Как отсортировать объекты в списке по дате в Java? | for-each.dev

1. Обзор

В этом руководстве мы обсудим сортировку объектов в списке по дате. Большинство методов или примеров сортировки позволяют пользователю сортировать список в алфавитном порядке, но в этой статье мы обсудим, как это сделать с объектами Date .

Мы рассмотрим использование класса Comparator в Java для **пользовательской сортировки значений наших списков** .

2. Настройка

Давайте посмотрим на сущность Employee , которую мы будем использовать в этой статье:

```
public class Employee implements Comparable<Employee> {

    private String name;
    private Date joiningDate;

    public Employee(String name, Date joiningDate) {
        // ...
    }

    // standard getters and setters
}
```

Мы можем заметить, что мы реализовали интерфейс Comparable в классе Employee . Этот интерфейс позволяет нам определить стратегию сравнения объектов с другими объектами того же типа. Это используется для **сортировки объектов в их естественной упорядоченной форме или определенной методом compareTo() .**

3. Сортировка с использованием Comparable

В Java естественный порядок относится к тому, как мы должны сортировать примитивы или объекты в массиве или коллекции. Метод **sort() в java.util.Arrays и java.util.Collections должен быть согласованным и отражать семантику равенства.**

Мы будем использовать этот метод для сравнения текущего объекта и объекта, переданного в качестве аргумента:

```
public class Employee implements Comparable<Employee> {

    // ...

    @Override
    public boolean equals(Object obj) {
        return ((Employee) obj).getName().equals(getName());
    }

    @Override
    public int compareTo(Employee employee) {
        return getJoiningDate().compareTo(employee.getJoiningDate());
    }
}
```

Этот метод compareTo() будет **сравнивать текущий объект с объектом, отправляемым в качестве параметра.** В приведенном выше примере мы сравниваем дату присоединения текущего объекта с переданным объектом Employee.

3.1. Сортировка по возрастанию

В большинстве случаев метод compareTo `()` **описывает логику сравнения объектов с естественной сортировкой.** Здесь мы сравниваем поле даты прихода сотрудника на работу с другими объектами того же типа. Любые два сотрудника вернут 0, если у них одинаковая дата присоединения:

```
@Test
public void givenEmpList_SortEmpList_thenSortedListinNaturalOrder() {
    Collections.sort(employees);
    assertEquals(employees, employeesSortedByDateAsc);
}
```

Теперь `Collections.sort(employees)` будет сортировать список сотрудников по `дате присоединения`, а не по первичному ключу или имени. Мы видим, что список отсортирован по `дате` присоединения сотрудников — теперь это становится естественным порядком для класса `Employee` :

```
[(Pearl,Tue Apr 27 23:30:47 IST 2021),
(Earl,Sun Feb 27 23:30:47 IST 2022),
(Steve,Sun Apr 17 23:30:47 IST 2022),
(John,Wed Apr 27 23:30:47 IST 2022)]
```

3.2. Сортировка по убыванию

Метод `Collections.reverseOrder()` сортирует **объекты, но в обратном порядке, в соответствии с естественным порядком.** Это возвращает компаратор, который будет выполнять упорядочение в обратном порядке. Он `выдаст исключение NullPointerException` , когда объект вернет значение `null` при сравнении:

```
@Test
public void givenEmpList_SortEmpList_thenSortedListinDescOrder() {
    Collections.sort(employees, Collections.reverseOrder());
    assertEquals(employees, employeesSortedByDateDesc);
}
```

4. Сортировка с использованием `компаратора`

4.1. Сортировка по возрастанию

Давайте теперь воспользуемся реализацией интерфейса `Comparator для сортировки нашего списка сотрудников`. Здесь мы на лету передадим анонимный параметр внутреннего класса API `Collections.sort()` :

```
@Test
public void givenEmpList_SortEmpList_thenCheckSortedList() {

    Collections.sort(employees, new Comparator<Employee>() {
        public int compare(Employee o1, Employee o2) {
            return o1.getJoiningDate().compareTo(o2.getJoiningDate());
        }
    });

    assertEquals(employees, employeesSortedByDateAsc);
}
```

Мы также можем заменить этот синтаксис на лямбда-синтаксис Java 8, который значительно уменьшит наш код, как показано ниже:

```
@Test
public void givenEmpList_SortEmpList_thenCheckSortedListAscLambda() {

    Collections.sort(employees, Comparator.comparing(Employee::getJoiningDate));

    assertEquals(employees, employeesSortedByDateAsc);
}
```

Метод `compare(arg1, arg2)` принимает **два аргумента универсального типа и возвращает целое число.** Поскольку оно отделено от определения класса, мы можем определить пользовательское сравнение на основе различных переменных и сущностей. Это полезно, когда мы хотим определить другую пользовательскую сортировку для сравнения объектов-аргументов.

4.2. Сортировка по убыванию

Мы можем отсортировать заданный список `сотрудников` в порядке убывания, обратив сравнение объектов сотрудников, т. е. сравнивая `Employee2` с `Employee1` . Это изменит сравнение и, таким образом, вернет результат в порядке убывания:

```
@Test
public void givenEmpList_SortEmpList_thenCheckSortedListDescV1() {

    Collections.sort(employees, new Comparator<Employee>() {
        public int compare(Employee emp1, Employee emp2) {
            return emp2.getJoiningDate().compareTo(emp1.getJoiningDate());
        }
    });

    assertEquals(employees, employeesSortedByDateDesc);
}
```

Мы также можем преобразовать приведенный выше метод в более краткие формы, используя лямбда-выражения Java 8. Это будет выполнять те же функции, что и вышеприведенная функция, с той лишь разницей, что код содержит меньше строк кода по сравнению с приведенным выше кодом. Хотя это также делает код менее читаемым. При использовании Comparator мы на лету передаем анонимный внутренний класс для API `Collections.sort()` :

```
@Test
public void givenEmpList_SortEmpList_thenCheckSortedListDescLambda() {

    Collections.sort(employees, (emp1, emp2) -> emp2.getJoiningDate().compareTo(emp1.getJoiningDate()));
    assertEquals(employees, employeesSortedByDateDesc);
}
```

5. Вывод

В этой статье мы рассмотрели, как сортировать коллекцию Java по объекту `Date` как в восходящем, так и в нисходящем режимах.

Мы также кратко рассмотрели лямбда-функции Java 8, которые полезны при сортировке и помогают сделать код кратким.

Как всегда, полные примеры кода, используемые в этой статье, можно найти на GitHub .