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

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);
}
```

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);
}
```

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);
}
```

WICHTIGSTE METHODEN VON STRINGS

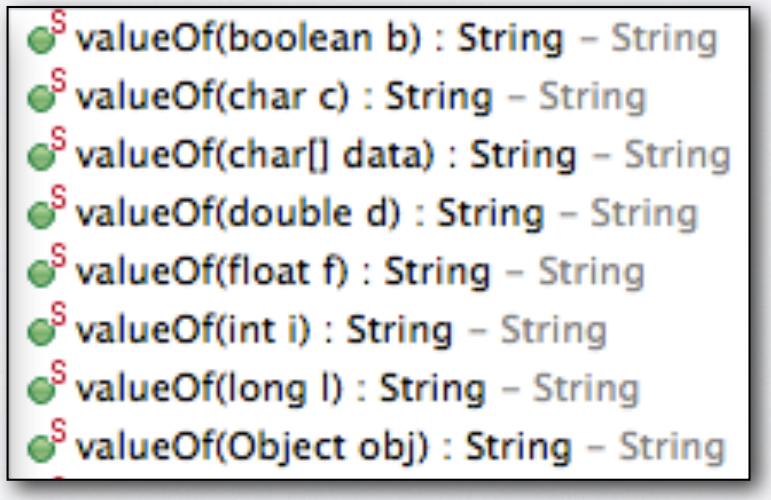
```
public StringBeispiel4() {
    String string1 = "Hallo!";

    // 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);
}
```

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);¶
}¶
```

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

STRINGBUFFER/-BUILDER

- 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);
}
```

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

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

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

- Fehler**objekte**
  - 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)
```

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

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

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;
}
```

TRY WITH RESOURCES


```
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
    }
}
```

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 sein
  - 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

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!

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

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;
    }
}
```

GENERICS

```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
1  package ba.java.weiteres.generics;
2
3  public class AlteListeVerwendung {
4
5    public AlteListeVerwendung() {
6      // Hmm...eigentlich will ich nur Integer zulassen, geht aber nicht :-(
7      AlteListe liste = new AlteListe(10);
8      liste.addElement(5);
9      liste.addElement(1.5); // führt zu keinem Compiler-Fehler!
10
11     // Der Rückgabetyp ist Object
12     Object objectFromListe = liste.elementAt(0);
13     Integer integerFromListe = (Integer) objectFromListe;
14   }
15 }
16
17
18
19
20
21
22
23
24
25
26
```

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

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)

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
```

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());
    }
}
```

PROJECT COIN - DIAMOND OPERATOR


```
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

- Anonyme 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, …

```
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