() + " ---");

System.out.println(" >> Salary: " + currentEmployee.getSalary());

System.out.println();

}

}

}

**class** **Employee** {

**private** String fullName;

**private** int salary;

**public** **Employee**(String fullName, int salary) {

this.fullName = fullName;

this.salary = salary;

}

**public** String **getFullName**() {

**return** fullName;

}

**public** int **getSalary**() {

**return** salary;

}

}

**class** **EmployeeComparator** **implements** **Comparator**<Employee> {

@Override

**public** int **compare**(Employee o1, Employee o2) {

**if** (o1 == o2) {

**return** 0;

}

**if** (o1 == null) {

**return** -1; // o1 < o2

}

**if** (o2 == null) {

**return** 1; // o1 > o2

}

int s = o1.getSalary() - o2.getSalary();

**if** (s != 0) {

**return** s;

}

**return** o1.getFullName().compareTo(o2.getFullName());

}

}

Output:

--- Serving employee: Jerry ---

>> Salary: 500

--- Serving employee: Mickey ---

>> Salary: 2000

--- Serving employee: Tom ---

>> Salary: 2000

--- Serving employee: Donald ---

>> Salary: 3000

--- Serving employee: Daffy ---

>> Salary: 5000

- Для сортировки массивов (так-же и ArrayList) через **Comparable** используется **Arrays.sort(arr),** через **Comparator** используется **Arrays.sort(arr, comparator)**.

- Для сортировки коллекций List через **Comparable** используется **Collections.sort(list)** (обратная сортировка Collections.sort(list,Collections.reverseOrder())), через **Comparator** используется **Collections.sort(list, comparator)** (обратная сортировка Collections.sort(list,Collections.reverseOrder(comparator))),для ArrayList можно написать еще **list.sort(comparator)**.

**-** Для сортировки коллекций **TreeMap/TreeSet** через **Comparable** ничего не нужно т.к. при заполнении колекция сортируется автоматически (обратная сортировка NavigableSet<String> obratSet = ss.descendingSet(); для **TreeMap** descendingKeySet()- возвращает ключи в обр.поряд., descendingMap() – возвращает ключ+значение отсортированные в обр.поряд. по значению), через **Comparator** сразу при создании коллекции объект компаратора пишется в скобках (Set<Flat> ss = new TreeSet<>(comparator)).

- Если для сортировки используется несколько Comparator, то в этом случае сортируется по Compar1, а если несколько значений окажутся одинаковыми, то они отсортируются по Compar2:

Comparator<Flat> cf = new Compar1().thenComparing(new Compar2());

Set<Flat> sss = new TreeSet<>(cf);

В Java для введения порядка среди определённых объектов можно написать компаратор — класс, содержащий метод compare, которая сравнивает два объекта.

Альтернативой компаратору является естественный порядок объектов: объект реализует интерфейс Comparable, который содержит метод compareTo, позволяющий сравнить этот объект с другим.

Сравнивающая функция должна вернуть 0, если объекты равны, отрицательное число (обычно -1), если первый объект меньше второго, и положительное число (обычно 1), если первый больше. Обычно реализация такой функции не представляет сложностей, но имеется один случай, о котором многие забывают.  
  
Сравнение используется различными алгоритмами от сортировки и двоичного поиска до поддержания порядка в сортированных коллекциях вроде TreeMap. Эти алгоритмы завязаны на три важных свойства сравнивающей функции: рефлексивность (сравнение элемента с самим собой всегда даёт 0), антисимметричность (сравнение A с B и B с A должны дать разный знак) и транзитивность (если сравнение A с B и B с C выдаёт одинаковый знак, то и сравнение A с C должно выдать такой же). Если сравнивающая функция не удовлетворяет этим свойствам, алгоритм может выдать совершенно непредсказуемый результат. Причём скорее всего вы не получите никакого исключения, просто результат будет неверный.  
  
Как обнаружилось, несоблюдение этих свойств — не такая уж редкая ситуация. Проблема возникает при сравнении вещественных чисел — float или double.  
  
Предположим, у нас имеется класс с полем типа double, и мы хотим упорядочивать объекты этого класса в зависимости от значения поля. Нередко можно встретить такой код:

public class DoubleHolder implements Comparable<DoubleHolder> {

double d;

public DoubleHolder(double d) {

this.d = d;

}

@Override

public int compareTo(DoubleHolder o) {

return d > o.d ? 1 : d == o.d ? 0 : -1;

}

@Override

public String toString() {

return String.valueOf(d);

}

}

У приведённого метода compareTo есть две проблемы. Первая — незначительная — он не различает +0.0 и -0.0: new DoubleHolder(-0.0).compareTo(new DoubleHolder(+0.0)) вернёт 0. Иногда это нестрашно, но в случае сортировки элементы с +0.0 и -0.0 расположатся в произвольном порядке, что будет смотреться некрасиво. Тем не менее, всё это мелочи по сравнению с NaN.

**NaN** ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) Not-a-Number, «нечисло») — одно из особых состояний [числа с плавающей запятой](https://ru.wikipedia.org/wiki/%D0%A7%D0%B8%D1%81%D0%BB%D0%B0_%D1%81_%D0%BF%D0%BB%D0%B0%D0%B2%D0%B0%D1%8E%D1%89%D0%B5%D0%B9_%D0%B7%D0%B0%D0%BF%D1%8F%D1%82%D0%BE%D0%B9).

К операциям, приводящим к появлению NaN в качестве ответа, относятся:

* все математические операции, содержащие NaN в качестве одного из операндов;
* [деление нуля на ноль](https://ru.wikipedia.org/wiki/%D0%94%D0%B5%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5_%D0%BD%D0%B0_%D0%BD%D0%BE%D0%BB%D1%8C);
* деление бесконечности на бесконечность;
* умножение нуля на бесконечность;
* сложение бесконечности с бесконечностью противоположного знака;
* вычисление квадратного корня отрицательного числа[[1]](https://ru.wikipedia.org/wiki/NaN#cite_note-1);
* логарифмирование отрицательного числа.

Число NaN (как в типе double, так и во float) — это довольно специфичная вещь. Оно не больше, не меньше и не равно никакому другому числу. В результате мы сразу получаем нарушение свойств сравнения:

DoubleHolder nan = new DoubleHolder(Double.NaN);

DoubleHolder zero = new DoubleHolder(0.0);

System.out.println("nan.compareTo(nan): "+nan.compareTo(nan)); System.out.println("nan.compareTo(zero): "+nan.compareTo(zero)); System.out.println("zero.compareTo(nan): "+zero.compareTo(nan));

nan.compareTo(nan): -1

nan.compareTo(zero): -1

zero.compareTo(nan): -1

Ни рефлексивности, ни антисимметричности не наблюдается. Можно встретить и такую реализацию сравнения:

public int compareTo(DoubleHolder o)

{ return d > o.d ? 1 : d < o.d ? -1 : 0; }

Здесь все три предыдущих сравнения выдадут ноль, то есть как будто бы свойства соблюдаются. Но, конечно, радоваться рано:

DoubleHolder nan = new DoubleHolder(Double.NaN);

DoubleHolder zero = new DoubleHolder(0.0);

DoubleHolder one = new DoubleHolder(1.0);

System.out.println("zero.compareTo(nan): "+zero.compareTo(nan)); System.out.println("nan.compareTo(one): "+nan.compareTo(one)); System.out.println("zero.compareTo(one): "+zero.compareTo(one));

zero.compareTo(nan): 0

nan.compareTo(one): 0

zero.compareTo(one): -1

Здесь нарушается транзитивность: первый объект равен второму, второй равен третьему, но первый третьему не равен.  
  
Чем же это грозит простому обывателю? Чтобы понять это, создадим простой список и попробуем его перемешать и посортировать несколько раз:

List<DoubleHolder> data = new ArrayList<>();

for(int i=1; i<=10; i++) {

data.add(new DoubleHolder(i));

}

data.add(new DoubleHolder(Double.NaN));

for(int i=0; i<10; i++) {

Collections.shuffle(data);

Collections.sort(data);

System.out.println(data);

}

Вывод в каждом запуске отличается и может выглядеть, например, так:

NaN, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0]

[NaN, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0]

[1.0, NaN, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0]

[NaN, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0]

[1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, NaN]

[2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, NaN, 1.0, 9.0, 10.0]

[1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 10.0, NaN, 9.0]

[NaN, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0]

[NaN, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0]

[1.0, NaN, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0]

Или для второй реализации compareTo:

[1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 8.0, 9.0, NaN, 7.0, 10.0]

[1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, NaN]

[1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, NaN]

[1.0, 2.0, 9.0, 10.0, NaN, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0]

[1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, NaN]

[1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, NaN]

[2.0, 6.0, NaN, 1.0, 3.0, 4.0, 5.0, 7.0, 8.0, 9.0, 10.0]

[2.0, 4.0, NaN, 1.0, 3.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0]

[1.0, 3.0, NaN, 2.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0]

[1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, NaN]

Примерно в половине случаев элемент с NaN прибивается к началу или концу сортируемого списка, а в других случаях начинает блуждать, портя порядок окружающих элементов.  
  
С коллекциями, использующими DoubleHolder в качестве ключа, тоже ничего хорошего не произойдёт. Возьмём, к примеру, TreeSet. Со второй реализацией compareTo всё довольно просто: так как элемент, содержащий NaN, равен любому другому, то в непустое множество вставить его не получится, потому что оно решит, что такой элемент уже есть. Но берегитесь, если вы вставили NaN-элемент первым: после этого во множество не выйдет добавить ничего другого.  
  
Первый вариант сравнивающей функции психоделичнее. Напишем, например, такой тест:

Set<DoubleHolder> set = new TreeSet<>();

for(int i=0; i<50; i++) {

set.add(new DoubleHolder( i%10==0 ? Double.NaN : i%10 )); // %-деление по модулю

}

System.out.println(set);

Мы вставили по пять элементов, содержащих NaN, и по пять элементов, содержащих каждую цифру от 1 до 9. В результате имеем следующее:

[NaN, NaN, 1.0, 2.0, 3.0, NaN, NaN, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, NaN]  
  
Вполне ожидаемо увидеть пять раз NaN: ведь они не равны друг другу. Но из-за неправильных сравнений и некоторые другие элементы вставились по нескольку раз. Можете сами посмотреть, что случится с TreeMap. Заметьте, что один случайно попавший NaN может испортить всю коллекцию, причём это не всегда легко отладить: коллекция может долго существовать в некорректном состоянии и делать вид, что всё нормально.  
  
Что любопытно, этой проблемы вообще не должно существовать. Ещё в JDK 1.4 появились специальные статические методы **Float.compare** **и Double.compare**, которые сделают всё за вас, корректно обработав специальные случаи. Надо лишь написать:

public int compareTo(DoubleHolder o) {

return Double.compare(d, o.d);

}

Видимо, сказывается обманчивая простота алгоритма сравнения: порой программист считает, что проще написать самому, чем искать в документации стандартный способ сделать это.  
  
Примеры неправильного сравнения double/float в различных открытых проектах: [JTS](http://grepcode.com/file/repo1.maven.org/maven2/com.vividsolutions/jts/1.13/com/vividsolutions/jts/index/intervalrtree/IntervalRTreeNode.java#72), [Batik](http://grepcode.com/file/repo1.maven.org/maven2/fr.avianey.apache-xmlgraphics/batik/1.8pre-20141006/org/apache/batik/gvt/flow/FlowRegions.java" \l "188), [Hadoop](http://grepcode.com/file/repo1.maven.org/maven2/org.apache.hadoop/hadoop-common/2.5.1/org/apache/hadoop/io/DoubleWritable.java" \l "78), [Hudson](http://grepcode.com/file/repo1.maven.org/maven2/org.eclipse.hudson/hudson-core/3.2.1/hudson/ExtensionComponent.java" \l "70), [ICU4J](http://grepcode.com/file/repo1.maven.org/maven2/com.ibm.icu/icu4j/54.1.1/com/ibm/icu/util/ULocale.java#2108), [Lucene](http://grepcode.com/file/repo1.maven.org/maven2/org.apache.lucene/lucene-smartcn/3.6.2/org/apache/lucene/analysis/cn/smart/hhmm/PathNode.java" \l "35). Трудно определить, в каких случаях это может привести к проблемам, но это тот случай, когда я бы исправлял безусловно: правильный и надёжный вариант обычно при этом ещё и короче неправильного.  
  
Чтобы изменить ситуацию, я [написал](https://code.google.com/p/findbugs/source/detail?r=ef65d4ee875d25690a5dcefbf34a1a7c378f1276) маленький детектор для FindBugs, который находит некорректно реализованные функции сравнения и предлагает использовать Float.compare/Double.compare.  
  
Вообще для всех примитивных типов есть подобные методы. Если вам надо сравнить несколько полей по очереди используя Comparable, можно написать так:

public class MyObject **implements** **Comparable**<MyObject> {

double d;

int i;

String s;

char c;

@Override

public int compareTo(MyObject o) {

int result;

result = Double.compare(d, o.d);

if(result != 0) return result;

result = Integer.compare(i, o.i);

if(result != 0) return result;

result = s.compareTo(o.s);

if(result != 0) return result;

result = Character.compare(c, o.c);

return result;

}

}

Смотрится лучше, чем куча веток с больше и меньше.

Если вы пользуетесь **[Guava](http://habrahabr.ru/post/244347/)** или чем-то подобным, тогда так:

@Override

public int compareTo(MyObject o) {

return ComparisonChain.start()

.compare(d, o.d)

.compare(i, o.i)

.compare(s, o.s)

.compare(c, o.c)

.result();

}

Еще один пример:

**public** **class** User **implements** Comparable <User>{//добавляем возможность сравнивать объекты User

**private** String name;

**private** Integer age;

**private** String email;

**public** User(String name, **int** age, String email) {

**this**.name = name;

**this**.age = age;

**this**.email = email;

}

@Override

//реализуем метод compareTo интерфейса Comparable

**public** **int** compareTo(User o) {

//используем метод compareTo из класса String для сравнения имен

**int** result = **this**.name.compareTo(o.name);

//если имена одинаковые - сравниваем возраст,

используя метод compareTo из класса Integer

**if** (result == 0) {

result = **this**.age.compareTo(o.age);

}

**return** result;

}

@Override

**public** String toString() {

**return** "{" +

"name='" + name + '\'' +

", age=" + age +

", email='" + email + '\'' +

'}';

}