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 benützt 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.*
- Welche Arten von Collections gibt es?
  - Seit Java 5 sind die Collections generisch
  - Namens Konvention: <Stil><Interface>
- Wichtigste Interfaces: List, Set, Map, Queue
- Tree vs. Hash Implementierungen

4. - Beispiel: ArrayList, Array ist der Stil und List das Interface
HashMap, Hash ist der Stil und Map das Interface
5. - 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
6. - 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.
Herr Stroetmann wird dort allerdings noch viel Input liefern!

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

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

```
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 (1)

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

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

  - java.util.GregorianCalendar und java.util.Date

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)

WICHTIGE UTILITY KLASSEN (2)

- java.lang.System
  - getProperty() (Zeilentrenner, Tempverzeichnis, ...)
  - 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 (3)

- 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

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();¶
    }¶
}¶
```

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 (1)

- 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 (2)

- Die Klasse java.lang.Class
  - Erreichbar über getClass() der Klasse Object
  - Methoden zur Abfrage der Member der Klasse, sowie weiterer Eigenschaften

```
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();¶
}¶
```

```
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.

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
- 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, ...

```
public class AnnotationBeispiel {

    public AnnotationBeispiel() {
        UserBean bean = new UserBean();
        bean.name = "James";
        bean.surename = "Hetfield";
        bean.birthDate = new Date(63, 7, 3); // 3. August 1963
        // XMLSerializeHelper ist kein Bestandteil von Java
        System.out.println(XMLSerializeHelper.instance().serialize(bean));
    }

    @XmlRootElement
    public static class UserBean implements Serializable {
        private static final long serialVersionUID = -9091053716626960391L;

        @XmlElement(name = "vorname")
        public String name;
        public String surename;
        @XmlElement(name = "geburtstag")
        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
- Synchrone, Asynchrone Verfahren und Signaturen unterstützt

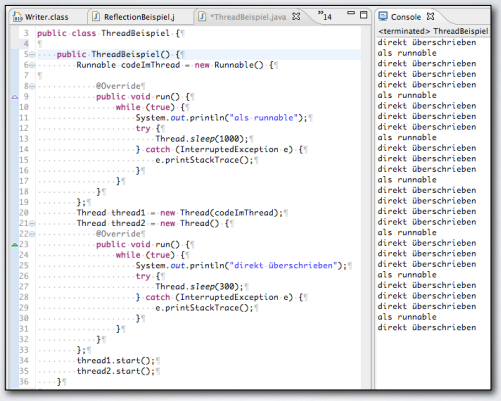
2. - Cipher cipher = Cipher.getInstance(algorithmName);

MULTITHREADING

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

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

MULTITHREADING BEISPIEL



JAVA 8

Grafik von http://javafx.com

AGENDA

- Lambdas
- Methodenreferenzen
- Parallel Sort
- Bulk Operations
- Neue Date API

JAVA 8

- Rahmen dieser Vorstellung:
  - Was gibt es wichtiges neues
  - Beispielhafte Nutzung

LAMBDAS

- Innere Klassen mit neuer Syntax zu verwenden

```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);
    }
}
```

PARALLEL SORT

- Parallele Sortierung eines java.util.Arrays

```
parallelSort(byte[] a)
parallelSort(byte[] a, int fromIndex, int toIndex)
parallelSort(char[] a)
parallelSort(char[] a, int fromIndex, int toIndex)
parallelSort(double[] a)
parallelSort(double[] a, int fromIndex, int toIndex)
parallelSort(float[] a)
parallelSort(float[] a, int fromIndex, int toIndex)
parallelSort(int[] a)
parallelSort(int[] a, int fromIndex, int toIndex)
parallelSort(long[] a)
parallelSort(long[] a, int fromIndex, int toIndex)
parallelSort(short[] a)
parallelSort(short[] a, int fromIndex, int toIndex)
parallelSort(T[] a)
parallelSort(T[] a, Comparator<? super T> c)
parallelSort(T[] a, int fromIndex, int toIndex)
parallelSort(T[] a, int fromIndex, int toIndex, Comparator<? super T> c)
```


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) { ... }
}
```

NEUE DATE API

```
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

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

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

        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);
    }
}
```


HAPPY HACKING