universität innsbruck

Comparator und Comparable

Programmiermethodik

Lukas Kaltenbrunner, Simon Priller
Universität Innsbruck

Vergleiche von Objekten

- Für beliebige Elementtypen ist aber nicht ohne weiteres klar, welches von zwei Objekten das „größere“ ist.
- Viele Methoden (z.B. `sort`, `binarySearch` der `Collections`-Klasse) vergleichen Elemente, um sie sortieren zu können.

- Zwei Möglichkeiten:
  - Interface `Comparable<T>` implementieren
  - Klasse, die `Comparator<T>` implementiert, verwenden

Comparable

- Durch das Implementieren des generischen Interface `Comparable<T>` zeigt eine Klasse an, dass es eine natürliche Ordnung für Objekte der Klasse gibt.
- Das Interface enthält nur die Vergleichsmethode `compareTo`.

```
public interface Comparable<T> {
    int compareTo(T o);
}
```

compareTo (1)

- Gibt für den Aufruf `x.compareTo(y)` zurück:
  - `< 0` falls x < y ist.
  - `0` falls x == y ist.
  - `> 0` falls x > y ist.
- Wirft eine `NullPointerException`, falls der übergebene Parameter `null` ist.
- Im Unterschied zu `equals` ist aufgrund der Generizität des `Comparable` Interfaces kein Cast nötig (Kompilierfehler bei Typinkompatibilität).

compareTo (2)

- `sgn(x)` entspricht der Signum-Funktion.

$$\operatorname{sgn}(x) = \begin{cases} -1 & \text{falls} \quad x < 0 \\ 0 & \text{falls} \quad x = 0 \\ +1 & \text{falls} \quad x > 0 \end{cases}$$

- Eine korrekte Implementierung von `compareTo` erfüllt:
  - `sgn(x.compareTo(y)) == -sgn(y.compareTo(x))`
    - Impliziert `x.compareTo(y)` darf nur genau dann dann eine Exception werfen, wenn `y.compareTo(x)` auch eine wirft.
  - Transitivität
    Beispielsweise: `(x.compareTo(y) > 0 && y.compareTo(z) > 0)` dann gilt auch `x.compareTo(z) > 0`
  - Wenn `x.compareTo(y) == 0` dann `sgn(x.compareTo(z)) == sgn(y.compareTo(z))` für beliebige z.
- Empfehlung: `(x.compareTo(y) == 0) == x.equals(y)`
  Gleichheit sollte mit `equals` übereinstimmen.
  - Abweichungen sollten dokumentiert werden!

Comparable – Anmerkungen

- Für das Vergleichen von Referenztypen, sollte die `compareTo` Methode rekursiv aufgerufen werden.
- Falls eine von der Standardsortierung abweichende Sortierung benötigt wird, soll ein `Comparator` verwendet werden.
- Bei primitiven Datentypen sollten die statischen Vergleichsmethoden der Wrapper-Typen verwendet werden.
  - Implementierungsfehler können dadurch vermieden werden.
  - Beispiel: `Integer.compare(i1, i2)` anstatt `i1 - i2`
- Falls eine Klasse mehrere signifikante Objektvariablen hat:
  - Mit dem Vergleich der signifikantesten Objektvariable beginnen.
  - Schrittweise zur am wenigsten signifikanten Objektvariable durcharbeiten.
- Comparable als Typebound:
  - Comparable konsumiert immer Elemente (PECS).
  - `Comparable<? super T>` sollte im Allgemeinen `Comparable<T>` bevorzugt werden.

Beispiel Comparable - Product

```
public class Product implements Comparable<Product> {
    …
    public Product(int productID, String name) {
        this.productID = productID;
        this.name = name;
    }
    …
    @Override
    public int compareTo(Product o) {
        return Integer.compare(productID, o.productID);
    }
    …
}
```

```
public static void main(String[] args) {
    List<Product> shoppingList = new ArrayList<>();
    shoppingList.add(new Product(5, "Chocolate"));
    shoppingList.add(new Product(1, "Milk"));
    shoppingList.add(new Product(4, "Potato"));
    shoppingList.add(new Product(3, "Mango"));

    Collections.sort(shoppingList);
    System.out.println(shoppingList);
}
```

Ausgabe:

```
[Product{productID=1, name='Milk'}, Product{productID=3, name='Mango'},
Product{productID=4, name='Potato'}, Product{productID=5, name='Chocolate'}]
```

src/at/ac/uibk/pm/comparable/product/

Beispiel Comparable - Rectangle

```
@Override
public int compareTo(Rectangle o) {
    int result = Integer.compare(width, o.width);
    if (result != 0) {
        return result;
    }
    return Integer.compare(length, o.length);
}
```

```
public static void main(String[] args) {
    List<Rectangle> rectangles = new ArrayList<>();
    rectangles.add(new Rectangle(10, 10));
    rectangles.add(new Rectangle(10, 4));
    rectangles.add(new Rectangle(20, 4));

    Collections.sort(rectangles);
    System.out.println(rectangles);

}
```

Ausgabe:

```
[Rectangle{width=10, length=4}, Rectangle{width=10, length=10}, Rectangle{width=20, length=4}]
```

src/at/ac/uibk/pm/comparable/rectangle/

Comparator

- Oft sollen Objekte nach verschiedenen Kriterien verglichen werden.
- Daher sind viele Collection-Methoden mit einem zusätzlichen Parameter, einem Comparator, überladen.
  - Dient wie Comparable zum Vergleich von Exemplaren.
  - Zum Vergleich wird aber das übergebene Comparator-Objekt benutzt.
  - Kann auch für nicht selbst implementierte Klassen verwendet werden.
- `Comparator<T>` ist ein generisches Interface mit einer Methode `compare`.
  - Methode akzeptiert zwei Objekte des Elementtyps und liefert ein `int`-Ergebnis für den Vergleich der zwei Objekte.

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

compare (1)

- Gibt zurück:
  - `< 0` wenn `o1 < o2` ist.
  - `0` wenn `o1 == o2` ist.
  - `> 0` wenn `o1 > o2` ist.
- Wirft eine `NullPointerException` falls einer der übergebenen Parameter `null` ist und `null` nicht erlaubt ist.

compare (2)

- sgn(x) entspricht der Signum-Funktion.

$$\operatorname{sgn}(x) = \begin{cases} -1 & \text{falls } x < 0, \\ 0 & \text{falls } x = 0, \\ 1 & \text{falls } x > 0. \end{cases}$$

- Eine korrekte Implementierung von `compare` erfüllt:
  - `sgn(compare(x, y)) == -sgn(compare(y, x))`
    - Impliziert `compare(x, y)` darf nur genau dann dann eine Exception werfen, wenn `compare(y, x)` auch eine wirft.
  - Transitivität
    `(compare(x, y) > 0 && compare(y, z) > 0)` dann gilt auch `compare(x, z) > 0`
  - Wenn `compare(x, y) == 0` dann
    - `sgn(compare(x, z)) == sgn(compare(y, z))` für beliebige z.
- Empfehlung: `(compare(x, y) == 0) == x.equals(y)` Gleichheit sollte mit `equals` übereinstimmen!
  - Abweichungen sollten dokumentiert werden!

Comparator – Implementierung (1)

- Soll dazu dienen, verschiedene Sortierungen vornehmen zu können, da dieser der sort-Methode übergeben werden kann (und nicht in der zu sortierenden Klasse fix spezifiziert ist).
- Möglichkeit 1: Eigene Klasse für jeden Comparator (unübersichtlich, viele kleine Klassen)

```
public class ProductNameComparator implements Comparator<Product> {
    @Override
    public int compare(Product o1, Product o2) {
        return o1.getName().compareTo(o2.getName());
    }
}
```

```
shoppingList.sort(new ProductNameComparator());
System.out.println(shoppingList);
```

Ausgabe:

```
[Product{productID=5, name='Chocolate'}, Product{productID=3, name='Mango'},
Product{productID=1, name='Milk'}, Product{productID=4, name='Potato'}]
```

Comparator – Implementierung (2)

- Möglichkeit 2: Innere Klasse

```
shoppingList.sort(new Comparator<Product>() {
    @Override
    public int compare(Product p1, Product p2) {
        return p1.getName().compareTo(p2.getName());
    }
});
```

- Möglichkeit 3: Lambda Ausdruck

```
shoppingList.sort((p1, p2) -> p1.getName().compareTo(p2.getName()));
```

```
shoppingList.sort(Comparator.comparing(Product::getName));
```

Quellen

- Christian Ullenboom: **Java ist auch eine Insel: Einführung, Ausbildung, Praxis**, Rheinwerk Verlag, 16. Auflage, 2022 (Java 17)
- Michael Inden: **Der Weg zum Java-Profi: Konzepte und Techniken für die professionelle Java-Entwicklung**, dpunkt.verlag, 5. Auflage, 2021
- Joshua Bloch: **Effective Java**, Addison-Wesley Professional, 3. Auflage, 2018