Exceptions I

Zentrale Konzepte:

- ⊙ Geordnete Sammlungen: HashTree, TreeSet, Comparable
- ⊙ Defensive Programmierung
- ⊙ Ausnahmebehandlung: Exceptions
- ⊙ Auslösen einer Exception: Throw-Anweisung
- ⊙ Auffangen einer Exception: Try/Catch-Block
- ⊙ Wiederaufsetzen des Programms

Projekt *Adressbuch* (einmal etwas anders)

- ⊙ Speichert Kontakte sowohl unter dem Namen der Person als auch unter der Telefonnummer
- ⊙ Kontakte können verwaltet werden:
  - ⊙ Kontakte eintragen
  - ⊙ Kontakte suchen
  - ⊙ Kontakte löschen
  - ⊙ Alle Kontakte auflisten (geordnet)
- ⊙ Grafische Benutzerschnittstelle

TreeMap

TreeMap enthält seine Schlüssel-Wert Paare aufsteigend geordnet nach Schlüssel.

Bsp: `TreeMap<String, Kontakt> buch = new TreeMap<String, Kontakt>();`

enthält Paare (Name, Kontakt), die alphabetisch nach dem Namen einer Person sortiert sind.

Voraussetzung: Die Klasse, nach der sortiert werden soll, muss das Interface *Comparable* implementieren.

Methoden (Auswahl):

- `SortedMap<K, V> headMap(K key)` : liefert eine geordnete Map zurück, die alle Schlüssel-Wert Paare enthält, deren Schlüssel kleiner *key* sind.
- `SortedMap<K, V> tailMap(K key)`: liefert eine geordnete Map zurück, die alle Schlüssel-Wert Paare enthält, deren Schlüssel größer oder gleich *key* sind.

TreeSet

TreeSet enthält eine Menge von Elementen, wobei jedes Element nur genau einmal in der Menge vorkommen darf. Die Elemente sind aufsteigend geordnet.

Bsp:
```
TreeSet<Kontakt> kontakte =
            new TreeSet<Kontakt>(buch.values());
```

enthält alle Werte-Elemente aus der *TreeMap buch, aufsteigend geordnet*.

Voraussetzung: Die Klasse, nach der sortiert werden soll, muss das Interface *Comparable*<*T*> implementieren.

Methoden (Auswahl):

- `SortedSet<E> subSet(E fromElement, E toElement)` : liefert eine geordnete Teilmenge zurück, die alle Elemente von *fromElement* (inklusiv) bis *toElement* (exklusiv) enthält.
- `SortedSet<E> tailSet(E fromElement)`: liefert eine geordnete Teilmenge zurück, die alle Elemente enthält, die größer oder gleich *fromElement* sind.

Interface *Comparable*

Comparable<*T*> ist ein generisches Interface und legt fest, dass alle Klassen *T*, die *Comparable*<*T*> implementieren eine Methode anbieten, mit der die Instanzen der Klasse sortiert werden können.

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

Wobei der Aufruf `x.compareTo(y);` folgende Werte liefert:

Negative Zahl: wenn x < y

Positive Zahl: wenn x > y

0: wenn x = y

Forderung: Die Implementierung von *compareTo(..)* muss konsistent mit *equals(..)* sein, d.h. wenn `x.compareTo(y) == 0` dann muss `x.equals(y) == true` sein.

Bsp: Implemetieren des Interface *Comparable*

```
public class Kontakt implements Comparable<Kontakt>
{
  private String name;
  private String telefon;
  private String email;

...

  public int compareTo(Kontakt jenerKontakt)
  {
    int vergleich = name.compareTo(jenerKontakt.getName());

    if(vergleich == 0)
      vergleich = telefon.compareTo(jenerKontakt.getTelefon());

    if(vergleich == 0)
      vergleich = email.compareTo(jenerKontakt.getEmail());

    return vergleich;
  }
```

String implementiert das Interface Comparable<String> und besitzt daher die compareTo(..) Methode.

Bsp: Implemetieren der Methode *equals(..)*

Da die Implementierung von *compareTo(..)* konsistent mit *equals(..)* sein muss, bietet sich an, die equals-Methode auf der Basis der compareTo-Methode zu realisieren.

```
public boolean equals(Object jenes)
{
   if(this == jenes)
      return true;

   if(jenes == null)
      return false;

   if(!(jenes instanceof Kontakt))
      return false;

   Kontakt jenerKontakt = (Kontakt) jenes;

   return compareTo(jenerKontakt) == 0;
}
```

Methodenparameter muss vom Typ Object sein, da sonst die Methode nicht die von Object geerbte Methode überschreibt.

Objektreferenzen identisch?

Argument gleich null?

Argument anderer Typ?

Object casten in Kontakt

Liefert der Vergleich den Wert 0, so sind die Objekte gleich.

Gründe für das Auftreten von Fehlern

- Logische Fehler: Programm verhält sich nicht so wie geplant (Fehler des Programmierers)
  - Z.B. Berechnung des Durchschnittswertes statt des Medians (Statistik)
- Methodenaufruf mit ungültigen Werten (Fehler von wem?)
  - Z.B. get-Methoden von Sammlungsobjekt mit ungültigem Index
- Objekt wird in inkonsistenten oder unangemessenen Zustand versetzt (Fehler des Klassennutzers)
  - Z.B. durch Ableiten einer Klasse und Setzen ungültiger Werte.

Fehler liegen nicht immer im Einflussbereich des Programmierers

- Fehler sind oft umgebungsbedingt
  - Fehlerhafte URL eingegeben
  - Netzwerkunterbrechung
- Dateiverarbeitung ist besonders fehleranfällig
  - Fehlende Dateien
  - Falsche Zugriffsrechte

⇒ Fehler sollten vorausgesehen und nach Möglichkeit im laufenden Programm behandelt werden, damit es nicht zu einem Absturz des Programms kommt.

Defensive Programmierung

- Klasse *Adressbuch* ist eine typische Dienstleistungsklasse, ihre Methoden werden von außen angestoßen
- Entwickler einer Dienstleistungsklasse kann zwei Perspektiven einnehmen:
  - Alle Klienten verhalten sich korrekt und stellen nur korrekte Anfragen
  - Klienten können jederzeit inkorrekte Anfragen stellen, daher muss alles unternommen werden, diese abzuwehren.
- Defensive Programmierung: Der Dienstanbieter sorgt dafür, dass fehlerhafte Anfragen kein Fehlverhalten des Dienstleisters hervorrufen können.

Methodenparameter

- Methodenparameter machen eine Klasse extrem verwundbar:
  - Parameterwerte des Konstruktors initialisieren das Objekt.
  - Methodenparameter beeinflussen das Verhalten eines Objekts.
- Das Prüfen der Methodenparameter ist eine defensive Maßnahme.

Bsp: Prüfen des Schlüssels

```
public void deleteKontakt(String schluessel) {
    if(schluesselBekannt(schluessel)) {
        Kontakt kontakt = buch.get(schluessel);
        buch.remove(kontakt.getName());
        buch.remove(kontakt.getTelefon());
        anzahlEintraege--;
    }
}
```


Fehlermeldung an den Benutzer

- Benutzer durch Konsolenausgabe oder Nachrichtenfenster informieren

```
public void deleteKontakt(String schluessel) {
    if(schluesselBekannt(schluessel)) {
        Kontakt kontakt = buch.get(schluessel);
        buch.remove(kontakt.gibName());
        buch.remove(kontakt.gibTelefon());
        anzahlEintraege--;
    }
    else System.out.println("Schlüssel ist falsch.");
}
```

- Gibt es immer einen Benutzer?
- Kann der Benutzer das Problem lösen?

Fehlermeldung an den Klienten (aufrufende Methode)

- Klienten informieren indem man einen boolschen Wert zurück liefert.

  Bsp: `public boolean deleteKontakt(String schluessel)`

- Was ist, wenn die Methode einen regulären Rückgabetyp benötigt?

  - Primitiver Typ: bestimmten Wertebereich des Typs für Fehlermeldung nutzen

    Bsp: `public int indexOf(Object o)` (liefert -1 falls *o* nicht gefunden wird)

  - Referenztyp: Rückgabewert *null* kann Fehler signalisieren.

    Bsp: `public Kontakt getKontakt(String schluessel)`

- Was ist, wenn mehrere Fehlertypen unterschieden werden sollen?
  (Objekt nicht gefunden/ungültiger Parameterwert)?

Reaktion des Klienten

- Rückgabewert prüfen
  - Versuchen den Fehler zu beheben (logischer Fehler).
  - Versuchen Programmversagen zu vermeiden (Fehler durch Nutzereingabe).

- Rückgabewert ignorieren
  - Kann nicht verhindert werden!
  - Führt vermutlich zu Programmabsturz.

=> Meldung von schwerwiegenden Fehlern per Rückgabewert birgt viele Nachteile

Exception Handling in Java

- Besonderes Sprachfeature in Java.
- Exceptions sind normale Java-Klassen die Ausnahmen beschreiben. Sie können aus der Java Bibliothek kommen oder selbst geschrieben sein.
- Exceptions können von Methoden unabhängig von Rückgabewerten *geworfen* werden.
- Der Klient kann auf eine Exception explizit reagieren, er muss sie *fangen*, ansonsten führen diese unmittelbar zum Programmabsturz.
- Die Behandlung von sog. geprüften Exceptions wird vom Compiler überwacht, d.h. der Klient wird gezwungen diese zu fangen.
- Spezielle Aktionen zur Fehlerbehebung werden unterstützt.

Auslösen einer Exception: Throw-Anweisung

- Ein Exception-Objekt wird konstruiert: `new ExceptionTyp("...");`
- Das Exception-Objekt wird geworfen: `throw ...`

Bsp: Werfen einer Exception

```
public Kontakt getKontakt(String schluessel)
{
    if(schluessel == null || schluessel.trim().length() == 0) {
        throw new IllegalArgumentException(
                    "Ungültiger schluessel in gibKontakt");
    }
    return buch.get(schluessel);
}
```

Die Auswirkungen einer Exception

- Die auslösende Methode wird sofort beendet.
- Es wird kein Rückgabewert zurück geliefert.
- Die Ablaufkontrolle kehrt nicht zum Aufrufpunkt des Klienten zurück. D.h. der Klient kann nicht einfach fortfahren.
- Der Klient (aufrufende Methode) kann eine Exception auffangen und diese behandeln.
- Der Klient kann die Exception ignorieren. Dann bricht das Programm ab, wenn es keine andere Methode gibt, die die Exception auffängt.

Das Auffangen einer *Exception*: Try-Catch-Block

- Der Klient (aufrufende Methode) kann einen problematischen Aufruf in einem Try-Block ausführen und die *Exception* in einem Catch-Block auffangen und behandeln.
- Der Catch-Block bekommt die ausgelöste Exception-Instanz als Argument übergeben.
- Das *Exception*-Objekt kann weitere Informationen enthalten, die für die Behandlung der Ausnahme benötigt wird.

```
try {
        // eine oder mehrere geschützte Anweisungen
}
catch(ExceptionTyp e) {
        // die Exception melden und eventuell wieder aufsetzen
}
```

Bsp: Auffangen einer *Exception*

```
try {
    Kontakt kontakt = ab.getKontakt(null);
    System.out.println("Hier kommt man nur hin, wenn kein
                        Fehler auftritt.");
    System.out.println(kontakt);
}
catch(IllegalArgumentException e) {
    System.out.println(e);
    //Hier kann der Klient weitere Anweisungen ausführen.
}
System.out.println("Hier kommt man nur hin, wenn der Fehler
                    im catch-Block gefangen wurde.");
```

Exception wird hier ausgelöst, Try-Block wird verlassen.

Programmausführung wird hier fortgesetzt, wenn ein Fehler aufgetreten ist und der Fehlertyp passt.

Bsp: Wiederaufsetzen nach einer *Exception*

Werden die Eingabewerte über eine Benutzerschnittstelle eingegeben, so kann der Klient im Falle eines Fehlers:

- Dem Nutzer den Fehler melden und
- den Nutzer zur erneuten Eingabe des Wertes auffordern.

Bsp: einfache Methode zum Einlesen des Schlüssels von der Konsole

```
public static String schluesselEinlesen() {
    Scanner scanner= new Scanner(System.in);
    String schluessel = "";

    System.out.println("Schlüssel des Kontakts:");
    schluessel = scanner.nextLine();
    return schluessel;
}
```

Eingabestrom: Konsole

Einlesen der aktuellen Zeile

Bsp: Wiederaufsetzen nach einer *Exception*

```
Adressbuch ab = new Adressbuch();
boolean ungueltig = true;
...
while(ungueltig) {
   try {
      Kontakt kontakt = ab.getKontakt(schluesselEinlesen());
      System.out.println(kontakt);
      ungueltig = false;
   }
   catch(IllegalArgumentException e) {
        System.out.println(e.getMessage());
   }
}
```