# JAVA

## Wiederholung

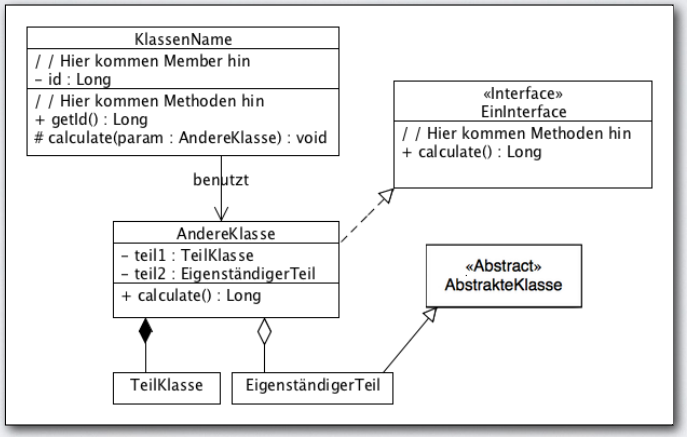
# WIEDERHOLUNG ALLGEMEIN

- public static void main(String args[]) {...}
- Namenskonventionen:
  - KlassenNamen beginnen mit Großbuchstaben
  - Variablen- und Methoden-Namen mit kleinem Buchstaben
  - Beide haben CamelCase

# WDH REFERENZEN

```java
public class Rechteck {

  private int laenge;

  private int breite;

  public Rechteck(int laenge) { this(laenge, 12); }

  public Rechteck(int laenge, int breite) {
    this.laenge = laenge;
    this.breite = breite;
  }


  private static Rechteck doSomething(Rechteck rechteck) {
    return rechteck;
  }

  public static void main(String[] args) {
    Rechteck rechteck1 = new Rechteck(1, 2);
    Rechteck rechteck2 = new Rechteck(1, 2);
    System.out.println(rechteck1 == rechteck2); // false, obwohl Inhalt gleich
    System.out.println(rechteck1 == doSomething(rechteck1)); // true
    String string1 = "rechteck";
    String string2 = "rechteck";
    String string3 = new String(string2);
    System.out.println(string1 == "rechteck"); // true, weil Strings literale sind
    System.out.println(string1 == string2); // true, gleicher Grund
    System.out.println(string1 == string3); // false
    System.out.println(compareStringsReference(string1, string2)); // true
    System.out.println(compareStringsReference(string1, string3)); // false
    System.out.println(compareStrings(string1, string2)); // true
    System.out.println(compareStrings(string1, string3)); // true
  }

  public static boolean compareStringsReference(String string1, String string2) {
    return string1 == string2;
  }

  public static boolean compareStrings(String string1, String string2) {
    if (string1 != null) {
      return string1.equals(string2);
    }
    return string1 == string2;
  }

}
```

# UML


KlassenName
/ / Hier kommen Member hin
- id : Long
/ / Hier kommen Methoden hin
+ getId() : Long
# calculate(param : AndereKlasse) : void
benutzt
«Interface»
EinInterface
/ / Hier kommen Methoden hin
+ calculate() : Long
AndereKlasse
- teil1 : TeilKlasse
- teil2 : EigenständigerTeil
+ calculate() : Long
«Abstract»
AbstrakteKlasse
TeilKlasse
EigenständigerTeil

# ÜBUNG (30 MIN)

- OO-Modellierung einer Bibliothek
  - Die Bibliothek besitzt Bücher und Zeitschriften, welche an Studenten ausgeliehen werden
  - Um die Ausleihfrist zu überprüfen wird notiert, wann etwas ausgeliehen wird
- Ziel: Klassendiagramm mit Klasse, Attributen, Methoden und Beziehungen

# JAVA

## Weiterführende Spracheigenschaften

# AGENDA

- Strings
- Exceptions
- Enums
- Generics
- Lambdas & Methods
- Bulk-Operations

# DIE KLASSE STRING

- Zeichenketten werden in Java als String repräsentiert
- Wie der Typ char sind auch Strings Unicode-fähig
- String hat viele Methoden zur Manipulation
- Keine Kenntnis über inneren Aufbau von Nöten

Vergleiche: http://commons.apache.org



1. - Ein String entspricht einer Reihung von Elementen des Typs char.
   -> Wie in anderen Sprachen auch.
   - Wiederholung: Wie groß ist ein char?
   - char Arrays sollten in Java nur noch in Ausnahmefällen verwendet werden.
3. - String liefert eine Vielzahl von Methoden zur Manipulation und Bestimmung von Eigenschaften der Zeichenkette mit.
   -> Mehr dazu gleich
   - Trotz dieser großen Vielzahl von Methoden gibt es eine Sammlung an Helfermethoden, welche einem den Umgang mit Strings noch weiter erleichtern.
   -> Diese sind aber leider kein Bestandteil dieser Vorlesung ;-). Vgl. http://commons.apache.org
4. - Anders als zum Beispiel bei C muss sich der Entwickler keine Gedanken über den inneren Aufbau von Strings Gedanken machen.
   - Keine Gedanken „verschwenden“, ob man es aktuell mit einem leeren Array von chars zu tun hat.
   - Weiterer Grund: Der innere Aufbau von Strings sind weitestgehend anonym und bedeutungslos für den Entwickler -> „Strings funktionieren einfach“.

# WICHTIGSTE METHODEN VON STRINGS

```
public StringBeispiel() {
    // Konstruktoren:

    // Ein leerer String
    String leerer = new String();
    // Duplizierung
    String duplikat = new String("Hallo");
    // Mittels char-Arrays
    char[] array = {'H', 'a', 'l', 'l', 'o' };
    String charArray = new String(array);

    // Zeichenextraktion:

    // char charAt(int index)
    // Liefert das Zeichen am angegebenen nullbasierten Index.
    char charAt = duplikat.charAt(0);
    // String substring(int begin, int end)
    // Liefert den Teilstring von inklusiv Index begin bis exklusiv Index end.
    String subString1 = duplikat.substring(1, 3);
    // Eine Variante der Methode ohne end liefert stets den Teilstring bis zum Ende.
    String subString2 = duplikat.substring(3);
    // String trim()
    // Entfernt Leerzeichen am Anfang und Ende
    String trim = "    Hallo    ".trim();

    // Übung:
    System.out.println(leerer);
    System.out.println(duplikat);
    System.out.println(charArray);
    System.out.println(charAt);
    System.out.println(subString1);
    System.out.println(subString2);
    System.out.println(trim);
}
```



Ergebnis:

”

Hallo
Hallo
H
al
lo
Hallo

”

# WICHTIGSTE METHODEN VON STRINGS

```
public StringBeispiel2() {¶
    String string1 = "Hallo!";¶
    String string2 = "Wie gehts?";¶
    AudiQFuenf q5 = new AudiQFuenf();¶

    // Länge der Zeichenkette:¶
    ¶
    // int length()¶
    // Liefert die aktuelle Länge. Der Rückgabewert 0 bedeutet Leerstring.¶
    int length = string1.length();¶

    // Vergleich von Zeichenketten:¶

    // boolean equals(Object obj)¶
    // Analog zur gleichnamigen Methode der Klasse Object.¶
    // Vergleicht zwei Strings auf inhaltliche Gleichheit.¶
    boolean equals1 = string1.equals(string2);¶
    boolean equals2 = string1.equals("Hallo!"); // Was sollte man hier beachten?¶

    // Vergleich mit der String-Darstellung beliebiger Objekte möglich,¶
    // indem zuvor obj.toString() gerufen wird.¶
    boolean equals3 = q5.toString().equals("Ist das ein Q5?");¶

    // boolean equalsIgnoreCase(String s)¶
    // Wie equals, ignoriert jedoch die Unterschiede in Groß-/Kleinschreibung.¶
    boolean equals4 = "hAllo!".equalsIgnoreCase(string1);¶
    ¶
    // Übung:¶
    System.out.println(length);¶
    System.out.println(equals1);¶
    System.out.println(equals2);¶
    System.out.println(equals3);¶
    System.out.println(equals4);¶
}¶
```



Ergebnis

”
6
false
true
false
true
”

# WICHTIGSTE METHODEN VON STRINGS

```
public StringBeispiel3() {
    String string1 = "Hallo!";
    String stringA = "a";
    String stringB = "b";

    // Vergleich von Zeichenketten:

    // boolean startsWith(String s)
    // Testet, ob ein String mit der angegebenen Zeichenkette beginnt.
    boolean startsWith = string1.startsWith("Ha");
    // boolean endsWith(String s)
    // Testet, ob ein String mit der angegebenen Zeichenkette endet.
    boolean endsWith = string1.endsWith("o!");
    // int compareTo(String s)
    // Lexikalischer Vergleich beider Strings durch paarweisen Vergleich der einzelnen Zeichen
    // von links nach rechts. Ist der aktuelle String kleiner als s, wird ein negativer Wert
    // zurückgegeben. Ist er größer, wird ein positiver Wert zurückgegeben. Bei Gleichheit ist
    // der Rückgabewert 0. --> Wichtig für Sortierung und Collections!
    int compare1 = stringA.compareTo(stringB);
    int compare2 = stringB.compareTo(stringA);
    int compare3 = stringA.compareTo(stringA);

    // Übung:
    System.out.println(startsWith);
    System.out.println(endsWith);
    System.out.println(compare1);
    System.out.println(compare2);
    System.out.println(compare3);
}
```



Ergebnis:

„
true
true
-1
1
0
“

# WICHTIGSTE METHODEN VON STRINGS

```
public StringBeispiel4() {
    String string1 = "Hallo!";

    // Suchen in Zeichenketten:

    // int indexOf(String s)
    // Sucht das erste Vorkommen von s innerhalb der Zeichenkette. Wird s gefunden, wird der
    // Index des ersten übereinstimmenden Zeichens zurückgeliefert, ansonsten -1. Eine Variante
    // der Methode akzeptiert einen Parameter vom Typ char.
    int index1 = string1.indexOf("x");
    int index2 = string1.indexOf("l");
    // Fehlercode vs. explizite Ausnahme? Was ist euer Gefühl?

    // int indexOf(String s, int fromIndex)
    // Arbeitet wie die vorige Methode, beginnt allerdings mit der Suche erst bei fromIndex.
    // Auch hier akzeptiert eine Variante der Methode einen Parameter vom Typ char.
    int index3 = string1.indexOf("l", 3);

    // int lastIndexOf(String s)
    // Sucht nach dem letzten Vorkommen von s. Eine Variante der Methode akzeptiert einen
    // Parameter vom Typ char.
    int index4 = string1.lastIndexOf("l");

    // Übung:
    System.out.println(index1);
    System.out.println(index2);
    System.out.println(index3);
    System.out.println(index4);
}
```

105

Ergebnis:

”
-1
2
3
3
”

# WICHTIGSTE METHODEN VON STRINGS

```
public StringBeispiel5() {
    String string1 = "Hallo!";

    // Ersetzen von Zeichenketten:

    // String toLowerCase()
    // Wandelt die Zeichenkette in Kleinbuchstaben.
    String string2 = string1.toLowerCase();
    // String toUpperCase()
    // Wandelt die Zeichenkette in Großbuchstaben.
    String string3 = string1.toUpperCase();
    // String replace(char old, char new)
    // Einzelne Zeichen werden ersetzt: old durch new.
    String string4 = "Salat".replace("Salat", "Schnitzel");
    // String replaceAll(String regex, String new)
    // Ersetzt alle Teilstrings, die die Regular Experession regex trifft, durch den neuen
    // Teilstring new.
    String string5 = "Hallo123".replaceAll("(\\d)", "Zahl");
    // String replaceFirst(String regex, String new)
    // Ersetzt nur den ersten gefundenen Teilstring, den die Regular Experession regex trifft,
    // durch den neuen Teilstring new.
    String string6 = "Hallo123".replaceFirst("(\\d)", "Zahl");

    // Übung:
    System.out.println(string1);
    System.out.println(string2);
    System.out.println(string3);
    System.out.println(string4);
    System.out.println(string5);
    System.out.println(string6);
}
```

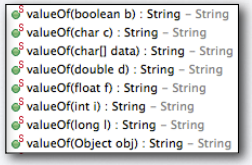
106

Ergebnis:

”
Hallo!
hallo!
HALLO!
Schnitzel
HalloZahlZahlZahl
HalloZahl23
”

# KONVERTIERUNG VON STRING

- Oftmals müssen primitive Daten in Strings gewandelt werden
  - Die Klasse String liefert dazu entsprechende statische String.valueOf(..)-Methoden:



# WEITERE EIGENSCHAFTEN VON STRINGS

- Die Klasse String ist final
- Strings sind Literale
- Verkettung durch + Operator
  - Beispiel: String string2 = "Hallo" + 123;
- Strings sind nicht dynamisch!
  - Inhalt und Länge Konstant
  - Jede Manipulation erzeugt ein neuen neuen String



1. - Eine eigene Ableitung ist nicht möglich
   - Es gibt also keine Möglichkeit, die vorhandenen Methoden auf „natürliche" Art und Weise zu ergänzen oder zu modifizieren.
   - Soll Beispielsweise die Methode „replace" etwas anderes machen, muss dies durch eine lokale Methode des Aufrufers geschehen, welche den String als Parameter bekommt.
   - Frage: Aus welchem Grund macht es Sinn, dass die Klasse String final ist?
   -> Einer der Gründe für diese Maßnahme ist die dadurch gesteigerte Effizienz beim Aufruf der Methoden von String-Objekten.
   -> Anstelle der dynamischen Methodensuche, die normalerweise erforderlich ist, kann der Compiler final-Methoden statisch kompilieren und dadurch schneller aufrufen. Daneben spielten aber auch Sicherheitsüberlegungen und Aspekte des Multithreading eine Rolle.
2. - Jedes String-Literal ist eine eigene Referenz auf ein Objekt
   - Der Compiler erzeugt für jedes Literal ein entsprechendes Objekt und verwendet es anstelle des Literals.
3. - Strings können durch den Plus-Operator verkettet werden.
   - Auch die Verkettung mit anderen Datentypen ist möglich. Es muss nur einer der Typen ein String sein.
   - Beispiel: String s = "Hallo" + 123;
4. - Strings werden wie gesagt Applikationsweit gespeichert und nur über Referenzen angesprochen.
   - D.h. bei der Initialisierung eines Strings wird der Inhalt (und somit auch die Länge) des Strings festgelegt und ist nicht mehr veränderbar.
   - Frage 1: Wie kann dann ein replace Funktionieren?
   -> Die replace Methode nimmt die Veränderung nicht auf dem Original-String vor, sondern erzeugt einen neuen String. Dieser ist mit dem gewünschten Inhalt gefüllt und gibt diese Referenz zurück.
   - Frage 2: Was passiert mit den „alten" String-Objekten?
   -> Strings sind Objekte und wenn keine Referenz mehr auf sie besteht, werden sie von der Garbage Collection weggeräumt.

# STRINGBUFFER

- Arbeitet ähnlich wie String, repräsentiert jedoch dynamische Zeichenketten
- Legt den Schwerpunkt auf Methoden zur Veränderung des Inhaltes
- Das Wandeln von StringBuffer zu String ist jederzeit möglich
- Laufzeitvorteile!

```
public StringBufferBeispiel() {¶
    String string1 = "Hallo!";¶
    StringBuffer buffer = new StringBuffer(string1);¶
    // Konkatenierung in Schleife¶
    for (int i = 0; i < 20; i++) {¶
        buffer.append(i);          // string1 += "" + i;
    }¶
    // Wandlung¶
    String bufferString = new String(buffer);¶

    // Übung:¶
    System.out.println(bufferString);¶
    System.out.println(buffer);¶
}¶
```



2. - Hinzufügen, Löschen, Ersetzen einzelner Zeichen
   - Konkatenieren von Strings
4. - Werden Strings massiv verändert oder konkateniert (z.B. in langlaufenden Schleifen) ist die Verwendung von StringBuffer zu empfehlen!
   - Der Java Compiler ersetzt (laut Spezifikation) nach Möglichkeit String-Konkatenierungen durch StringBuffer!
   -> Dies erhöht die Lesbarkeit enorm und verschlechtert nicht das Laufzeitverhalten.

# EXCEPTIONS

- Sinn von Exceptions
  - Strukturierte und separate (!) Behandlung von Fehlern, die während der Ausführung auftreten
  - Exceptions erweitern den Wertebereich einer Methode
- Beispiele: Array-Zugriff außerhalb der Grenze, Datei nicht gefunden, ...
- Begriffe: Exception, Throwing, Catching

110

2. - Ausnahmen, also Dinge, die eigentlich nicht passieren sollten, werden separat vom eigentlichen Programmcode behandelt und verarbeitet.
   - Das hat den Vorteil, dass der Quelltext wesentlich besser lesbar ist, da klar ist, was zur regulären Programmausführung und was zur Fehlerbehandlung gehört.
   -> Fehlercodes oft implizit und selbst sehr Fehleranfällig
3. - Exceptions erweitern somit den Wertebereich einer Methode.
   - Exceptions werden in die Signatur einer Methode aufgenommen. Schlüsselwort „throws XYZException“
4. - Wenn auf ein Element in einem Array zugegriffen werden möchte, dass außerhalb der Grenzen des Arrays liegt, wird das Programm nicht beendet.
   - Es fliegt eine IndexOutOfBoundsException. Diese kann separat vom regulären Programmcode behandelt werden.
5. - Exception: Die eigentliche Ausnahme, das Fehlerobjekt (!). Auch eine Exception ist ein Objekt.
   - Bsp: eine Instanz von IndexOutOfBoundsException oder FileNotFoundException.

   - Throwing: Auslösen bzw. werfen einer Exception.
   - Catching: Behandeln bzw. fangen einer zuvor geworfenen Exception.

# ABLAUF BEIM VERWENDEN VON EXCEPTIONS

1. Ein Laufzeitfehler oder eine Bedingung des Entwicklers löst eine Exception aus
2. Diese kann nun direkt behandelt und/oder weitergegeben werden
3. Wird die Exception weitergegeben, kann der Empfänger sie wiederum behandeln und/oder weitergeben
4. Wird die Exception gar nicht behandelt, führt sie zum Programmabbruch und Ausgabe einer Fehlermeldung

- Exceptions werden stets von „innen nach außen“ geworfen, d.h. eine Methode reicht die Exception nach außen an ihren Aufrufer weiter, usw.
- Sobald eine Exception auftritt wird der normale Programmablauf unterbrochen

# AUSLÖSEN UND BEHANDELN VON EXCEPTIONS

```
// Behandeln von Exceptions
public ExceptionsBeispiel() {
    // ... tue irgendwas
    try {
        tueEtwasMitEinerDatei();
    } catch (IndexOutOfBoundsException e) {
        // behandeln, weil etwas wegen einem Array kaputt ist
    } catch (FileNotFoundException e) {
        // behandeln, weil etwas wegen einer Datei kaputt ist
    } catch (Exception e) {
        // irgend etwas anderes ist kaputt gegangen.
    }
    // tue irgendwas anderes ...
}

// Auslösen von Exceptions
private void tueEtwasMitEinerDatei()
        throws IndexOutOfBoundsException,
               FileNotFoundException {
    // ...
}
```

# FEHLEROBJEKTE

- Fehlerobjekte
  - Instanzen der Klasse Throwable oder ihrer Unterklasse
  - Das Fehlerobjekt enthält Informationen über die Art des aufgetretenen Fehlers
- Wichtige Methoden von Throwable
  - String getMessage(), String toString(), void printStackTrace()

```
java.lang.RuntimeException: Hier ist was dummes passiert ..
        at ba.java.weiteres.ExceptionsBeispiel.tueEtwasMitEinerDatei(ExceptionsBeispiel.java:28)
        at ba.java.weiteres.ExceptionsBeispiel.<init>(ExceptionsBeispiel.java:11)
        at ba.java.weiteres.ExceptionsBeispiel.main(ExceptionsBeispiel.java:32)
```



- Fehlerobjekte sind in Java recht ausgiebig behandelt und die Mutterklasse aller Ausnahmen ist Throwable:
Auszug aus der JavaDoc von Throwable:

```
/**
 * The {@code Throwable} class is the superclass of all errors and
 * exceptions in the Java language. Only objects that are instances of this
 * class (or one of its subclasses) are thrown by the Java Virtual Machine or
 * can be thrown by the Java {@code throw} statement. Similarly, only
 * this class or one of its subclasses can be the argument type in a
 * {@code catch} clause. ...
 */
```

- Ein Fehlerobjekt hat viele Informationen über die Art des aufgetretenen Fehlers.

4. - Wichtige Methoden von Throwable sind:
   1. getMessage(): Liefert eine für den Menschen lesbare (!) Fehlernachricht zurück.
   2. toString(): Liefert eine String Repräsentation des Fehlers zurück: Name der Exceptionklasse und Fehlernachricht.
   3. printStackTrace()
      Schreibt den Trace des aktuellen Fehlerstacks auf den Standart Error Stream (System.err). Über den StackTrace lässt sich der Ursprung des Fehlers zurückverfolgen.

# FEHLERKLASSEN VON JAVA

- Alle Laufzeitfehler sind Unterklassen von Throwable
- Unterhalb von Throwable existieren zwei große Hierarchien
  - Error ist Superklasse aller schwerwiegender Fehler
  - Exception ist Superklasse aller abnormalen Zustände
- Es können eigene Klassen von Exception abgeleitet werden



1. - Alles was von Throwable erbt, kann in einem catch Block gefangen werden.
3. - Error sollte die Applikation nicht selber fangen, sondern vom System verarbeiten lassen.
   - Auszug aus der JavaDoc von Error.

```
/**
 * An {@code Error} is a subclass of {@code Throwable}
 * that indicates serious problems that a reasonable application
 * should not try to catch.
 */
```

4. - Eine Exception ist ein abnormaler Zustand, den die Applikation selber fangen und behandeln kann (an sinnvoller Stelle).
   - Auszug aus der JavaDoc von Exception:

```
/**
 * The class {@code Exception} and its subclasses are a form of
 * {@code Throwable} that indicates conditions that a reasonable
 * application might want to catch.
 */
```

5. - Es ist zu vermeiden, einfach eine Instanz von Exception zu schmeißen
   - Damit würden sich Fehler nicht im catch Block unterscheiden lassen und das Konzept verliert an Mächtigkeit.

# FANGEN VON FEHLERN...

- Es können einerseits durch verschiedene catch-Blöcke unterschiedliche Exception-Typen unterschieden werden
- Andererseits ist es üblich alle Fehler mittels der Oberklasse Exception gemeinsam zu behandeln, wenn eine Unterscheidung an dieser Stelle nicht nötig ist



1. - Sehr gute Unterscheidung der Fehlerfälle.
   - Manchmal nicht sinnvoll auf jede Fehlerart unterschiedlich zu reagieren.
2. - Beispiel GUI: Im Fehlerfall bei einer Aktion „Speichere Datei“ wird dem Benutzer nur angezeigt:
     - „Sorry, konnte nicht gespeichert werden. Versuch es bitte erneut.“
     - Obwohl es unterschiedliche Gründe haben kann.
   - Daher ist diese feingranulare Unterscheidung manchmal nicht sinnvoll.

# DIE FINALLY KLAUSEL

- Nach dem try-catch kann optional eine finally-Klausel folgen
- Der Code in finally wird immer ausgeführt
- Die finally-Klausel ist damit der ideale Ort für Aufräumarbeiten

```
private void tueEtwas() throws FileNotFoundException {
    File file = null;
    try {
        file = new File("irgend/ein/pfad");
        String string = leseDatei(file);
        System.out.println(string);
    } catch (FileNotFoundException e) {
        // Fehlerbehandlung
        System.err.println(e.getMessage());
        throw e;
    } finally {
        schliesseDatei(file);
    }
}

private void schliesseDatei(File file) {
    // Gebe Referenz auf Datei frei
}

private String leseDatei(File file) throws FileNotFoundException{
    // Datei lesen ein und gebe Inhalt zurück
    return null;
}
```



1. - In der finally-Klausel werden üblicherweise Aufräumarbeiten getätigt.
   - Zum Beispiel: Schließen von Dateien, Freigeben von Objekten, ...
2. - Auch wenn die eigentliche Methode durch ein re-throw oder return verlassen wurde.
3. - Daher finden dort die Aufräumarbeiten statt, die unbedingt sein müssen.
   - Vorsicht: Der Zugriff auf Variablen, die im try-Block deklariert wurden ist nicht möglich.
   - Vorsicht2: Der Zugriff auf Variablen, die im try-Block instanziiert wurden, ist mit Vorsicht zu genießen! Auf null prüfen!

# TRY WITH RESOURCES

7

```
public void tryWithResources() {
    try (InputStream fileInput =
            new FileInputStream(new File("pfad/zur/meiner/Datei"))) {
        // Hier wird die Datei eingelesen
    } catch (Exception e) {
        // Und hier der Fehler behandelt
    }
}
```



1. - Durch den try-with-resources Block werden die Ressourcen in der try(..) Klausel automatisch geschlossen.
   - Schlüsselwort ist das AutoCloseable Interface, welches implementiert werden muss.
   - In diesem Beispiel würde fileInput nach Ende des Applikationscodes (oder des Ausnahmecodes) automatisch geschlossen werden.

# CATCH-OR-THROW REGEL

- Jede Exception muss behandelt oder weitergegeben werden
  - Eine Exception abzufangen und nicht weiter zu behandeln ist im Allgemeinen nicht sinnvoll und sollte begründet werden.

# WEITERGABE VON EXCEPTIONS

- Soll eine Exception weitergegeben werden,
  - muss diese Exception durch throws angegeben sein
  - kann es entweder eine neu instanziierte Exception
  - oder eine Exception sein, die von „weiter unten kommt“, welche explizit behandelt wurde und mittels throw weitergeworfen wurde
  - oder eine Exception sein, die von „weiter unten kommt“, welche nicht von explizit im catch Block behandelt wurde



3. - Es passiert irgendwas im Programmablauf, was dazu führt, dass irgendwo im Code „throw new XYZException()“ steht.
4. - Eine Exception wird im catch Block gefangen, eine rudimentäre Behandlung durchgeführt (zum Beispiel geloggt) und dann die gleiche Exceptions mittels „throw existierendeException;“ weitergeworfen werden.
5. - Eine Exception wird nicht im catch Block gefangen. Dann wird diese automatisch weitergeworfen, ohne dass der folgende Code etwas davon mitbekommt.
   - Doch wie kann das sein? Exceptions müssen doch immer angegeben werden?
   -> Die RuntimeException macht es möglich! Siehe nächste Folie.

# RUNTIMEEXCEPTION

- Unterklasse von Exception
- Superklasse für alle Fehler, die nicht behandelt werden müssen
  - Entwickler kann entscheiden ob er diese Exception fängt
- Ausnahme von der catch-or-throw Regel
- Muss nicht in throws deklariert werden
- Eingeständnis zum Aufwand, aber gefährlich!



2. - Eine RuntimeException (und alle Klasse, die davon erben) können behandelt werden in einem catch-Block.
   - Sie müssen aber nicht behandelt werden!
3. - Einem Entwickler ist also die Freiheit gegeben, ob er eine RuntimeException fängt, oder eben nicht.
   - Es kann eine Methode, die eine RuntimeException wirft, benutzt werden, ohne dass ein catch-Block von Nöten ist.
4. - Genau aus diesem Grund ist die RuntimeException auch eine Ausnahme von der catch-or-throw Regel.
   - Wiederholung: Was besagt die catch-or-throw Regel?
   - Warum ist es also eine Ausnahme von dieser Regel?
5. - Zudem kommt der unglaublich ungünstige Umstand, dass RuntimeExceptions nicht in die Methodensignatur aufgenommen werden müssen.
   - Warum ist dies problematisch?
6. - Die RuntimeException war ein Eingeständnis an die Java Entwickler, damit der Aufwand für Fehlerbehandlung nicht zu groß wird.
   - In meinen Augen eines der gefährlichsten Features von Java. (Seit JDK1.0)
   - Die Applikation kann „auseinander brechen“, ohne dass es dem verwendeten Entwickler überhaupt klar ist was passieren kann.

   -> In meiner Erfahrung hat die RuntimeException zu sehr großen Systemabstürzen geführt!

   -> Einfach aus dem Grund der Unwissenheit!

# ÜBUNGEN

- https://github.com/unterstein/dhbw-java-lecture/tree/master/src/ba/java/uebungen
  - https://git.io/vrF8U

# ENUMERATION

- In der Praxis treten Datentypen mit kleinen und konstanten Wertemengen relativ häufig auf
- Verwendung wie andere Typen
  - z.B. Anlegen von Variablen des Typs
  - Variablen können nur vorgegebene Werte annehmen

```
// Lokale Enumeration
enum Farbe {
    ROT, GRUEN, BLAU
}

public EnumBeispiel() {
    // Deklaration
    Farbe farbe;
    // Initialisierung
    farbe = Farbe.ROT;
}
```

# ENUMS SIND KLASSEN

- Werte von Enums sind Objekte und werden auch so genutzt
- Alle Werte von enums sind singletons
- Enums selbst besitzen weitere Eigenschaften
  - values(), valueOf(String), toString(), equals(..)
  - Werte sind direkt in switch Anweisung verwendbar
- Es gibt spezielle Collections für Enums: EnumSet, EnumMap

123

1. - Vergleiche Beispiel von eben.
   - Enums werden deklariert und zugewiesen.
   - Namenskonvention: - Schreibweise von Enums an sich wie bei Klassen (Großbuchstabe + CamelCase)
     - Schreibweise von Werten in Großbuchstaben und Unterstrich
2. - Allerdings sind alle Werte von Enums singletons.
   - D.h. alle Werte existieren genau ein mal pro Applikation.
   - Wiederholung: Was hat das also zur Folge? Wie oft ist eine Farbe ROT also in einer Applikation vorhanden?
4. - values() liefert einen Array von allen Werten die für das Enum definiert sind.
   - valueOf(„ROT“) liefert den Enum Wert, welcher zur Farbe ROT gehört (Schreibweise wichtig!).
     -> Wird kein Wert zu dem angegebenen String gefunden, wird eine IllegalArgumentException geworfen (=RuntimeException).

   -> die Klasse enum implementiert damit das Interface: Compareable und Serializable.
5. - Werte sind direkt in einer switch Anweisung verwendbar.

   -> War bis Java7 ein echtes Feature, mittlerweile sind aber auch Strings in einer switch Anweisung verwendbar.
   - Von daher eher für alte Java Versionen interessant.
6. - Warum würde eine normale Liste wenig Sinn machen bei Enums (also Aufzählungen)?

   -> Sie sind singletons, eine Liste mit 100 Elementen die alle auf das eine Objekt „Rot“ zeigen macht wenig sinn..

Hinweis: Die Interfaces Compareable, Serializable und Collections werden wir später besprechen!

# ENUMS KÖNNEN ERWEITERT WERDEN

- Enums sind Klassen ..
  - .. und lassen sich auch so definieren

```
public enum Farbe {
    ROT(255, 0, 0), GRUEN(0, 255, 0), BLAU(0, 0, 255);

    private int r, g, b;

    private Farbe(int r, int g, int b) {
        this.r = r;
        this.g = g;
        this.b = b;
    }

    public String toRGB() {
        return "r: " + r + ", g: " + g + ", b: " + b;
    }
}
```



1. - Modifier des Konstruktors ist auf private limitiert.
   - Frage Warum?
   -> Wenn er public wäre, könnten sich externe Klassen ja diesem Konstruktor bedienen und sich eigene Rot Objekte machen.
   - Modifier von Werten des Enums ist implizit public.

# BEDARF NACH GENERICS

- bis Java 1.4
  - Wenn man einen Typ nicht explizit einschränken wollte, musste man auf Object arbeiten
  - Dies ist nicht typsicher -> Rückgabewert Object
  - Daher wurde sehr viel gecastet
  - Vor allem bei Collections ein prinzipielles Problem



1. - Ohne Generics konnten Klassen, die andere Typen verwenden diese aber nicht explizit einschränkten wollte, lediglich auf Object arbeiten
2. - Dies ist nicht typsicher, das heißt: Die Klasse kann in ihren Methoden zwar alle möglichen Objekte aufnehmen, bei der Rückgabe von gespeicherten Objekten kann der Typ der Objekte allerdings nicht mehr unterschieden werden.
   - Beispiel: Ich möchte eine Liste wo nur „ähnliche Objekte“ drin sind. Also zum Beispiel nur Autos oder Unterklassen.
   -> Nun ist es Schwachsinn für Jede Klasse eine eigene Listenimplementierung zu machen, genau so wie es Schwachsinn ist in diesem Beispiel Autos zusammen mit Kuchen zu speichern (was bei Object allerdings möglich wäre).
3. - Da nur auf Object gearbeitet wurde, musste viel von Object z.B. nach Auto und von Auto nach Object gecastet werden.
4. - Wie bereits in den Beispielen angesprochen ist dies vor allem bei Collections ein sehr großes Problem.
   - Eine Collection ist im Prinzip wie ein Array, nur wesentlich komfortabler.

# BEISPIEL

```java
package ba.java.weiteres.generics;

public class AlteListe {
  private Object[] data;
  private int size;

  public AlteListe(int maxSize) {
    this.data = new Object[maxSize];
    this.size = 0;
  }

  public void addElement(Object element) {
    if (size >= data.length) {
      throw new ArrayIndexOutOfBoundsException();
    }
    data[size++] = element;
  }

  public Object elementAt(int index) {
    if (size >= data.length) {
      throw new ArrayIndexOutOfBoundsException();
    }
    return data[index];
  }
}
```

```java
package ba.java.weiteres.generics;

public class AlteListeVerwendung {

  public AlteListeVerwendung() {
    // Hmm...eigentlich will ich nur Integer zulassen, geht aber nicht :-(
    AlteListe liste = new AlteListe(10);
    liste.addElement(5);
    liste.addElement(1.5); // führt zu keinem Compiler-Fehler!

    // Der Rückgabetyp ist Object
    Object objectFromListe = liste.elementAt(0);
    Integer integerFromListe = (Integer) objectFromListe;
  }
}
```



Anmerkung:
Man kann in Java keinen neuen generischen Array anlegen. Daher wird hier auf Objekt gearbeitet.

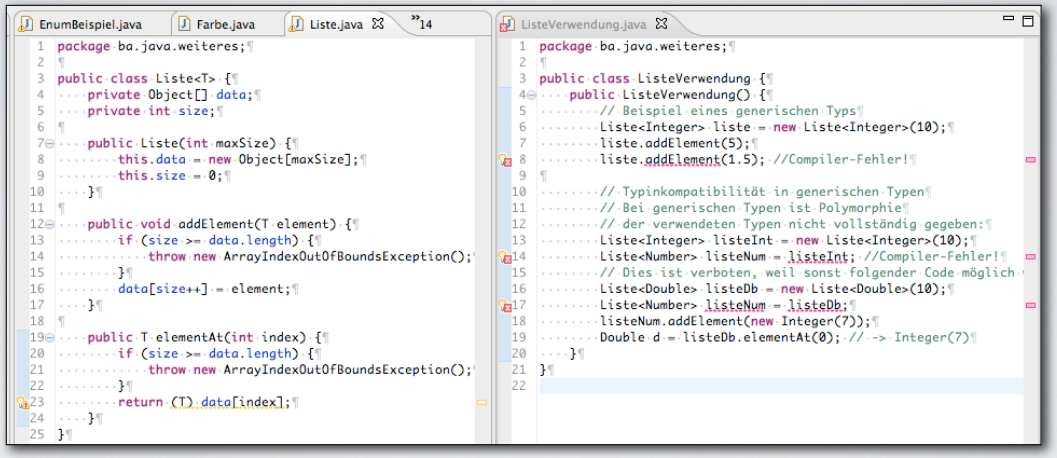
# SINN VON GENERICS

- Ab Java 5
  - Mittels Generics können typsichere Klassen unabhängig vom konkreten verwendeten Typ definiert werden
  - Die Klasse wird so implementiert, dass sie mit jedem Typ zusammenarbeiten kann (Kann eingeschränkt werden)
  - Mit welchem Typ sie zusammen arbeiten muss, wird bei Initialisierung festgelegt
- Generics ersetzen die Arbeit mit Object dennoch nicht (ganz)



3. - Mit welcher konkreten Klasse eine Klasse zusammen arbeiten muss, kann bei der Initialisierung festgelegt und eingeschränkt werden.
5. - Beim Instanziieren einer generischen Klasse muss der konkrete Typ, mit dem gearbeitet werden soll, angegeben werden.
- Danach kann diese Instanz nur mit diesem Typ arbeiten.
- Es gibt jedoch immer wieder Fälle, in denen der konkrete Typ irrelevant ist.
-> In diesen Fällen ist es nach wie vor gut, allgemein auf Object zu arbeiten.
Beispiel: List<Object> meinEigentum;

# BEISPIEL



EnumBeispiel.java | Farbe.java | Liste.java | 14

```
1  package ba.java.weiteres;
2
3  public class Liste<T> {
4      private Object[] data;
5      private int size;
6
7      public Liste(int maxSize) {
8          this.data = new Object[maxSize];
9          this.size = 0;
10     }
11
12     public void addElement(T element) {
13         if (size >= data.length) {
14             throw new ArrayIndexOutOfBoundsException();
15         }
16         data[size++] = element;
17     }
18
19     public T elementAt(int index) {
20         if (size >= data.length) {
21             throw new ArrayIndexOutOfBoundsException();
22         }
23         return (T) data[index];
24     }
25 }
```

ListeVerwendung.java

```
1  package ba.java.weiteres;
2
3  public class ListeVerwendung {
4      public ListeVerwendung() {
5          // Beispiel eines generischen Typs
6          Liste<Integer> liste = new Liste<Integer>(10);
7          liste.addElement(5);
8          liste.addElement(1.5); //Compiler-Fehler!
9
10         // Typinkompatibilität in generischen Typen
11         // Bei generischen Typen ist Polymorphie
12         // der verwendeten Typen nicht vollständig gegeben:
13         Liste<Integer> listeInt = new Liste<Integer>(10);
14         Liste<Number> listeNum = listeInt; //Compiler-Fehler!
15         // Dies ist verboten, weil sonst folgender Code möglich
16         Liste<Double> listeDb = new Liste<Double>(10);
17         Liste<Number> listeNum = listeDb;
18         listeNum.addElement(new Integer(7));
19         Double d = listeDb.elementAt(0); // -> Integer(7)
20     }
21 }
22
```



Anmerkung:
Man kann in Java keinen neuen generischen Array anlegen. Daher wird hier auf Objekt gearbeitet.

# WILDCARD ? ALS TYPPARAMETER

- „?“ Kann anstelle eines konkreten Typs angegeben werden, wenn der Typ selbst keine Rolle spielt
- Damit geht die Typsicherheit verloren, allerdings explizit
- Nur bei lesendem Zugriff erlaubt. Kein new Liste<?> möglich!

```
public void printAll(List<?> l) {
    for (Object o : l) {
        System.out.println(o);
    }
}
```

# GEBUNDENE WILDCARDS

- Abgeschwächte Form der ? Wildcard durch Einschränkung auf Subklassen einer gegebenen Superklasse mittels extends
- Es ist auch eine Einschränkung nach oben durch super möglich
- Auch die gebundene Wildcard ist nur lesend erlaubt

```
// Vorsicht: Das ist nicht unsere Klasse Liste!
public void printAllNumber(List<? extends Number> list) {
    List<? extends Number> liste2; // Auch dies ist lesend!
    for(Number numb : list) {
        System.out.println(numb.doubleValue());
    }
}
```



2. - Aber äußerst selten!

# PROJECT COIN

7

```
public void projectCoin() {
    List<Integer> integerList = new LinkedList<Integer>();
    Map<Integer, List<Integer>> integerZuIntegerListMap =
            new HashMap<Integer, List<Integer>>();
    Map<Integer, Map<Integer, List<Integer>>> blabla =
            new HashMap<Integer, Map<Integer, List<Integer>>>();
    // Das ist zu viel! Daher ProjectCoin ab Java7
    // Der Compiler weiß schon am Besten, was ich initialisieren will,
    // kümmert er sich auch um die Generic Handling
    List<Integer> integerList7 = new LinkedList<>();
    Map<Integer, List<Integer>> integerZuIntegerListMap7 = new HashMap<>();
    Map<Integer, Map<Integer, List<Integer>>> blabla7 = new HashMap<>();
}
```

# LAMBDAS

- Innere Klassen mit neuer Syntax zu verwenden
- Nur für Interfaces mit einer Methode


```java
public class Lambdas {
    public static void main(String[] args) {
        Button btn = new Button();
        // Java < 8
        btn.setOnAction(new EventHandler<ActionEvent>() {
            @Override
            public void handle(ActionEvent event) {
                System.out.println("Hello World!");
            }
        });
        // Java >= 8
        btn.setOnAction(
            event -> System.out.println("Hello World!")
        );
    }
}
```

# METHODEN REFERENZEN

- Übergabe von Methoden als Referenzen in andere Methoden

```java
public class Lambdas {
    public int compareByLength(String in, String out) {
        return in.length() - out.length();
    }

    public static void main(String[] args) {
        Lambdas myObject = new Lambdas(args);
        // Man kann in Java 8 die Referenz einer Methode übergeben
        Arrays.sort(args, myObject::compareByLength);
    }
}
```

# BULK OPERATIONS

- Bulk Operationen liefern Möglichkeit etwas auf allen Elementen einer Collection zu tun (durch Lambdas)
- z.B. filtern, manipulieren, …

```java
public class BulkOperations {

    public static void main(String[] args) {
        List<String> names = Arrays.asList("Smith", "Adams", "Crawford");
        List<Person> people = find("London");

        // Streams verwenden um zu filtern
        List<Person> anyMatch = people.stream().
            filter(p -> (names.stream().anyMatch(p.name::contains))).collect(Collectors.toList());

        // Statt anyMatch kann man auch gezielter suchen
    }

    public static List<Person> find(String location) { ... }
}
```

# ÜBUNGEN

- https://github.com/unterstein/dhbw-java-lecture/tree/master/src/ba/java/uebungen
  - https://git.io/vrF8U

# JAVA

## Java Klassenbibliotheken

# AGENDA

- Allgemeines
- Collections
- Utility-Klassen
- Dateihandling
- Reflection
- Weiterführende API

# UMFANG DER KLASSENBIBLIOTHEKEN

- Die Klassenbibliothek von Java ist groß und mächtig
- Rahmen dieser Vorstellung:
  - Was gibt es wichtiges und wo finde ich es?
- Die Klassenbibliothek ist gut in JavaDoc dokumentiert



4. - Soll ein Element der Klassenbibliothek benutzt werden, lohnt es sich in die Dokumentation zu schauen.
   - Alternativ kann auch direkt der Sourcecode von Java angeschaut werden.
   - Ich würde einfach Control+LinksKlick machen auf einer Klasse, die ich benutzen möchte und einfach in den Source Code rein schauen.

# COLLECTIONS

- Eine Collection ist eine Datenstruktur, um Mengen von Daten aufzunehmen und zu verarbeiten
  - Die Verwaltung wird gekapselt
- Ein Array ist einfachste Art der Collection
  - Collections sind aber mächtiger und einfacher zu benutzen
  - Daher werden Arrays nur relativ selten in Java benutzt



2. - Zugriff auf die Daten ist nur über definierte Methoden möglich.
   - Ermöglicht neue Sichten auf die Daten, beispielsweise Zugriff auf die Daten über Schlüssel.
5. - Dass wir in dem Beispiel das letzte mal Arrays benutzt haben war, weil die Collections schon eine reichhaltige API zur Sortierung haben und wir ja von Hand sortieren wollten.

# WICHTIGSTE COLLECTIONS

- java.util.*
- Seit Java 5 sind die Collections generisch
- Namens Konvention: <Stil><Interface>
- Wichtigste Interfaces: List, Set, Map, Queue
- Tree vs. Hash Implementierungen



3. - Beispiel: ArrayList, Array ist der Stil und List das Interface
HashMap, Hash ist der Stil und Map das Interface
4. - List = Geordnete Collections, die Duplikate erlauben
Set = Collections ohne Duplikate
Map = Collections, die eine Schlüssel/Wert-Ablage erlaubt
Queue = Collections, die sog. „Warteschlangen“ darstellen, und Objekte meist FIFO (first-in-first-out) behandeln
5. - TreeMap vs. HashMap und TreeSet vs HashSet
JavaDoc von TreeMap
```
 * <p>This implementation provides guaranteed log(n) time cost for the
 * {@code containsKey}, {@code get}, {@code put} and {@code remove}
 * operations.  Algorithms are adaptations of those in Cormen, Leiserson, and
 * Rivest's <em>Introduction to Algorithms</em>. */
```
JavaDoc von HashMap
```
 /* <p>This implementation provides constant-time performance for the basic
 * operations (<tt>get</tt> and <tt>put</tt>), assuming the hash function
 * disperses the elements properly among the buckets. */
```

Daher ist die HashMap vorzuziehen, außer man will später auch eine geordnete Ausgabe wieder bekommen.

Ihr werdet unterschiede im Laufe des Studiums noch ausreichend erklärt bekommen :-)

# ARBEITSWEISE HASH-TABELLE

```
Map<Integer, String> mitarbeiter = new HashMap<>();
mitarbeiter.put(4711, "Klaus");
mitarbeiter.put(4712, "Peter");
mitarbeiter.put(4747, "Hans");
System.out.println(mitarbeiter.get(4711));
```

- Schlüssel-Wert-Paare
- Berechnung Schlüssel durch Hash-Funktion
- Schlüssel dient als Index eines internen Arrays
- Füllgrad bzw. load factor



- Die Hash-Tabelle arbeitet mit Schlüssel-Werte-Paaren. Aus dem Schlüssel wird nach einer mathematischen Funktion – der so genannten Hash-Funktion – ein Hashcode berechnet.
- Dieser dient dann als Index für ein internes Array.
- Dieses Array hat am Anfang eine feste Größe. Wenn später eine Anfrage nach dem Schlüssel gestellt wird, muss einfach diese Berechnung erfolgen, und wir können dann an dieser Stelle nachsehen.
- Falls eine Kollision auftritt, wird ein kleines Behälterobjekt mit dem Schlüssel und Wert aufgebaut und als Element an die Liste angehängt. Eine Sortierung findet nicht statt.

4. - Maß des Füllstandes
   - Zwischen 0% und 100%.
   - Für performanten Zugriff sollte ein Füllstand von 75% nicht überschritten werden. Standard-Implementierungen beachten dies und vergrößern dynamisch die Array-Größe. Übernimmt Java allerdings für euch!

# IMPLEMENTIERUNGEN LIST

- ArrayList
- LinkedList
- Stack
- Vector



1. - ArrayList: Implementation eines „Resizable-Array“.
   Die Größe kann vorgegeben werden, wächst aber auch mit.
2. - LinkedList: Implementierung einer verlinkten Liste.
   Spezielle Methoden zum Zugriff auf das erste und letzte Element.
   Implementiert nicht nur das List Interface sondern auch das Queue Interface.
   Kann daher auch als Stack, Queue oder Double-Ended Queue (= Deque) eingesetzt werden.
3. - Stack: Klassischer LIFO (last-in-first-out) Speicher.
   Bietet die typischen push/pop Operationen.
4. - Vector: Implementation eines „Resizable-Array“
   Verhält sich wie eine ArrayList.
   Existiert historisch länger als eine ArrayList und ist im Gegensatz zu dieser synchronisiert (Multithreading).

An dieser Stelle aber die Vererbung von Stack beachten.

# IMPLEMENTIERUNGEN SET

- EnumSet
- HashSet
- TreeSet



1. - EnumSet: Spezialisiertes Set für Enums
   Alle Elemente des Sets müssen von einem gemeinsamen Enum stammen.
2. - HashSet: Set, das auf einer Hashtabelle aufbaut
   Die Reihenfolge der Elemente spielt hier keine Rolle.
   Die Performance der Operationen (add, …) ist konstant.
   Elemente müssen Hash-fähig sein (Methode hashCode()).
3. - TreeSet: Sortiertes Set
   Die Elemente werden automatisch sortiert.
   Dazu müssen die Elemente das Interface Comparable implementieren (Methode compareTo()).

# IMPLEMENTIERUNGEN MAP

- EnumMap
- HashMap
- Hashtable (auch klein table!)
- TreeMap



1. - EnumMap: Spezialisierte Map für Enums als Schlüssel.
   Alle Schlüssel müssen von einem gemeinsamen Enum stammen.
2. - HashMap: Implementierung einer Hashtabelle.
   Die Reihenfolge der Elemente spielt hier keine Rolle.
   Die Performance der Operationen (put, …) ist konstant.
   Elemente müssen Hash-fähig sein (Methode hashCode()).
3. - Hashtable: Implementierung einer Hashtabelle.
   Verhält sich wie eine HashMap.
   Existiert historisch länger als eine HashMap und ist im Gegensatz zu dieser synchronisiert (Multithreading)
4. - TreeMap: Sortierte Map.
   Die Schlüssel werden automatisch sortiert.
   Dazu müssen die Schlüssel das Interface Comparable implementieren (Methode compareTo()).

Frage: Kann man in eine Map oder ein Set „null“ adden?
- auf null kann man nicht hashCode() oder compateTo() aufrufen!

# DIE KLASSE COLLECTIONS

- Statische Methoden zur Manipulation und Verarbeitung von Collections
- Besonderes die Methoden zum Synchronisieren sind im Zusammenhang mit Multithreading wichtig

```
public CollectionsBeispiel() {
    List<Integer> meineListe = new LinkedList<>();
    meineListe.add(5);
    meineListe.add(2);
    meineListe.add(10);
    // sortiert Liste aufsteigend
    // Integer implementiert das Interface Comparable<Integer>
    Collections.sort(meineListe);
    // Ein spezieller Vergleicher,
    // der zur Sortierung heran gezogen wird
    Comparator<Integer> vergleicher = new Comparator<Integer>() {
        @Override
        public int compare(Integer int1, Integer int2) {
            return int2.compareTo(int1);
        }
    };
    // Liste speziell sortieren
    Collections.sort(meineListe, vergleicher);
    // dreht die Elemente der Liste (wieder) um
    Collections.reverse(meineListe);
    List<Integer> synchronisierteListe =
            Collections.synchronizedList(meineListe);
    List<Integer> unmodifizierbareListe =
            Collections.unmodifiableList(meineListe);
    Integer max = Collections.max(meineListe);
    // ...
}
```



1. - Durchsuchen
   - Sortieren
   - Kopieren
   - Erzeugen unveränderlicher Collections
   - Erzeugen synchronisierter Collections
2. - Nur die älteren Collections sind von Haus aus synchronisiert (Vector, Hashtable, ..).
   - Die neueren Collection-Klassen sind aus Performance-Gründen nicht thread-safe.
   - Diese können aber durch die Methoden von Collection zu solchen gemacht werden.

# DIE KLASSE ARRAYS

- Statische Methoden zur Manipulation und Verarbeitung von Arrays
- parallellSort

```
private void arraysBeispiele() {¶
    Integer[] meinArray = { 5, 2, 10, 25, 1, 20 };¶
    List<Integer> alsListe = Arrays.asList(meinArray);¶
    // sortiert Liste aufsteigend¶
    Arrays.sort(meinArray);¶
    // Vergleich von Arrays.equals geht auf den Inhalt!¶
    Integer[] zweiterArray = { 1, 2, 3 };¶
    boolean vergleich = Arrays.equals(meinArray, zweiterArray);¶
    // Es können Elemente gesucht werden¶
    int indexOfSearch = Arrays.binarySearch(meinArray, 20);¶
    System.out.println(Arrays.toString(meinArray));¶
    // Gibt "[1, 2, 5, 10, 20, 25]" aus¶
}¶
```



1.
- Durchsuchen
- Sortieren
- Vergleichen
- Vorbefüllen
- Inhalt als String ausgeben
- etc.
- Im Endeffekt das gleiche wie Collections nur mit Arrays

# WICHTIGE UTILITY KLASSEN

- Utility Klassen sind Klassen, welche sich nicht konkret einordnen lassen, die man aber immer wieder benötigt
- Wichtige Utility Klassen (1)
  - java.util.Random
  - java.security.SecureRandom
  - java.util.GregorianCalendar und java.util.Date

```
private void utilities() {
    Random random =
            new Random(System.currentTimeMillis() % 14930352);
    for(int i = 0; i<10;i++) {
        System.out.println(random.nextInt());
    }
}
```

Vergleiche: http://joda-time.sourceforge.net



3. - Random erzeugt Zufallszahlen.
4. - Calendar und Date sind zur Verarbeitung Datums- und Zeitwerten da.
   - Datumsarithmetik ist nicht einfach und spielt mindestens in der Liga von Zeichencodierungsproblemen.
   - Datumsarithmetik nicht hinreichend in der Java Standardbibliothek gelöst.
   - Date und Calendar sind stark veraltet, wird aber noch im Zusammenhang von Datenbanken gerne verwendet.
   - Tut euch selber einen gefallen und verwendet JodaTime (http://joda-time.sourceforge.net) oder die Java 8 Implementierung

# WICHTIGE UTILITY KLASSEN

- java.lang.System
  - getProperty() (HomeVerzeichnis, Tempverzeichnis, …)
  - getEnv()
  - lineSeparator()
  - Standard Streams (in, out, err)
  - exit()
  - gc()



1. - http://docs.oracle.com/javase/tutorial/essential/environment/sysprop.html
3. - Beendet das Programm
4. - Fordert die GC auf, sich in Gang zu setzen.
   - Wird im allgemeinen nicht benötigt und ist auch kein Garant, dass die GC startet.

# WICHTIGE UTILITY KLASSEN

- java.lang.Runtime
- java.lang.Math
- java.math.BigDecimal & BigInteger



1. - Starten und interagieren mit nativen Prozessen bzw. anderen Programmen.
2. - Die Methoden zur Fließkomma-Arithmetik, z.b. Winkel-Funktionen und Quadratwurzel.
3. - Klassen für beliebig große und genaue Zahlen für arithmetische Operationen.
   - Besonders im Banken-Breich sehr wichtig.
   Wiederholung: Wie viel passt in einen Long? 8 Byte

# NEUE DATE API


```java
public class NewDates {

    public static void main(String[] args) {
        Clock clock = Clock.systemUTC(); // UTC der Systemzeit

        Clock clock = Clock.systemDefaultZone(); // Uhrzeit in der Zeitzone des Systems

        long time = clock.millis(); // Millisekunden seit 01.01.1970

        ZoneId zone = ZoneId.of("Europe/Berlin"); // Zeitzone für Namen

        Clock clock = Clock.system(zone); // Setzt eine Zeitzone

        LocalDate date = LocalDate.now(); // LocalDate = menschenlesbare Zeitangaben
        String year = date.getYear();
        String month = date.getMonthValue();
        String day = date.getDayOfMonth();

        Period p = Period.of(2, HOURS);  // Zeitperioden
        LocalTime time = LocalTime.now();
        LocalTime newTime = time.plus(p);
    }
}
```

# DATEIEN UND VERZEICHNISSE

- Java bietet sehr umfangreiche API zum Datei- und Verzeichnishandling an
  - Handling von Dateien und Verzeichnisse selbst
  - Streams zum sequentiellen I/O
  - Random I/O
- java.io.*



3. - Streams sind eine objektorientierte Technik zur sequentiellen Ein- und Ausgabe von Dateiinhalten
4. - Der Zeilentrenner sollte niemals hardcodiert werden, sondern immer mit System.getProperty(„line.separator“) abgefragt werden.

# HANDLING VON DATEIEN

- Dateien und Verzeichnisse sind Objekte, nicht aber der Inhalt von Dateien

```
public FileBeispiel() {
    // Abstrahiert von Datei und Verzeichnis
    File homeVerzeichnis = new File("/Users/junterstein");
    File präsentation = new File(homeVerzeichnis.getAbsolutePath()
            + "/documents/java12.key");
    // Absolute und relative Namen unterstützt
    File testFile = new File("./test.jar");
    // Abfrage der Datei-/Verzeichnisattribute
    System.out.println(testFile.exists() + ":" + testFile.canRead()
            + ":" + testFile.canWrite());
    // Iterieren über Verzeichnisse
    for (File kind : homeVerzeichnis.listFiles()) {
        if (kind.isFile()) {
            // tue etwas
        }
        if (kind.isDirectory()) {
            // tue etwas anderes
        }
    }
    // Anlegen und Löschen von Dateien und Verzeichnissen
    testFile.delete();
    // mkdir() legt nur den angegebenen Ordner an, wenn der Vater existiert
    // mkdirs() legt alle Ordner an, bis der übergebene Pfad erreicht ist
    new File(homeVerzeichnis.getAbsolutePath()+"/mein/ausgedachter/Pfad").mkdirs();
    // Verwalten Temporärer Dateien
    try {
        File temporäreDatei = File.createTempFile("meinProjektName", "meineDateiEndung");
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

# JAVA.NIO.FILE.PATHS

- Pfadverkettung bitte mit Path und Paths

```
10  public class PathBeispiel {
11
12      public static void main(String[] args) {
13          File file = new File("mein/ausgedachter/Pfad");
14          File newFile = new File(file.getAbsolutePath() + "../../was/lustiges/dran"); // <-- unüblich
15          Path path = Paths.get(file.getAbsolutePath(), "../../was/lustiges/dran");
16          Path parent = path.getParent();
17          File parentFile = parent.toFile();
18          Path parent2 = parentFile.toPath();
19      }
```

# STREAMS

- Sehr abstraktes Konstrukt, zum Zeichen auf imaginäres Ausgabegerät zu schreiben oder von diesem zu lesen
  - Erste konkrete Unterklassen binden Zugriffsroutinen an echte Ein- oder Ausgabe
- Streams können verkettet und geschachtelt werden
  - Verkettung: Zusammenfassung mehrerer Streams zu einem
  - Schachtelung: Konstruktion von Streams mit Zusatzfunktion
- Bitte apache-commons verwenden: FileUtils, IOUtils, StreamUtils, …

Vergleiche: http://commons.apache.org



2. - Dateien, Strings, Netzwerkkommunikationskanäle, ...
4. - Verkettung Beispiel: mehrere Dateien als ein Stream behandeln mittels SequenceInputStream.
5. - Schachtelung: Konstruktion von Streams, die bestimmte Zusatzfunktionen übernehmen.
   - Am meisten verwendet: Puffern von Zeichen
   - BufferedInputStream(InputStream in)

- Beide Konzepte sind mit Java Sprachmitteln realisiert und können in eigenem Code erweitert werden.

# CHARACTER- UND BYTE-STREAMS

- Byte-Streams: Jede Transporteinheit genau 1 Byte lang
  - Problem bei Unicode (> 1 Byte) & Umständlich
- Character-Streams: Unicode-fähige textuelle Streams
- Wandlung von Character- und Byte-Streams und umgekehrt möglich



0. - Character Streams heißen Reader/Writer, Byte Streams heißen .. Streams (InputStream, OutputStream)
2. - Und die Transporteinheit steht nur als binäre Dateninformation zur Verfügung.

# CHARACTER-STREAMS FÜR DIE AUSGABE

- Abstrakte Klasse Writer
  - OutputStreamWriter
  - FileWriter
  - PrintWriter
  - BufferedWriter
  - StringWriter
  - CharArrayWriter
  - PipedWriter

```
private void charWriter() {
    String s;
    FileWriter fw = null;
    BufferedWriter bw = null;
    try {
        fw = new FileWriter("buffer.txt");
        bw = new BufferedWriter(fw);
        for (int i = 1; i <= 10000; ++i) {
            s = "Dies ist die " + i + ". Zeile";
            bw.write(s);
            bw.newLine();
        }
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        // Java Version < 7 Bedarf etwas
        // mehr aufwand zum stream closen
        try {
            if (bw != null) {
                bw.close();
            }
            if (fw != null) {
                fw.close();
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```



1. - Basis alles sequentiellen Character-Ausgaben.
2. - Basisklasse für alle Writer, die einen Character-Stream in einen Byte-Stream umwandeln.
3. - Writer zur Ausgabe in eine Datei als konkrete Ableitung von OutputStreamWriter
4. - Ausgabe von Textformaten.
5. - Writer zur Ausgabepufferung, um die Performance beim tatsächlichen Schreiben (der Datei) zu erhöhen.
6. - Writer zur Ausgabe eines String.
7. - Writer zur Ausgabe eines Streams in ein char-Array.
8. - Writer zur Ausgabe in einen PipedReader -> Linux Pipe Konstrukt.

# CHARACTER-STREAMS FÜR DIE EINGABE

- Abstrakte Klasse Reader
  - InputStreamReader
  - FileReader
  - BufferedReader
  - LineNumberReader
  - StringReader
  - CharArrayReader
  - PipedReader

```
private void charReader() {
    String line = null;
    try (BufferedReader br =
            new BufferedReader(new FileReader("config.sys"))) {
        while ((line = br.readLine()) != null) {
            System.out.println(line);
        }
        br.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```



1. - Basis aller sequentiellen Character-Eingaben.
2. - Basisklasse für alle Reader, die einen Byte-Stream in einen Cahracter-Stream umwandeln.
3. - Reader zum Einlesen aus einer Datei als konkrete Ableitung von InputStreamReader.
4. - Reader zur Eingabepufferung und zum Lesen kompletter Zeilen.
5. - Ableitung des BufferedReader mit der Fähigkeit, Zeilen zu zählen.
6. - Reader zum Einlesen von Zeichen aus einem String.
7. - Reader zum Einlesen von Zeichen aus einem Char-Array.
8. - Reader zum Einlesen von Zeichen aus einem PipedWriter -> Beispiel 1 Slide vorher.

# BYTE-STREAMS

- Bei Character-Streams wird von Readern und Writern (analog zur jeweiligen Basisklasse) gesprochen, hier spricht man von Byte-Streams, also InputStreams und OutputStreams
- Abstrakte Klasse InputStream und OutputStream
- Funktionieren genauso wie Character-Streams, arbeiten jedoch auf Bytes
- Spezialfall: ZipOutputStream und ZipInputStream im Paket java.util.zip zum Schreiben und lesen von Archivdateien

Vergleiche: http://commons.apache.org



2. - Inklusive einer analogen Klassenhierarchie wie bei Writern und Readern.

# RANDOM I/O

- Streams vereinfachen den sequentiellen Zugriff auf Dateien
- Manchmal wahlfreier Zugriff auf Dateien notwendig
- RandomAccessFile stellt entsprechende Methoden zur Verfügung um in Dateien zu navigieren/lesen/schreiben

# INTROSPECTION & REFLECTION

- Möglichkeit, zur Laufzeit Klassen zu instanziieren und zu verwenden ohne diese zur Compilezeit zu kennen
- Abfragemöglichkeiten, welche Member eine Klasse besitzt
- Reflection ist ein sehr mächtiges Werkzeug, sollte aber mit Bedacht eingesetzt werden
  - Reflection Code schlägt oft erst zur Laufzeit fehl
  - Macht Code schwer lesbar



1. - Wichtig zum Beispiel für Anwendungen mit PlugIn-Schnittstellen
   - Prominentestes Beispiel: Eclipse

# INTROSPECTION & REFLECTION

- Die Klasse java.lang.Class

```
try {
    String meinKlassenName = "ba.java.auto.AudiQFuenf";
    Class<?> meineKlasse = Class.forName(meinKlassenName);
    Object meinObjekt = meineKlasse.newInstance();
    AudiQFuenf meinAudi = (AudiQFuenf) meinObjekt;
} catch (Exception e) {
    e.printStackTrace();
}
```

  - Erreichbar über getClass() der Klasse Object
  - Methoden zur Abfrage der Member der Klasse, sowie weiterer Eigenschaften

```
try {
    String str = "Test";
    Class<?> clazz = str.getClass();
    Method m = clazz.getDeclaredMethod("length");
    Object ret = m.invoke(str);
    System.out.println(ret);
} catch (Exception e) {
    e.printStackTrace();
}
```



3. - Über die Relections API sind ganz hässliche Hacks möglich.
-> Es können modifier von Klassen verändert werden und auch auf private Member zugegriffen werden (lesen und schreibend).
- Welches Tool, was ihr alle schon benutzt habt, verwendet ganz Exzessiv die Reflections API und die Möglichkeit auf private Member zuzugreifen?
-> Jeder Debugger.

# ÜBUNGEN

- https://github.com/unterstein/dhbw-java-lecture/tree/master/src/ba/java/uebungen
  - https://git.io/vrF8U

# SERIALISIERUNG

- Fähigkeit, ein Objekt im Hauptspeicher in ein Format zu konvertieren, um es in eine Datei zu schreiben oder über das Netzwerk zu transportieren
- Deserialisierung: Umkehrung der Serialisierung in ein Objekt
- Manchmal wird Serialisierung zur Persistenz eingesetzt
  - Meist nicht die klassische Serialisierung, sondern XML, JSON
- Standard Serialisierung ist binärbasiert!



1. - Bei Client/Server Anwendungen spielt Serialisierung oft eine Rolle, um Objekte zwischen Client und Server zu transportieren.
   - Das Objekt wird in einem binären Format gespeichert, also nicht für den Menschen lesbar!
3. - In kleineren Anwendungen ohne Datenbank werden Objekte halt in eine Datei geschrieben und können aus dieser wieder geladen werden.
4. - XML bietet eigene Stream Writer und Reader mit Java Sprachmitteln.
5. - Nachteile von Binärserialisierung (Auszug „Java ist auch eine Insel“):

Der klassische Weg von einem Objekt zu einer persistenten Speicherung führt über den Serialisierungsmechanismus von Java über die Klassen ObjectOutputStream und ObjectInputStream. Die Serialisierung in Binärdaten ist aber nicht ohne Nachteile. Schwierig ist beispielsweise die Weiterverarbeitung von Nicht-Java-Programmen oder die nachträgliche Änderung ohne Einlesen und Wiederaufbauen der Objektverbunde. Wünschenswert ist daher eine Textrepräsentation. Diese hat nicht die oben genannten Nachteile.

Ein weiteres Problem ist die Skalierbarkeit. Die Standard-Serialisierung arbeitet nach dem Prinzip: Alles, was vom Basisknoten aus erreichbar ist, gelangt serialisiert in den Datenstrom. Ist der Objektgraph sehr groß, steigt die Zeit für die Serialisierung und das Datenvolumen an. Verglichen mit anderen Persistenz-Konzepten, ist es nicht möglich, nur die Änderungen zu schreiben. Wenn sich zum Beispiel in einer sehr großen Adressliste die Hausnummer einer Person ändert, muss die gesamte Adressliste neu geschrieben werden – das nagt an der Performance.

# XML SUPPORT

- eXtensible Markup Language
- XML ist ein semi-strukturiertes Datenformat
- Es können in XML eigene Strukturen abgebildet werden, die jedoch nur mit eigenem allgemeinen Parser verarbeitet werden können
- Java liefert DOM und SAX Parser mit und bietet Möglichkeiten zur Transformation von XML Dokumenten



4. - Unterschied DOM zu SAX?
   - DOM baut den gesamten XML Baum erstmal im RAM auf und dann kann bequem auf dem Baum navigiert und gearbeitet werden.
   - SAX ließt das XML sequentiell ein und schmeißt Events, wenn bestimmte Sachverhalte eintreten.
   -> SAX ist unbequemer, aber wesentlich speicherfreundlicher.
0. - Es ist allerdings die Verwendung von JAXB zu empfehlen.
   - Java Architecture for Xml Binding
   - Erlaubt es normale Java Klassen per Annotation automatisch in XML zu serialisieren und zu deserialisieren.

# ANNOTATIONS

- Möglichkeit Java Code mit Meta Informationen zu versehen
- Nur ganz kurze Einführung in Zusammenhang mit JAXB!
- @Override, ...

```
9  public class AnnotationBeispiel {
10 
11     public AnnotationBeispiel() {
12         UserBean bean = new UserBean();
13         bean.name = "James";
14         bean.surename = "Hetfield";
15         bean.birthDate = new Date(63, 7, 3); // 3. August 1963
16         // XMLSerializeHelper ist kein Bestandteil von Java
17         System.out.println(XMLSerializeHelper.instance().serialize(bean));
18     }
19 
20     @XmlRootElement
21     public static class UserBean implements Serializable {
22         private static final long serialVersionUID = -9091053716626960391;
23 
24         @XmlElement(name = "vorname")
25         public String name;
26         public String surename;
27         @XmlElement(name = "geburtstag")
28         public Date birthDate;
```

Console

```
<terminated> AnnotationBeispiel [Java Application] /Library/Java/JavaVirtualMachines/1.7.0.jdk/Contents/H
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<userBean>
    <vorname>James</vorname>
    <surename>Hetfield</surename>
    <geburtstag>1963-08-03T00:00:00+01:00</geburtstag>
</userBean>
```

# JDBC

- Java Database Connectivity
- Schnittstelle zwischen einer Applikation und einer SQL-DB
- Es sind weitere Treiber der Datenbank-Hersteller notwendig



3. - Java gibt nur die Schnittstelle vor, aber nicht die Datenbank-abhängige Implementierung.
0. - Weitere OR-Mapper sind nicht im Sprachkern enthalten.
- Die javax.persistence API kommt in der JavaEE mit und die Implementierung von ... mehr oder weniger guten Drittherstellern ;)

# NETZWERK

- Java stellt API zur Socket-Programmierung via TCP/IP bereit
- Über Streams ist das Lesen und Schreiben möglich
- Dies ist im Allgemeinen umständlich, weshalb meist andere Mechanismen verwendet werden



0. - Welche Möglichkeiten über das Netzwerk zu kommunizieren sind etabliert und verbreitet?
- Webservices - REST/SOAP - Wobei REST schon besser ist :)

# RMI

- Remote Method Invocation
- Socket Programmierung erlaubt Austausch von Daten
  - RMI erlaubt transparente Methodenaufrufe auf Server
- RMI abstrahiert
- RMI ist nur eine Möglichkeit für verteilte Objekte



4. - Dem Verwender eines Objektes kommt es vor, als würde er auf einem lokalen Objekt arbeiten.
   - Java RMI kümmert sich um die Übertragung der Objekte über das Netzwerk.
5. - CORBA, ...
   - Der Ansatz von Webservices adressiert ein viel größeres Feld an Möglichkeiten, wie es die bloße RMI kann.
   - Die Vorlesung verteilte Systeme im 4. oder 5. Semester beschäftigt sich sehr intensiv damit --> Aufpassen :)

# SICHERHEIT UND KRYPTOHRAPHIE

- Daten müssen oft verschlüsselt werden
  - Verschlüsselung von Dateien
  - Verschlüsselung von Daten bei Netzwerktransport
- Symmetrische, Asymmetrische Verfahren und Signaturen unterstützt



2. - Cipher cipher = Cipher.getInstance(algorithmName);

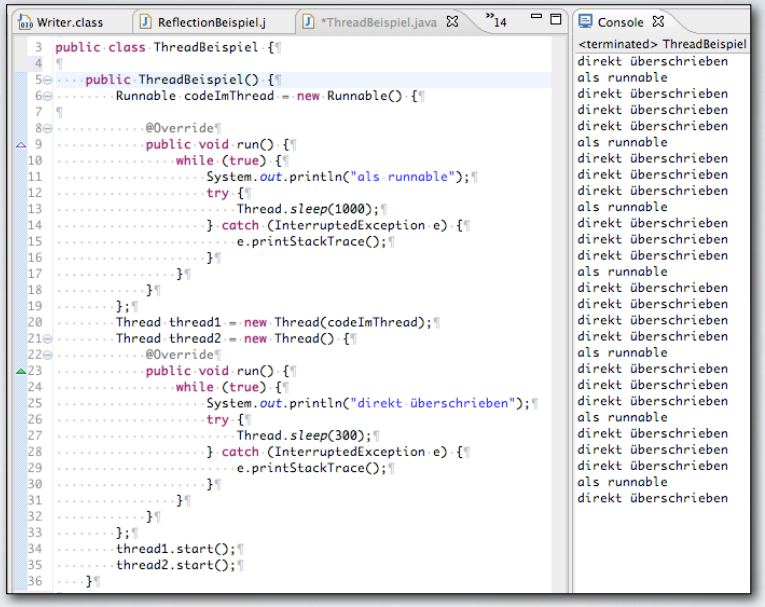
# MULTITHREADING

- Konzepte der Nebenläufigkeit von Programmteilen
  - Ähnelt einem Prozess, arbeitet aber auf einer feineren Ebene
- Threads sind in Java direkt als Sprachkonstrukt umgesetzt
- Threads können synchronisiert werden



1. - Beispiel: GUI Thread mit Sound abspielen, Eingabe erfassen und Eingabe validieren
2. - Vergleiche Vorlesung „Betriebssysteme“
4. - Schlüsselwort synchronized

# MULTITHREADING BEISPIEL




```
3  public class ThreadBeispiel {
4
5      public ThreadBeispiel() {
6          Runnable codeImThread = new Runnable() {
7
8              @Override
9              public void run() {
10                 while (true) {
11                     System.out.println("als runnable");
12                     try {
13                         Thread.sleep(1000);
14                     } catch (InterruptedException e) {
15                         e.printStackTrace();
16                     }
17                 }
18             }
19         };
20         Thread thread1 = new Thread(codeImThread);
21         Thread thread2 = new Thread() {
22             @Override
23             public void run() {
24                 while (true) {
25                     System.out.println("direkt überschrieben");
26                     try {
27                         Thread.sleep(300);
28                     } catch (InterruptedException e) {
29                         e.printStackTrace();
30                     }
31                 }
32             }
33         };
34         thread1.start();
35         thread2.start();
36     }
```