# Java Collections Selbststudium

## 1. Implementation einer eigenen Collection

In der Gruppenarbeit haben sie eine Collection-Variante (*Set* oder *Bag*) implementiert, welche die Daten entweder sortiert oder unsortiert ablegt. Implementieren Sie nun noch die diagonal gegenüberliegende Version gemäss der Tabelle mit den verschiedenen Varianten:

Die Lösungen finden Sie auf den folgenden Seiten.

| | | **Sortierung** | |
|---|---|---|---|
| | | **Sortiert** | **Nicht sortiert** |
| **Semantik** | **Set-Semantik** | *SortedSet* | *UnsortedSet* |
| | **Bag-Semantik** | *SortedBag* | *UnsortedBag* |

## 2. Abwägungen der verschiedenen Datenstrukturen

Gegeben sind mehrere Anwendungsfälle. Entscheiden Sie sich für die passendste der vier Datenstrukturen und begründen Sie Ihren Entscheid.

Teilweise sind mehrere Antworten richtig. Wichtig ist, dass Sie für Ihre gewählte Datenstruktur eine passende Begründung haben.

a) Eine Werbeagentur sammelt E-Mail Adressen, an welche Sie dann Spammails versendet. Welche Datenstruktur empfiehlt sich für die E-Mail Sammlung?

Da jede E-Mail Adresse nur 1x vorkommen soll wird eine Set-Struktur benötigt. Da jede Einfüge-Operation eine Suchoperation benötigt, was bei sortierten Daten schneller geht, empfiehlt sich die **SortedSet** Variante.

Eine andere Möglichkeit wäre das **UnsortedSet**, da Einfüge-Operationen sowieso in O(n) liegen und hier hauptsächlich eingefügt wird sowie am Schluss über die komplette Liste iteriert wird.

Eine dritte Möglichkeit wäre die Verwendung eines **UnsortedBags** mit der Begründung, dass schnell eingefügt werden soll und es egal ist, ob ein Empfänger die E-Mails mehrmals empfängt.

b) Sie entwerfen Software für eine Bibliothek, wo alle Bücher abgespeichert werden. Wenn ein Buch in mehreren Exemplaren vorhanden ist, wird es auch mehrmals abgelegt (weil die Exemplare einzeln ausgeliehen werden können). Die Kunden der Bibliothek nutzen das System für die Suche (nach Titel).

Ein Buch kann auch mehrfach vorkommen: Bag-Semantik: Die Kunden führen viele Suchanfragen durch: Sortierte Datenablage => **SortedBag**

c) Die Kartbahn Wohlen möchte eine Statistik über die Rundenzeiten der Fahrer machen. Sie möchte am Ende des Tages die schnellste und langsamste Rundenzeit auslesen.

Da die genau gleiche Rundenzeit öfters vorkommen kann muss eine Bag-Semantik Collection verwendet werden. Da die Rundenzeiten nur am Ende des Tages ausgewertet werden und während des ganzen Tages Zeiten eingefügt werden empfiehlt sich eine Datenstruktur, die schnell Einfügeoperationen durchführen kann. Die Reihenfolge ist weniger wichtig. Am Ende des Tages braucht die Suche für die schnellste und langsamste Rundenzeit noch eine Iteration mit O(n). Deshalb empfiehlt sich hier **UnsortedBag**.

d) Sie sollen einem Telefonanbieter eine Datenstruktur für eine Anzeige der letzten 10 gewählten Nummern vorschlagen. Der Speicherplatz auf dem Telefon ist beschränkt. Es können maximal 10 Plätze belegt werden. Die Telefonnummern werden in der Reihenfolge abgelegt, wie diese gewählt wurden, jedoch wird jede Nummer nur einmal abgelegt.

Da jede Telefonnummer nur einmal vorkommen darf muss eine Set-Semantik gewählt werden. Da hier zusätzlich die Anforderung besteht, dass die Reihenfolge bekannt sein muss, muss hier in **UnsortedSet** verwendet werden.

© Michael Henninger, Prof. Dr. Wolfgang Weck

```java
public class UnsortedBag<E> extends AbstractArrayCollection<E> {
   public static final int DEFAULT_CAPACITY = 100;
   private int size = 0;
   private E[] data;

   public UnsortedBag() { this(DEFAULT_CAPACITY); }

   @SuppressWarnings("unchecked")
   public UnsortedBag(int capacity) { data = (E[])new Object[capacity]; }

   @Override
   public boolean add(E e) {
      if (e == null) throw new NullPointerException("null is not allowed");
      if (size == data.length) throw new IllegalStateException("Collection is full");
      data[size] = e;
      size++;
      return true;
   }

   @Override
   public boolean remove(Object o) {
      int i = indexOf(o);
      if (i >= 0) {
         data[i] = data[size - 1];
         data[size - 1] = null;
         size--;
         return true;
      } else return false;
   }

   @Override
   public boolean contains(Object o) { return indexOf(o) >= 0; }

   @Override
   public Object[] toArray() { return Arrays.copyOf(data, size()); }

   @Override
   public int size() { return size; }

   private int indexOf(Object o) {
      if (o == null) throw new NullPointerException();
      int i = size - 1;
      while (i >= 0 && !data[i].equals(o)) { i--; }
      return i;
   }
}
```

```
public class SortedBag<E extends Comparable<? super E>>
extends AbstractArrayCollection<E> {
   public static final int DEFAULT_CAPACITY = 100;
   private int size = 0;
   private E[] data;

   public SortedBag() { this(DEFAULT_CAPACITY); }

   @SuppressWarnings("unchecked")
   public SortedBag(int capacity) { data = (E[])new Comparable[capacity]; }

   @Override
   public boolean add(E e) {
      if (e == null) throw new NullPointerException();
      if (size == data.length) throw new IllegalStateException("Collection is full");
      int i = size;
      while (i > 0 && data[i - 1].compareTo(e) > 0) {
         data[i] = data[i - 1];
         i--;
      }
      data[i] = e;
      size++;
      return true;
   }

   @Override
   public boolean remove(Object o) {
      int i = Arrays.binarySearch(data, 0, size, o);
      if (i >= 0) {
         while (i != size - 1) {
            data[i] = data[i + 1];
            i++;
         }
         data[i] = null;
         size--;
         return true;
      } else return false;
   }

   @Override
   public boolean contains(Object o) {
      return Arrays.binarySearch(data, 0, size, o) >= 0;
   }

   @Override
   public Object[] toArray() { return Arrays.copyOf(data, size()); }

   @Override
   public int size() { return size; }
}
```

```java
public class UnsortedSet<E> extends AbstractArrayCollection<E> implements Set<E> {
   public static final int DEFAULT_CAPACITY = 100;
   private int size;
   private E[] data;

   public UnsortedSet() { this(DEFAULT_CAPACITY); }

   @SuppressWarnings("unchecked")
   public UnsortedSet(int capacity) { data = (E[])new Object[capacity]; }

   @Override
   public boolean add(E e) {
      if (indexOf(e) < 0) {
         if (size == data.length)
            throw new IllegalStateException("Collection is full");
         data[size] = e;
         size++;
         return true;
      } else return false;
   }

   @Override
   public boolean remove(Object o) {
      int i = indexOf(o);
      if (i >= 0) {
         data[i] = data[size - 1];
         data[size - 1] = null;
         size--;
         return true;
      } else return false;
   }

   @Override
   public boolean contains(Object o) { return indexOf(o) >= 0; }

   @Override
   public Object[] toArray() { return Arrays.copyOf(data, size()); }

   @Override
   public int size() { return size; }

   private int indexOf(Object o) {
      if (o == null) throw new NullPointerException();
      int i = size - 1;
      while (i >= 0 && !data[i].equals(o)) { i--; }
      return i;
   }
}
```

```
public class SortedSet<E extends Comparable<? super E>>
extends AbstractArrayCollection<E> implements Set<E> {
   public static final int DEFAULT_CAPACITY = 100;
   private int size = 0;
   private E[] data;

   public SortedSet() { this(DEFAULT_CAPACITY); }

   @SuppressWarnings("unchecked")
   public SortedSet(int capacity) { data = (E[])new Comparable[capacity]; }

   @Override
   public boolean add(E e) {
      if (e == null) throw new NullPointerException();
      int i = indexOf(e);
      if (i < 0) {
         if (size == data.length)
            throw new IllegalStateException("Collection is full");
         i = -i - 1;
         for (int j = size; j != i; j--) { data[j] = data[j - 1]; }
         data[i] = e;
         size++;
         return true;
      } else return false;
   }

   @Override
   public boolean remove(Object o) {
      int i = indexOf(o);
      if (i >= 0) {
         while (i != size - 1) {
            data[i] = data[i + 1];
            i++;
         }
         data[i] = null;
         size--;
         return true;
      } else return false;
   }

   @Override
   public boolean contains(Object o) { return indexOf(o) >= 0; }

   @Override
   public Object[] toArray() { return Arrays.copyOf(data, size()); }

   @Override
   public int size() { return size; }

   private int indexOf(Object o) {
      return Arrays.binarySearch(data, 0, size, o);
   }
}
```