Inhalt

Motivation

- Szenario: (sehr) große Datenmenge
- Aufgabe: Suche eines Elements mit gegebenem Wert, zB:
  boolean contains(T e)
- Aufwand (in allen bisher bekannten Datenstrukturen): $\mathcal{O}(n)$

Der Aufwand könnte erheblich reduziert werden, wenn der Datenbestand nach Werten des Typs T sortiert vorliegen würde!

dazu notwendig:

- Datensätze miteinander vergleichen können
- Daten des zugrunde liegende Typs T sollten „vergleichbar“ sein

Schnittstelle Comparable<T>

Das (API-)Interface Comparable<T> fordert eine Vergleichsfunktion:

```
int compareTo(T o)
```

Mittels dieser Methode kann „kleiner“, „gleich“ und „größer“ für den Typ T definiert werden:

Der Aufruf `a.compareTo(b)` liefert

- einen negativen Wert, gdw a „kleiner“ als b
- den int-Wert 0, gdw a „gleich“ b
- einen positiven Wert, gdw a „größer“ als b

Die Methode compareTo(T e)

Die Java-Dokumentation formuliert folgende Bedingungen an die Implementierung:

- Es ist „dringend empfohlen“ die Methode so zu implementieren, dass sie „konsistent“ mit equals ist, dass also
  $a.\text{compareTo}(b) == 0 \Leftrightarrow a.\text{equals}(b)$

Es ist sicher zu stellen, dass für alle $a, b, c \in T$ gilt:

- $a.\text{compareTo}(b) < 0 \Leftrightarrow b.\text{compareTo}(a) > 0$
- $(a.\text{compareTo}(b) < 0 \land b.\text{compareTo}(c) < 0) \Rightarrow a.\text{compareTo}(c) < 0$
- $a.\text{compareTo}(b) == 0 \Rightarrow (a.\text{compareTo}(c) < 0 \Leftrightarrow b.\text{compareTo}(c) < 0)$

Natürliche Ordnung

- Durch die oben genannten Bedingungen induziert die compareTo()-Methode eine (totale) Ordnungsrelation auf den Objekten des Datentyps.
- Diese Ordnungsrelation heißt „innere Ordnung“ oder „natürliche Ordnung“ des Typs T.
- Klassen, die das Interface Comparable<T> implementieren, heißen auch „natürlich geordnet“.

Beispiele:
Folgende Klassen der Java-Bibliothek haben eine natürliche Ordnung:

- die Wrapper-Klassen Double, Float, Integer ...
  Ordnung entspricht der üblichen $\leq$-Ordnung
- String: Ordnung im lexikographischen Sinne

Beispiel: Klasse Stud

Wir wollen Stud-Objekte nach ihrer Matrikelnummer vergleichen:

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

    private String name;
    private int matNr;

    // ...

    @Override
    public int compareTo(Stud s) {
        return this.matNr - s.matNr;
    }
}
```

Inhalt

Funktionales Interface Comparator<T>

- Das Interface Comparator<T> definiert (ausschließlich) die Methode compare(T t, T s)
- Die Methode compare(T t, T s) hat die gleiche Semantik wie t.compareTo(s) der Schnittstelle Comparable<T>,
- das heißt: für einen Komparator c und zwei T-Objekte a und b liefert der Aufruf c.compare(a, b)
  - einen negativen Wert, gdw a „kleiner“ als b
  - den int-Wert 0, gdw a „gleich“ b
  - einen positiven Wert, gdw a „größer“ als b
- Auch diese Methode „sollte“ verträglich mit equals() sein, dh es sollte a.equals(b) immer denselben Wahrheitwsert liefern wie c.compare(a, b) (für jeden Comparator c).
- Auch diese Methode induziert eine Ordnungsrelation auf dem Datentyp T.

Comparable vs Comparator

Unterschiede zur Schnittstelle Comparable<T>

- Durch einen Komparator kann einer Klasse eine „äußere Ordnung“ gegeben werden, auch wenn die Klasse keine „innere“ Ordnung hat.
- Einer Klasse mit innerer (natürlicher) Ordnung können weitere alternative Ordnungskriterien gegeben werden.

Die beiden Schnittstellen Comparable<T> und Comparator<T> mit ihren beiden Methoden t.compareTo(s) bzw c.compare(a, b) stellen alternative Möglichkeiten dar, Objekte miteinander zu vergleichen.

Sie sind nicht - wie die Schnittstellen Iterator <T> und Iterable <T> - einander ergänzende Konzepte.

Beispiel MinimumSuche

Mit innerer Ordnung (denn Integer **implements** Comparable<Integer>)

```
public static Integer min(Integer[] a) {
    Integer m = a[0];
    for(int i = 1; i < a.length; i++)
        if(a[i].compareTo(m) < 0)
            m = a[i];
    return m;
}
```

Mit äusserer Ordnung

```
public static <T> T min(T[] a, Comparator<T> c) {
    T m = a[0];
    for(int i = 1; i < a.length; i++)
        if(c.compare(a[i], m) < 0)
            m = a[i];
    return m;
}
```

Anwendung auf Personen (1)

Personen alphabetisch nach Namen vergleichen:
Nutze die natürliche Ordnung von Strings

```
public class NameComparator implements Comparator<Person> {
    @Override
    public int compare(Person o1, Person o2) {
        return o1.name().compareTo(o2.name());
    }
}
```

Personen nach Alter vergleichen (min = ältester):

```
public class GebJahrComparator implements Comparator<Person> {
    @Override
    public int compare(Person o1, Person o2) {
        return o1.gebJahr() - o2.gebJahr();
    }
}
```

Anwendung auf Personen (2)

Nutzung der Komparatoren:

```
Person d = new Person("Deniz", 1987)
Person a = new Person("Anna", 2000);
Person[] pArr = {a, d};
Person erster = min(pArr, new NameComparator());
Person aeltester = min(pArr, new GebJahrComparator());
```

Inhalt

Motivation: Minimumsuche

schon gesehen: „Minimum“ von Arrays verschiedener Basistypen

- int-Arrays, Integer-Arrays, Stud-Arrays . . .:
  nutzen die natürliche Ordnung von int bzw Integer
- Person-Arrays:
  nutzen äußere Ordnung durch Übergabe externer Komparatoren

Frage/Ziel: (Wie) Lässt sich diese Methode auf Dynamische Arrays übertragen
ohne externe Komparatoren übergeben zu müssen?

Warum ist das keine Lösung?

```
public static <T> T min(DynArray<T> arr) {
    T m = arr.get(0);
    for(int i = 1; i < arr.size(); i++) {
        if (arr.get(i).compareTo(m) < 0)
            m = arr.get(i);
    }
    return m;
}
```

Es müsste garantiert werden können,

- dass es für den Typ T eine compareTo()-Methode gibt
- also dass der Typ T das Interface Comparable<T> implementiert
- also dass T ein Untertyp von Comparable<T> ist

Typeinschränkung mit extends

Genau das leistet die Typeinschränkung:

```
public static <T extends Comparable<T>> T min(DynArray<T> arr) {
    T m = arr.get(0);
    for(int i = 1; i < arr.size(); i++) {
        if (arr.get(i).compareTo(m) < 0)
            m = arr.get(i);
    }
    return m;
}
```

- Der Ausdruck <T **extends** Comparable<T>> schränkt die für T erlaubten Argumente ein
- auf Typen, die über eine compareTo()-Methode verfügen.

Einschränkung auf mehrere Obertypen

Es kann auch erforderlich sein, dass ein Typ T mehrere Bedingungen erfüllen muss, zB

- Unterklasse einer Klasse K zu sein
- ein Interface *Ifc*1 zu implementieren
- ein Interface *Ifc*2 zu implementieren

Die (nicht-generische) Klasse C würde diese Bedingungen erfüllen:

```
class C extends K implements Ifc1, Ifc2 {
    // ...
}
```

Generisch: Forderung an das Typargument T:

```
class  GenClass<T extends K & Ifc1 & Ifc2>  {
    // ...
}
```