Exceptions I

Zentrale Konzepte:

- ⊙ HashTree, TreeSet, Comparable
- ⊙ Defensive Programmierung
- ⊙ Exceptions
- ⊙ Throw-Anweisung
- ⊙ Try/Catch-Block
- ⊙ Wiederaufsetzen des Programms

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

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


Exceptions II

Zentrale Konzepte:

- ⊙ Hierarchie der Exception Klassen
- ⊙ Geprüfte/Ungeprüfte Exceptions
- ⊙ Propagieren von Exceptions
- ⊙ Umgang mit mehreren Exceptions / Polymorphie
- ⊙ Finally-Block

Bsp: Werfen einer Exception

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

Problem

- Ein Schlüssel mit einem leeren String ist ein Eingabefehler des Benutzers (Wiederaufsetzen möglich)
- Ein Schlüssel mit Wert null ist ein logischer Fehler des Programmierers (kein Wiederaufsetzen möglich)

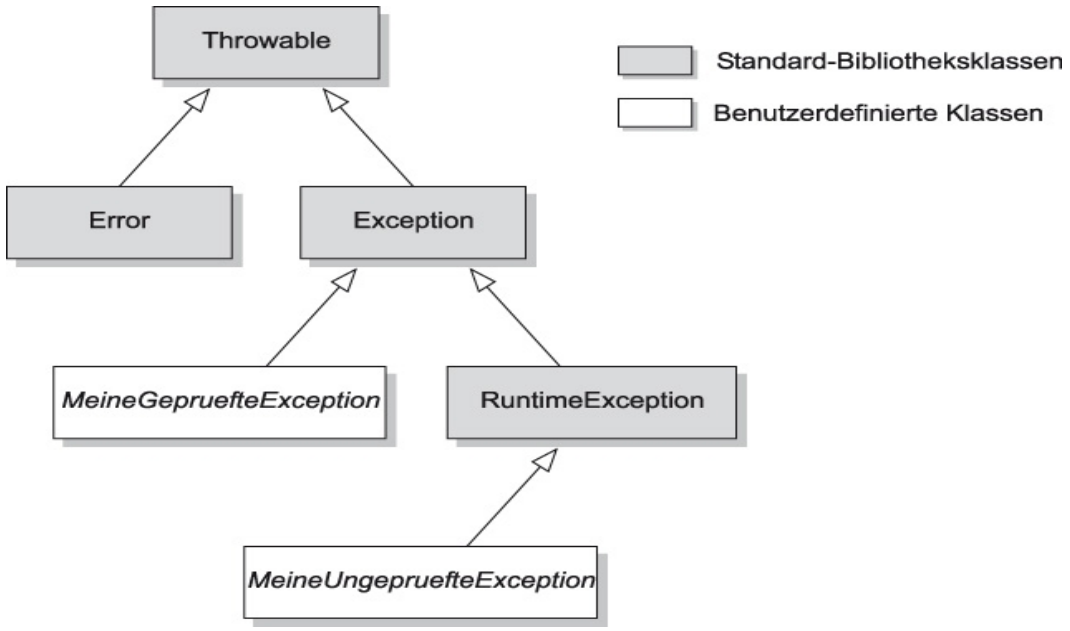
Fehlerarten

- Fehler, bei denen ein Wiederaufsetzen des Programms möglich ist:
  - z.B. durch falsche Nutzereingaben hervorgerufen
  - z.B. durch die Laufzeitumgebung hervorgerufen (z.B. Netzwerk steht nicht zur Verfügung)
- Fehler, bei denen ein Wiederaufsetzen nicht möglich ist:
  - z.B. logische Fehler im Programm

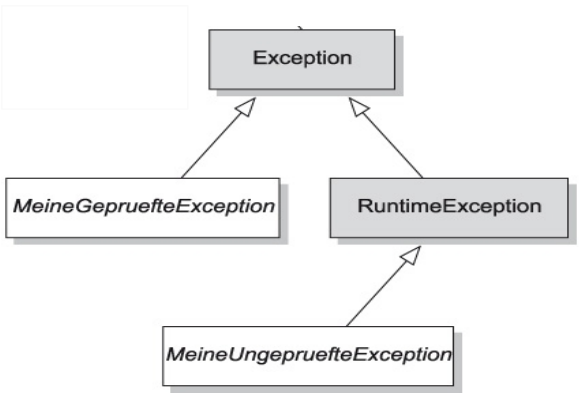
=> Besser wären zwei unterschiedliche Arten von Exceptions:

- Exceptions die der Programmierer fangen muss (Wiederaufsetzen)
- Exceptions die einen Programmabbruch zur Folge haben (Korrektur durch Programmierer).

Hierarchie der Exception Klassen



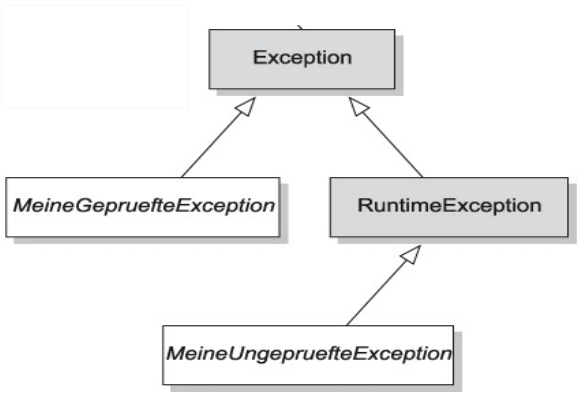
Kategorien von Exceptions



Ungeprüfte Exceptions

- Subklasse von *RuntimeException*
- Compiler prüft die Behandlung der Exception im Klienten nicht, d.h. sie ist optional.
- Werden verwendet für unvorhergesehene Fehler (logische Fehler im Programm).
- Die Fehlerbehandlung im Klienten ist schwierig.

Kategorien von Exceptions



Geprüfte Exceptions

- Subklasse von *Exception*
- Wird verwendet für vorhersehbare Fehler.
- Die Fehlerbehandlung im Klienten ist möglich, Programm muss nicht abstürzen.
- Die Behandlung wird durch den Compiler erzwungen (Try-Catch-Block)
- Die auslösende Methode muss die Exceptions deklarieren (throws-Klausel)
- Die Exceptions erscheinen in der Java-Dokumentation der Methode

Umgang mit geprüften Exceptions (Dienstleister)

- Der Programmierer schreibt eine Exception-Klasse, die den Fehler beschreibt.

```
public class UngueltigerSchluesselException extends Exception
{
   private String schluessel;

   public UngueltigerSchluesselException(String schluessel){
        this.schluessel = schluessel;
    }

   public String getSchluessel(){
        return schluessel;
    }

   public String toString(){
        return "Der Schlüssel '"
                + schluessel + "' ist ungültig.";
    }
}
```

Umgang mit geprüften Exceptions (Dienstleister)

- Die Methode, in der eine geprüfte Exception geworfen wird, muss dies in einer throws-Klausel deklarieren.
- Javadoc-Tag @throws erlaubt Spezifikation der Exception in der Dokumentation

```
/**
  * Schlage einen Namen nach und liefere den zugehörigen
  * Kontakt.
  * @param schluessel der Name zum Nachschlagen.
  * @return den zum Schluessel gehörenden Kontakt.
  * @throws UngueltigerSchluesselException besagt, dass der
  * Schlüssel ein leerer String war.
  */
 public Kontakt getKontakt(String schluessel)
                      throws UngueltigerSchluesselException {
   ...
}
```


Besser: zwei verschiedene Exceptions

```
/**
  * Schlage einen Namen nach und liefere den zugehörigen
  * Kontakt.
  * @param schluessel der Name zum Nachschlagen.
  * @return den zum Schluessel gehörenden Kontakt.
  * @throws UngueltigerSchluesselException wenn schluessel
  * einen leeren String enthält.
  */
 public Kontakt getKontakt(String schluessel)
                      throws UngueltigerSchluesselException {
    if(schluessel == null)
        throw new IllegalArgumentException("Null-Wert in gibKontakt.");
    if(schluessel.trim().length() == 0)
        throw new UngueltigerSchluesselException(schluessel);
    return buch.get(schluessel);
}
```

Umgang mit geprüften Exceptions (Klient)

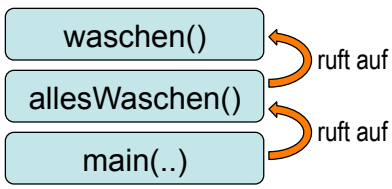
- Die Methode, die die fehlerwerfende Methode aufgerufen hat, muss die geprüfte Exception in einem Try-Catch-Block behandeln oder diese weiter propagieren.

```
public static void main(String[] arg) {
   Adressbuch ab = new Adressbuch();
   ...

   while(ungueltig) {
      try {
           schluessel = schluesselEinlesen();
           Kontakt kontakt = ab.getKontakt(schluessel);
           System.out.println("Hier kommt man nur hin, wenn keine
                                                  Exception auftritt.");
           System.out.println(kontakt);
           ungueltig = false;
      }
      catch(UngueltigerSchluesselException e) {
           System.out.println(e);
      }
   }
...}...}
```


Aufruf-Stapel der JVM

- Die Java-Virtual-Machine verwaltet aufgerufene Methoden auf einem Stapel.
- Wird in einer Methode eine andere aufgerufen, so landet diese oben auf dem Stapel.
- Ist die Methode beendet, wird sie vom Stapel entfernt. Die Abarbeitung macht in der darunter liegenden Methode bei der Zeile weiter, die nach dem Aufruf der vorhergehenden Methode kommt.




```
public class Waschmaschine {
   Waesche waesche = new Waesche();

   public void allesWaschen(){
      waesche.waschen();
   }

   public static void main(String[] args){
      WaschMaschine wm = new Waschmaschine();
      wm.allesWaschen()
   }
}
```

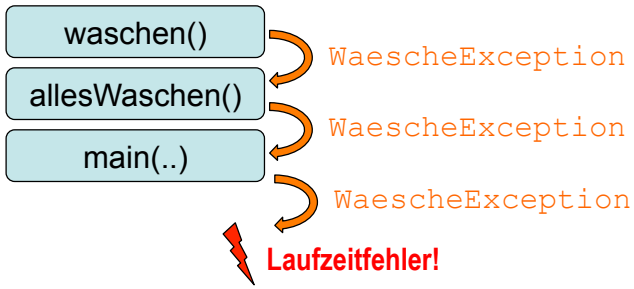
Propagieren von Exceptions

```
public class Waschmaschine {
   Waesche waesche = new Waesche();

   public void allesWaschen()throws WaescheException {
      waesche.waschen(); //wirft WaescheException
   }

   public static void main(String[] args) throws WaescheException {
      WaschMaschine wm = new Waschmaschine();
      wm.allesWaschen();
   }
}
```

Exception wird hier nicht behandelt sondern propagiert



Exceptions ausweichen (Klient)

- Die Exception muss nicht unbedingt im Klienten, behandelt werden. Dieser kann die Exception mit einer throws Klausel an seinen eigenen Klienten propagieren.

```
public class Adressbuch {
   ...
   public boolean schluesselBekannt(String schluessel)
                        throws UngueltigerSchluesselException {
      ...
   }
   public Kontakt getKontakt(String schluessel)
                        throws UngueltigerSchluesselException {
      if(schluesselBekannt(schluessel)) return buch.get(schluessel);
          else return null;
   }
}
```

Exception wird hier nicht behandelt sondern propagiert

Exception wird hier ausgelöst

Mehrere Exceptions Werfen

- Eine Methode kann durchaus mehrere geprüfte Exceptions werfen. Diese müssen alle, per Kommata getrennt, in der throws-Klausel deklariert werden.

```
public Kontakt getKontakt(String schluessel)
            throws UngueltigerSchluesselException,
                   KeinPassenderKontaktException {
  if(schluesselBekannt(schluessel))
              return buch.get(schluessel);
    else throw new KeinPassenderKontaktException(schluessel);
  }
```


Mehrere Exceptions Fangen

- Eine Methode kann mehrere Exceptions fangen.

```
public static void main(String[] arg) {
   ...
   try {
        schluessel = schluesselEinlesen();
        Kontakt kontakt = ab.getKontakt(schluessel);
        System.out.println(kontakt);
        ungueltig = false;
   }
   catch(UngueltigerSchluesselException e) {
        System.out.println(e);
   }
   catch(KeinPassenderKontaktException e) {
        System.out.println(e);
   }
   ...
```

Mehrere Exceptions Fangen: Polymorphie

- Nutzt der catch-Block einen Supertyp, so werden alle Exceptions gefangen, die Instanzen eines Subtyps sind.

```
public static void main(String[] arg) {
    ...

    try {
         schluessel = schluesselEinlesen();
         Kontakt kontakt = ab.getKontakt(schluessel);
         System.out.println(kontakt);
         ungueltig = false;
    }
    catch(Exception e) {
         System.out.println(e);
    }
    ...
```

Superklasse für alle Exceptions => Hier werden alle Exceptions abgefangen

Supertyp im catch-Block: Fallunterscheidung

- Wenn alle Exceptions in einem catch-Block gefangen werden, kann man differenziert reagieren, indem man *instanceof* nutzt.

```
public static void main(String[] arg) {
    ...

    try {
         schluessel = schluesselEinlesen();
         Kontakt kontakt = ab.getKontakt(schluessel);
         System.out.println(kontakt);
         ungueltig = false;
    }
    catch(Exception e) {
         if(e instanceof RuntimeException) throw (RuntimeException)e;
            else System.out.println(e);
    }
    ...
```

Superklasse für alle ungeprüften Exceptions.

Die Exception wird weiter geworfen.

Polymorphie im catch-Block: Anordnung der catch-Blöcke

- Reihenfolge der catch-Blöcke: erst die Subklassen, dann die Superklassen

```
public static void main(String[] arg) {
...
   try {
        Kontakt kontakt = ab.getKontakt(null);
        System.out.println(kontakt);
   }
   catch(Exception e) {
        System.out.println("Ein allgemeiner Fehler ist aufgetreten. ");
   }
   catch(KeinPassenderKontaktException e) {
        System.out.println("Ein spezieller Fehler ist aufgetreten. ");
        System.out.println(e);
   }
...
```

Superklasse für alle Exceptions => Hier werden alle Exceptions abgefangen

Diese Spezialbehandlung wird nie ausgeführt.

finally-Klausel

Die finally-Klausel erlaubt dem Programmierer, Anweisungen zu schreiben, die auf jeden Fall vor Verlassen der Methode noch ausgeführt werden, d.h. auch wenn:

- Der try-Block scheitert (geprüfte Exception)
- Der try-Block scheitert (ungeprüfte Exception) und die Exception nicht behandelt wird.
- Der try-Block erfolgreich war (keine Exception)
- Der try- oder catch-Block return-Anweisungen enthalten

```
finally {
    / / Diese Anweisungen werden auf jeden Fall ausgeführt!
}
```

Bsp: Finally-Klausel

```
public int methode(String m) {
    ...
    try {
          ...
          return 10;
    }
    catch(MyException e) {
          System.out.println(e);
          return 20;
    }
    finally {
         System.out.println("Das wird auf jeden Fall ausgeführt.");
    }
}
```

Keine Exception => vor return wird finally-Block abgearbeitet

Exception wird gefangen => vor return wird finally-Block abgearbeitet