10.9 Sortiert vs unsortiert

Java hat Mechanismen, die viel verwendete Typen (Zahlen, strings) so ordnet, wie man es erwarten würde.

Die Interfaces `SortedSet` und `SortedMap` garantieren diese Ordnung (z.B. beim Durchlaufen per Iterator). Die Implementationen dazu sind `TreeSet` und `TreeMap`.

```
public static void main ( String[] args )
{
    Set<Double> doubles = new HashSet<Double>() ;
    //      SortedSet<Double> doubles = new TreeSet<Double>() ;
    for ( int ii=0 ; ii<10 ; ii++ ) {
        double rnd = Math.random() ;
        System.out.println ( " rnd: " + rnd ) ;
        doubles.add( new Double(rnd) ) ;
    }
    for ( Double dbl : doubles )
    {
        System.out.println ( dbl) ;
    }
}
```

Run this with `HashSet` (not sorted):

```
javac TreeTest.java
java TreeTest
 rnd: 0.5742673601258754
 rnd: 0.6379131747799471
 rnd: 0.9470754646268826
 rnd: 0.48257207437543337
 rnd: 0.48907917638636467
 rnd: 0.9690572075291323
 rnd: 0.7017416146968838
 rnd: 0.6280795517955831
 rnd: 0.35759262379150636
 rnd: 0.3874256327597535
0.6379131747799471
0.48257207437543337
0.6280795517955831
0.48907917638636467
0.5742673601258754
0.9470754646268826
0.7017416146968838
0.3874256327597535
0.35759262379150636
0.9690572075291323
```

Run this with `TreeSet` (sorted):

```
javac TreeTest.java
java TreeTest
 rnd: 0.9229000667290833
 rnd: 0.6506446530811612
 rnd: 0.19893116548115397
 rnd: 0.8883119797221786
 rnd: 0.32339531020019785
 rnd: 0.20124376468228355
 rnd: 0.2693512615983832
 rnd: 0.01869731295031296
 rnd: 0.541463282473842
 rnd: 0.18872774571163498
0.01869731295031296
0.18872774571163498
0.19893116548115397
0.20124376468228355
0.2693512615983832
0.32339531020019785
0.541463282473842
0.6506446530811612
0.8883119797221786
0.9229000667290833
```

Bem:

- Eigentlich wechseln wir oben nicht das Interface, sondern die Implementation.

  Wir wechseln von einer Implementation, die nichts über Ordnung sagt, auf eine, welche eine Ordnung einhält.

  In dem Sinne garantiert `SortedSet` eigentlich nur weitere Interface-Funktionen (s. Javadoc), die wir aber gar nicht nutzen.

  Allerdings ist es nicht möglich, `HashSet` als Implementation von `SortedSet` zu nehmen.

  Man kann sich, bis zu einem gewissen Grade, also doch durch die Wahl des Interfaces bzgl. bestimmter Eigenschaften der Implementation versichern.

10.10 Algorithmen: shuffle

```
        public static void main ( String[] args ) {
                List agents = new ArrayList() ;
                for ( int ii=0 ; ii<10 ; ii++ ) {
                        Agent ag = new Agent(ii) ;
                        agents.add( ag ) ;
                }
                printAllAgentIds(agents) ;
                Collections.shuffle(agents) ;
                printAllAgentIds(agents) ;
                Collections.shuffle(agents) ;
                printAllAgentIds(agents) ;

        }
```

results in

```
0 1 2 3 4 5 6 7 8 9
9 6 5 3 0 4 8 2 1 7
5 3 8 4 0 2 6 9 7 1
```

`Collections.shuffle(...)` randomisiert die Reihenfolge, in der die Elemente im Container sind. (Macht nur Sinn bei Containern, die intern nicht sortiert sind, also gerade kein `SortedXxx`.)

10.11 Algorithmen: sort

10.11.1 Beispiel

```
class Sort {
    public static void main(String args[]) {
        List l = Arrays.asList(args);
        Collections.sort(l);
        System.out.println(l);
    }
}
```

und dann

```
% java Sort i walk the line

[i, line, the, walk]
```

Bem.:

- *Nicht* `l.sort()`.

  Das ist vielleicht unerwartet, lässt sich aber begründen:

  - `Collections.sort(List ll)` nimmt jede Klasse als input, welche das `List` Interface implementiert.
  - Somit ist auch jede neue Implementation von `List` *automatisch* sortierbar, ohne dass man dafür etwas tun muss!!
  - Vielleicht noch wichtiger: Jeder *neue* Algorithmus, welcher `List` als Input nimmt, funktioniert automatisch für alle bereits vorhandenen Implementationen von `List`, ohne dass man diese Implementationen nochmal anfassen muss.

10.11.2 Comparable interface

```
class Agent implements Comparable<Agent>
{
    private int id ;
    public int getId() { return this.id ; }

    private double age ;
    public double getAge() { return this.age ; }

    public Agent ( int id, double age ) { this.id = id ; this.age = age ; }

    public int compareTo ( Agent otherAg )
    {
        if ( this.getId() > ag.getId() ) {
            return 1 ;
        } else if ( this.getId() == ag.getId() ) {
            return 0 ;
        } else {
            return -1 ;
        }
    }
}
```

sowie

```
class ComparableTest
{
    public static void main( String[] args ) {
        List<Agent> agents = new ArrayList<>() ;

        System.out.println("\ngenerate agents with random id/age ...") ;
        for ( int ii=0 ; ii<10 ; ii++ ) {
            Agent ag = new Agent( (int)(100000.*Math.random()), Math.random() ) ;
            agents.add(ag) ;
        }

        System.out.println("\ndemonstrate that id is in rnd sequence:") ;
        for ( Agent ag : agents ) {
            System.out.println( " agentId: " + ag.getId() ) ;
        }

        System.out.println( "\nsort the agents ... " ) ;
        Collections.sort( agents ) ;

        System.out.println("\ndemonstrate that they are now sorted:") ;
        for (Agent ag : agents ) {
            System.out.println( " agentId: " + ag.getId() ) ;
        }
    }
}
```

generates output

```
generate agents with random id/age ...

demonstrate that id is in rnd sequence:
 agentId: 6714
 agentId: 55239
 agentId: 41544
 agentId: 80092
 agentId: 20564
 agentId: 97537
 agentId: 32367
 agentId: 72901
 agentId: 53261
 agentId: 93272

sort the agents ...

demonstrate that they are now sorted:
 agentId: 6714
 agentId: 20564
 agentId: 32367
 agentId: 41544
 agentId: 53261
 agentId: 55239
 agentId: 72901
 agentId: 80092
 agentId: 93272
 agentId: 97537
```

Interpretation: Eine Klasse, die `Comparable` implementiert, hat eine natürliche Ordnung, die halt gerade durch Comparable gegeben wird.

Wenn man `compareTo` nicht implementiert, oder `implements Comparable` nicht dranschreibt, dann geht's nicht. **[[show]]**

Das ist (natürlich) auch die gleiche Struktur, auf der z.B. `SortedSet` beruht.

10.11.3 Comparator interface

Kann aber passieren, dass man es nicht natürlich sortieren will, sondern nach einem anderen Kriterium. In unserem Beispiel z.B. nach “age”. Dann muss man den “comparator” angeben:

```
import java.util.* ;
class ComparatorTest {
  public static void main( String[] args ) {
    List<Agent> agents = new ArrayList<>() ;

    System.out.println( "\ngenerate agents with random id/age ..." ) ;
    for ( int ii=0 ; ii<10 ; ii++ ) {
      Agent ag = new Agent( (int)(100000.*Math.random()), Math.random() );
      agents.add(ag) ;
    }

    System.out.println( "\ndemonstrate that id/age are in rnd sequence: " ) ;
    for ( Agent ag : agents ) {
      System.out.println( "agentId: "+ag.getId()+" age: "+ag.getAge() ) ;
    }

    System.out.println( "\nsort the agents according to comparator ..." ) ;
    Collections.sort( agents, cmp ) ;  // compile error --> use quick fix
    //                        ^^^

    System.out.println( "\nprint out the agents: " ) ;
    for ( Agent ag : agents ) {
      System.out.println( " id: "+ag.getId()+" age: "+ag.getAge() ) ;
    }
  }
}
```

"quick fix" will introduce `Comparator<? super Agent> cmp`. Da wir `? super` nicht verstehen, brauchen wir es vermutlich auch nicht, und löschen es daher. Anschließend vervollständigen wir

```
Comparator<Agent> cmp = new MyAgentComparator() ;
```

Anschließend wieder per "quick fix" Erstellung der fehlenden Klasse, und dann per "quick fix" Erstellung der unimplemented methods.

Den Altersvergleich muss man anschließend tatsächlich selber programmieren:

```
import java.util.* ;
final class MyAgentComparator implements Comparator<Agent> {
    public int compare (Agent a1, Agent a2 ) {
        if ( a1.getAge() > a2.getAge() ) {
            return 1 ;
        } else if ( a1.getAge() == a2.getAge() ) {
            return 0 ;
        } else {
            return -1 ;
        }
    }
}
```

Ergibt ...

```
generate agents with random id/age ...

demonstrate that id/age are in rnd sequence:
 agentId: 89790 age: 0.0090816424041064 71
 agentId: 83720 age: 0.9791337592250887
 agentId: 17771 age: 0.9220664934169156
 agentId: 98368 age: 0.9639119481446523
 agentId: 12921 age: 0.3660989342017913
 agentId: 25834 age: 0.7697270453658757
 agentId: 68783 age: 0.750712047387106
 agentId: 87519 age: 0.838813734046214
 agentId: 72795 age: 0.5303931837440602
 agentId: 29467 age: 0.8549615750730793

sort the agents according to comparator ...

print out the agents:
 id: 89790 age: 0.0090816424041064 71
 id: 12921 age: 0.3660989342017913
 id: 72795 age: 0.5303931837440602
 id: 68783 age: 0.750712047387106
 id: 25834 age: 0.7697270453658757
 id: 87519 age: 0.838813734046214
 id: 29467 age: 0.8549615750730793
 id: 17771 age: 0.9220664934169156
 id: 98368 age: 0.9639119481446523
 id: 83720 age: 0.9791337592250887
```

Wie funktioniert das?

Referenz auf eine Instanz einer Klasse, die nur die gewünschte Funktion enthält:

- `Collections.sort( agents, ... )` erhält als zweites Argument eine Klasse, die das Comparator-Interface erfüllt.
- Damit ist sichergestellt, dass die Methode ....compare(obj1, obj2) existiert.
- Der Sortier-Algorithmus ist nun so gebaut, dass diese Methode zum Vergleichen beim Sortieren benutzt wird.

Man kann (natürlich) auch die sortierten Collections nach einer anderen als der natürlichen Ordnung sortieren, z.B.

```
SortedSet<...> sortedSet = new HashSet<>( new MyComparator() ) ;
```

10.12 Algorithmen: binarySearch

Im Prinzip:

- Fülle die Collection `coll` mit beliebigem Inhalt.
- Sortiere die Collection via `Collections.sort(coll)`, evtl. mit entsprechendem Comparator.
- Suche nach einem Element mit Hilfe von

  `Collections.binarySearch( coll, key )` .

Das ist z.B. brauchbar in folgender Situation:

- Es existiert eine sinnvolle natürliche Ordnung von Objekten. (Z.B. Kanten in Verkehrsgraphen geordnet nach ID.)
- Man greift auf die Objekte u.a. über den dadurch definierten Schlüssel (key) zu.

Der vielleicht offensichtliche Ansatz ist mit `Map`. Zugriff dann über

```
Link link = coll.get("123")  // Loesung mit "Map"
```

Mit binarySearch stattdessen:

```
...
Collections.sort( coll ) ;
...
Link link = Collections.binarySearch( coll, new Agent("123") ) ;
// (Loesung mit binarySearch)
...
```

Die Lösung mit binarySearch ist tendenziell langsamer beim Zugriff per "key", aber möglicherweise signifikant sparsamer im Speicherverbrauch, sowie tendenziell schneller für andere Zugriffe.

10.13 Allgemeiner Aufbau der Java Collections

Java Collections besteht im Wesentlichen aus drei Teilen: **Interfaces** (einschl. **Iteratoren**), **Implementationen** und **Algorithmen**.

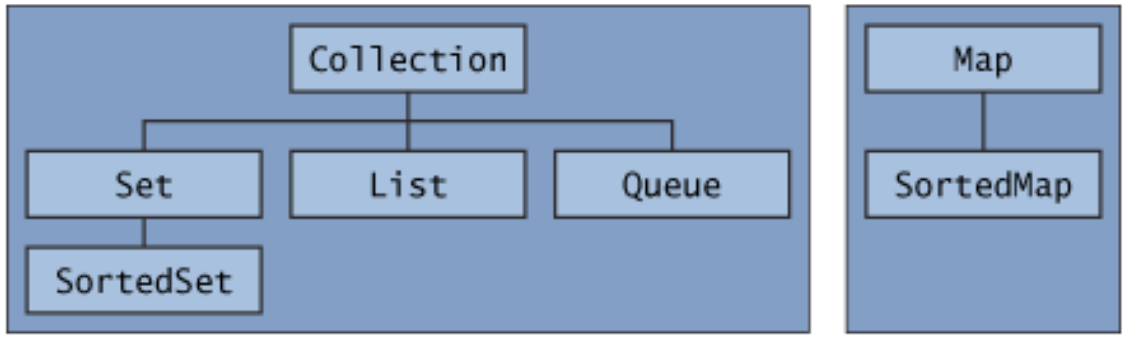
1. **Interfaces** definieren Zugänge zu Datenstrukturen, die dazu dienen, grössere Mengen von Elementen zu verwalten.

   Insbesondere: **Iteratoren** sind Konstrukte, die man sich im weitesten Sinne als Zeiger vorstellen kann. Sie werden z.B. dazu verwendet, eine Aktion auf alle Elemente einer Collection anzuwenden.

2. **Implementationen** sind praktische Implementierungen davon. Sie nehmen dem Programmierer die Arbeit ab, sich um jedes Detail der Speicherverwaltung kümmern zu müssen.

3. **Algorithmen** bestehen aus Routinen, die alle auf Collections arbeiten. Sie nehmen uns Programmierern die Arbeit ab, fehleranfällige und immer wiederkehrende Arbeiten wie das Implementieren von schnellen Sortieralgorithmen auszuführen. Die Algorithmen machen starken Gebrauch von Iteratoren.

10.13.1 Typen von Collections (= verschiedene Interfaces):



Zwei generelle Typen:

- "Normale" Collections. Sammlung von Objekten.
- Map. (key,value)-Paare.

Duplizität von Elementen:

- `Set` und `Map`: Elemente (bei `Map`: "key-value-pairs") können nicht doppelt vorkommen.
- `List` und `Queue`: Elemente können doppelt vorkommen.

  (Etwas verwirrend ist `List` diejenige Collection, welche einem Array am ähnlichsten ist.)

Sortiert vs. unsortiert:

- `List` ist ohnehin in fester Reihenfolge.
- Bei `Set` und `Map` kann die Reihenfolge entweder vorhersehbar festgelegt werden, oder sie wird dem Computer überlassen. Die zweite Version ist meistens effizienter.

  (Wir nehmen dennoch inzwischen fast immer die erste ... leichter zu testen.)
- Bei `Queue` ist das Sortierkriterium für das *vorderste* Element festgelegt. Für viele Zwecke reicht das aus, und ist dann effizienter als eine vollständig sortierte Datenstruktur.

10.13.2 Implementierung

Ein gegebenes Interface kann auf viele verschiedene Arten implementiert werden. Z.B.

- Teile des Interfaces sind optional. Eine Implementierung kann z.B. sparsamer aber dafür schneller sein.
- Eine Implementierung kann schlechtere theoretische Eigenschaften haben, in der Praxis aber meistens schneller sein (Hash vs Trees).

Bereits diskutiert: `ArrayList` vs. `LinkedList` als Implementationen von `List`.

Bem: *Der Implementierungs-Typ sollte möglichst nur bei `new` sichtbar werden, ansonsten verwendet man durchgängig den Interface-Typ.*

10.13.3 Algorithmen

Java Collections enthält eine Reihe von Algorithmen. Z.B.:

- Algorithmen zum Auffinden und Ersetzen von Elementen,
- Algorithmen zum Sortieren eines Containers,
- Algorithmen zum Kopieren einer Sequenz von Elementen aus einem Container in einen anderen,
- Algorithmen, die gewisse Operationen auf allen oder einem Teil der Elemente eines Containers ausführen.

10.14 Iteratoren

Siehe die bereits behandelten Beispiele.

```
interface Iterator<E> {
   boolean hasNext() ;
   E next() ;
   void remove() ; // removes last element returned by next()
}
```

Bedeutung von `E`: wie unter “Generics” besprochen: deklarierter Typ der Objekte, welche im Container drin sind.

```
interface ListIterator<E> extends Iterator<E> {
   boolean hasPrevious() ;
   E previous() ;

   void remove() ; // removes last element returned by next() or previous()

   void set(E o) ; // replace last element returned by next() or previous()
   void add(E o) ; // insert at iterator position

   int nextIndex() ;
   int previousIndex() ;
}
```

Bem.:

- Bei den meisten Collection-Typen kann man auch rückwärts iterieren.
- Veränderungen abseits der Iterator-Position sind schwierig oder unmöglich.[3]

[3] ArrayList, LinkedList:
The iterators returned by this class's iterator and listIterator methods are fail-fast: if list is structurally modified at any time after the iterator is created, in any way except through the iterator's own remove or add methods, the iterator will throw a ConcurrentModificationException. Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future.
Note that the fail-fast behavior of an iterator cannot be guaranteed ...

10.15 Weitere `Collection` Interfaces

10.15.1 `Set` Interface

Collection, bei der keine zwei Elemente doppelt vorkommen dürfen.

Ist das teuer? Zwei Eckpunkte:

- Eine moderne Implementierung sortiert die Elemente irgendwie (z.B. Hash Table, oder binärer Baum), und kann darüber in ca. $\log n$ Schritten Einmaligkeit prüfen (z.B. Hash Table: bei guter Hash-Funktion konstante Zeit; binärer Baum: maximal $\log n$).
- Wenn man es nicht braucht, dann ist selbst dies teuer.

10.15.2 `Queue` Interface

Eine Queue ((Warte-)Schlange) ist eine Datenstruktur, deren *erstes* Element einem bestimmten Sortierkriterium genügt. (Für allgemeines "object ordering" siehe später.)

Z.B.: Das erste Element ist das "älteste" (FIFO = First In First Out Queue).

Daher hat eine Queue spezielle Operationen für dieses spezielle Element. Z.B. `peek()` zum Angucken des ersten Elementes, und `poll()` zum Herausnehmen.

Genau genommen gibt es die drei wichtigen Funktionen (hinzufügen, angucken, herausnehmen) doppelt:

- einmal werfen sie eine Exception ($\approx$ Programmabbruch, s. später), wenn es einen Fehler gibt,
- das andere Mal geben sie spezielle Werte zurück.

Letzteres ist von Vorteil, wenn dieser Fall Teil des normalen Programmablaufes ist.

(Z.B. bei der Programmierung einer Kreuzungslogik ... wo die Warteschlage in der Realität durchaus leer laufen kann ... das wäre also *kein* Fehler des Programmes.)

Tabelle dazu:

	Throws exception	Returns special value
Insert	add(e)	offer(e)
Remove	remove()	poll()
Examine	element()	peek()

Je nach Programmstruktur kann man diese Befehle m.E. auch sinnvoll durcheinander benutzen – ich kann z.B. durchaus eine mathematische Warteschlange simulieren, bei der ich immer einfügen darf (also `add(e)`: wenn das Einfügen nicht geht, dann ist es ein Fehler, und es gibt eine Exception), die aber leerlaufen darf (also `peek()`).

Vollständiges Interface:

```
public interface Queue<E> extends Collection<E> {
    E element();
    boolean offer(E e);
    E peek();
    E poll();
    E remove();
}
```

Typische Implementationen des `Queue` Interfaces sind:

- `LinkedList` (kennen wir schon; sortiert FIFO)
- `PriorityQueue` (sortiert das erste Element entweder entsprechend "natural ordering" oder entsprechend einem speziell angegebenen "Comparator". Siehe später.)

10.16 Vorteile von Collections

- **It reduces programming effort:** By providing useful data structures and algorithms, a collections framework frees you to concentrate on the important parts of your program, rather than the low-level plumbing required to make it work. By facilitating interoperability among unrelated APIs (as described below), the collections framework frees you from writing oodles of adapter objects or conversion code to connect APIs.

- **It increases program speed and quality:** The collections framework does this primarily by providing high-performance, high-quality implementations of useful data structures and algorithms. Also, because the various implementations of each interface are interchangeable, programs can be easily tuned by switching collection implementations. Finally, because you're freed from the drudgery of writing your own data structures, you'll have more time to devote to improving the quality and performance of the rest of the program.

- **It allows interoperability among unrelated APIs:** The collections interfaces will become the "lingua franca" by which APIs pass collections back and forth. If my network administration API furnishes a Collection of node names, and your GUI toolkit expects a Collection of column headings, our APIs will interoperate seamlessly even though they were written independently.

- **It reduces the effort to learn and use new APIs:** Many APIs naturally take collections on input and output. In the past, each such API had a little "sub-API" devoted to

manipulating its collections. There was little consistency among these ad-hoc collections sub-APIs, so you had to learn each one from scratch and it was easy to make mistakes when using them. With the advent of standard collections interfaces, the problem goes away.

- **It reduces effort to design new APIs:** Designers and implementers don't have to reinvent the wheel each time they create an API that relies on collections. They just use the standard collections interfaces.
- **It fosters software reuse:** New data structures that conform to the standard collection interfaces are by nature reusable. The same goes for new algorithms that operate on objects that implement these interfaces.

10.17 StringBuilder

(gehört nicht unbedingt hierhier, ist aber an dieser Stelle gut zu erklären)

```
class Test {
    public static void main ( String[] args )
    {
        final int len = 10000 ;

        String str = "" ;
        for ( int ii=1 ; ii<len; ii++ ) {
            str += "ab" ;
        }
        System.out.println ( " str: " + str ) ;

        StringBuilder strb = new StringBuilder() ;
        for ( int ii=1 ; ii<len ; ii++ ) {
            strb.append ( "ab" ) ;
        }
        System.out.println ( " strb: " + strb ) ;
    }
}
```

Wenn man len auf 20000 oder 30000 setzt, dann wird die erste Option sehr deutlich langsamer.

Das Zusammenfügen von `Strings` ist *extrem* langsam.

Grund ist, dass `Strings` selber nicht veränderbar sind. Man kann also auch nichts anhängen, sondern muss bei *jeder* Veränderung einen komplett neuen String anfangen. – Vorteil ist, dass `Strings` "immutable" sind ... den Vorteil sieht man aber nicht sofort.

`StringBuilder` hingegen funktioniert in etwa wie `ArrayList`.

10.18 Performance

Generell: Benutze `ArrayList`, außer, wenn

- dies garantiert zu langsam.

  Langsam ist: Einfügen/löschen an beliebiger Stelle $\rightarrow$ `LinkedList`, siehe oben.

- dies nicht "passt" (`Set`, `Queue`, `Map`, ...).

Performance, genauer

Given an input size of N, it can be described as follows:

Name	Speed	Description
exponential time	slow	takes an amount of time proportional to a constant raised to the Nth power: K^N
polynomial time	rather slow	takes an amount of time proportional to N raised to some constant power: N^K
linear time	depends	takes an amount of time directly proportional to N: $K \cdot N$
logarithmic time	fast	takes an amount of time proportional to the logarithm of N: $\log(N)$
constant time	fastest	takes a fixed amount of time, no matter how large the input is: K

Amortized time: Gemittelt über viele (rnd) Zugriffe.