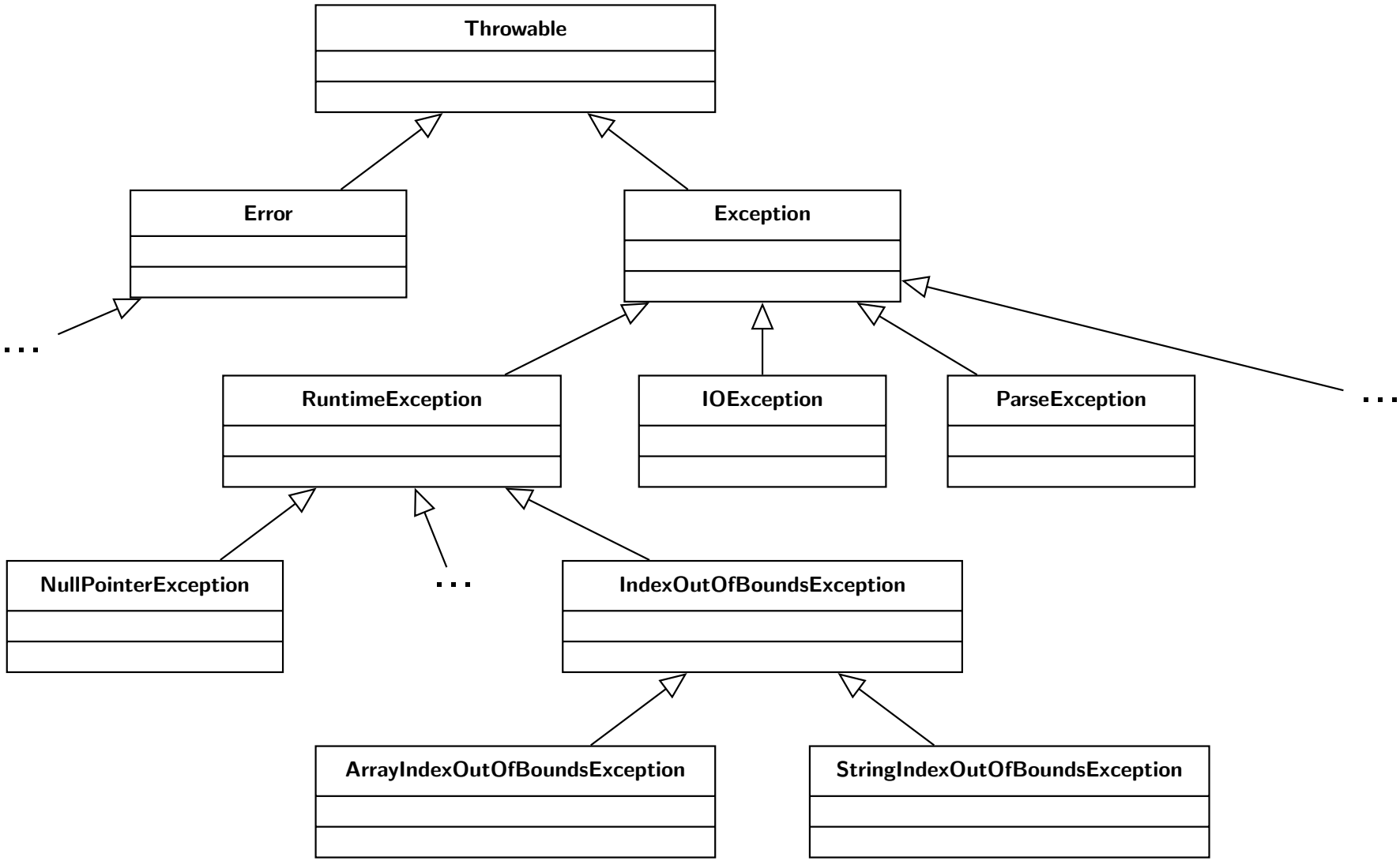
TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Software Entwicklung 1

Annette Bieniusa

AG Softech
FB Informatik
TU Kaiserslautern

Die Klassenhierarchie der Exceptions (Ausschnitt)



Klassifizierung von Ausnahmen und Fehlern

- Die Klasse `Throwable` aus dem Packet `java.lang` ist die Superklasse aller Ausnahmen- und Fehlerklassen in Java.
- `Error`s sind schwerwiegende Fehler, die in der Regel nicht vom Programm abgefangen und behandelt werden sollten.
- `Exception`s sollten vom Programm abgefangen und geeignet behandelt werden.

Beispiel: Ausnahmebehandlung

```
1  public class UeberlaufException extends Exception {}
2
3  public class ExceptionTest {
4   public static void main(String[] args) {
5    try {
6     int ergebnis = 0;
7     int m = Integer.parseInt(args[0]);
8     int n = Integer.parseInt(args[1]);
9     long aux = (long) m + (long) n;
10    if (aux > Integer.MAX_VALUE) {
11       throw new UeberlaufException(); // selbstdefiniert
12    }
13    ergebnis = (int) aux;
14   } catch (IndexOutOfBoundsException e) {
15      System.out.println("Zuwenig Argumente");
16   } catch (NumberFormatException e) {
17      System.out.println("Parameter ist keine int-Konstante");
18   } catch (UeberlaufException e) {
19      System.out.println("Ueberlauf bei Addition");
20   }
21  }
22
```

Reihenfolge der Exception-Handler

- Ein Handler für Exceptions einer Klasse `X` behandelt auch Exceptions aller von `X` abgeleiteten Klassen (Stichwort: Subtyp-Polymorphie).
- Die Reihenfolge der `catch`-Blöcke ist daher relevant bei der Ausnahmebehandlung.
- Zunächst müssen die Handler für die am meisten spezialisierten Klassen aufgelistet werden und dann mit zunehmender Generalisierung die Handler für die entsprechenden Superklassen. Dies wird vom Compiler überprüft.

```
3   public class UeberlaufException extends Exception {
4     UeberlaufException() {
5       super("Ueberlauf bei der Integer-Addition");
6     }
7   }
8   public class ExceptionTest {
9    public static void main(String[] args) {
10    try {
11     int ergebnis = 0;
12     int m = Integer.parseInt(args[0]);
13     int n = Integer.parseInt(args[1]);
14     long aux = (long) m + (long) n;
15     if (aux > Integer.MAX_VALUE) {
16        throw new UeberlaufException(); // selbstdefiniert
17     }
18     ergebnis = (int) aux;
19    } catch (ArrayIndexOutOfBoundsException e) {
20       System.out.println("Zuwenig Argumente");
21    } catch (Exception e) {
22       System.out.println("Ein Fehler ist aufgetreten");
23       e.getMessage();
24    }
25   }
```

Beispiel: Checked Exceptions

```
3  public static void main(String[] args) throws Exception {
4      double[] a = {};
5      System.out.println("avg = " + average(a));
6  }
7  static double average(double[] a) throws Exception {
8      checkPreconditions(a);
9      int sum = 0;
10     for (int i = 0; i < a.length; i++) {
11         sum += a[i];
12     }
13     return sum / a.length;
14 }
15 static void checkPreconditions(double[] a) throws Exception {
16     if (a == null) {
17         throw new Exception("Array darf nicht null sein");
18     }
19     if (a.length == 0) {
20         throw new Exception("Array darf nicht leer sein");
21     }
22 }
23
```

Checked und Unchecked Exceptions

- In Java zeigt die `throws`-Klausel in einer Methodensignatur an, dass eine Methode Exceptions auslöst bzw. weitergibt, ohne sie selbst abzufangen.
- **Checked Exceptions** müssen, wenn sie in einer Methode geworfen werden, entweder dort in einem Exception Handler behandelt werden, oder aber in einer `throws`-Klausel in der Methodensignatur angegeben werden. Dies wird vom Compiler überprüft.
- **Unchecked Exceptions** müssen nicht entsprechend deklariert werden. Nur Exceptions vom Typ `RuntimeException` und `Error` sind unchecked.

Ausgabe im Fehlerfall: Stacktraces

Die Ausführung führt zu einem Programmabbruch, weil die Ausnahme nicht in einem `try`-Block behandelt wurde.

Als Fehlermeldung wird ein sogenannter *Stacktrace* angezeigt:

```
Exception in thread "main" java.lang.Exception: Array darf nicht leer sein
    at ExceptionTest.checkPreconditions(ExceptionTest.java:20)
    at ExceptionTest.average(ExceptionTest.java:8)
    at ExceptionTest.main(ExceptionTest.java:5)
```

Vorteile von Exceptions

- Standardfall in der Ausführung von der Fehlerbehandlung syntaktisch getrennt
  $\Rightarrow$ **Exceptions nur zur Fehlerbehandlung verwenden!**
- Fehlerbehandlung wird an Aufrufer weitergegeben, da dieser evtl. besser reagieren kann
- Unterscheidung und Gruppierung von verschiedenen Fehlertypen möglich

Ein- und Ausgabe von Daten

Eingaben verarbeiten

Verschiedene Vorgehensweisen:

1. Die komplette Eingabe wird auf einmal bis zum Ende gelesen
   - als `String`
   - als `byte`-Array
2. Die Eingabe wird Stückweise gelesen und zur Verfügung gestellt.

Beispiel: Suchen und Ersetzen

```
public static void main(String[] args) throws IOException {
  String searchString = args[0];
  String replaceString = args[1];
  Path inFile = Paths.get(args[2]);
  Path outFile = Paths.get(args[3]);

  // Daten aus Eingabe-Datei lesen
  byte[] bytes = Files.readAllBytes(inFile);

  // Bytes in String umwandeln (mit UTF-8 Kodierung)
  String str = new String(bytes, StandardCharsets.UTF_8);

  // searchString durch replaceString ersetzen
  String newString = str.replace(searchString, replaceString);

  // String in Bytes umwandeln (mit UTF-8 Kodierung)
  byte[] newBytes = newString.getBytes(StandardCharsets.UTF_8);

  // Ergebnis in Ausgabe-Datei schreiben
  Files.write(outFile, newBytes);
}
```

Ströme zur Ein- und Ausgabe

Ströme zur Ein- und Ausgabe I

Begriffsklärung: Datenstrom

Ein **Strom** (engl. *Stream*) ist eine potentiell unendliche Folge von Daten.

Er wird von einer oder mehrerer Quellen mit Daten versorgt und erlaubt es, diese Daten der Reihe nach aus dem Strom herauszulesen.

Das Ende eines Stromes kann durch ein spezielles Datum markiert werden (z.B. in Java bei `char`-Strömen -1).

Ströme zur Ein- und Ausgabe II

Bei Operationen auf einem Strom kann es zu Verzögerungen kommen:

- beim Lesen, weil augenblicklich kein Zeichen vorhanden, der Strom aber noch nicht zu Ende ist;
- beim Schreiben, weil evtl. kein Platz im Strom vorhanden ist.

Die Verzögerungen führen zu einer Blockierung der ausgeführten Methode.

Einführung in Ströme I

Wir betrachten zunächst Ströme zum Lesen:

```
interface CharEingabeStrom {
    int read() throws IOException;
}
```

`read` liefert bei jedem Aufruf den nächsten `char`-Wert im Strom (Zahl zwischen 0 und 65535) und `-1` wenn Ende des Stroms erreicht ist.

Einführung in Ströme II

Diese Schnittstelle abstrahiert von der Quelle aus der gelesen wird. Mögliche Quellen:

1. Datenstruktur wie Array, Liste, String
2. Datei
3. Netzwerk
4. Standardeingabe, z.B. interaktive Eingabe vom Anwender
5. andere Programme
6. andere Ströme

Wir betrachten hier die Fälle 1, 2 und 6.

Lesen aus einer Datenstruktur

Wir betrachten das schrittweise Lesen der Zeichen eines Strings. Die Quelle des Stroms wird dem Konstruktor übergeben.

```
public class StringLeser implements CharEingabeStrom {
  private String zeichen;
  private int    index;

  public StringLeser(String s) {
    zeichen = s;
    index = 0;
  }

  public int read() {
    if (index == zeichen.length()) {
      return -1;
    } else {
      index++;
      return zeichen.charAt(index - 1);
    }
  }
}
```

Zusammenbauen von Strömen

Wir betrachten zwei Stromklassen, die aus anderen Strömen lesen und die Ströme modifizieren.

Die Konstruktoren nehmen dabei einen beliebigen `CharEingabeStrom` als Quelle:

$\rightarrow$ Subtyping at its best!

Zusammenbauen von Strömen

```
public class GrossBuchstabenFilter
                    implements CharEingabeStrom {
   private CharEingabeStrom eingabeStrom;

   public GrossBuchstabenFilter(CharEingabeStrom cs) {
      eingabeStrom = cs;
   }

   public int read() throws IOException {
      int z = eingabeStrom.read();
      if( z == -1 ) {
         return -1;
      } else {
         return Character.toUpperCase( (char)z );
      }
   }
}
```

Zusammenbauen von Strömen

```
public class UmlautSzFilter implements CharEingabeStrom {
   private CharEingabeStrom eingabeStrom;
   private int puffer = -1;

   public UmlautSzFilter( CharEingabeStrom cs ){
      eingabeStrom = cs;
   }

  ...
```

```
public int read() throws IOException {
    if( puffer != -1 ) {
        int z = puffer;
        puffer = -1;
        return z;
    } else {
        int z = eingabeStrom.read();
        if (z == -1) return -1;
        switch ((char) z) {
          case 'Ä':  puffer = 'e'; return 'A';
          case 'Ö':  puffer = 'e'; return 'O';
          case 'Ü':  puffer = 'e'; return 'U';
          case 'ä':  puffer = 'e'; return 'a';
          case 'ö':  puffer = 'e'; return 'o';
          case 'ü':  puffer = 'e'; return 'u';
          case 'ß':  puffer = 's'; return 's';
          default :                return z;
        }
    }
}
```

Verwendung zusammengebauter Ströme

```
public class StreamTest {
  public static void main(String[] args) throws IOException {
    String s = "Äneas opfert den Göttern edle Öle,\n"
             + "auf daß überall das Übel sich ändert.";

    CharEingabeStrom cs = new StringLeser(s);
    cs = new UmlautSzFilter(cs);
    cs = new GrossBuchstabenFilter(cs);
    int z = cs.read();
    while (z != -1) {
       System.out.print((char) z);
       z = cs.read();
    }
    System.out.println();
  }
}
```

Javas Stromklassen I

Stromklassen werden nach den Datentypen, die sie verarbeiten, und ihren Datenquellen bzw. -senken klassifiziert.

Stromklassen sind wichtige programmiertechnische Hilfsmittel.

Die Reader-/Writer-Klassen aus dem Paket `java.io` verarbeiten `char`-Ströme; die Input-/Output-Stromklassen verarbeiten `byte`-Ströme.

Javas Stromklassen II

Die Reader-Klassen unterstützen:

- das Lesen einzelner Zeichen: `int read();`
- das Lesen mehrerer Zeichen aus der Quelle und Ablage in ein char-Array: `int read(char[]);`
- das Überspringen einer Anzahl von Zeichen der Eingabe: `long skip(long);`
- die Abfrage, ob der Strom für das Lesen des nächsten Zeichens bereit ist;
- das Schließen des Eingabestroms: `void close();`
- Methoden zum Markieren und Zurücksetzen des Stroms.

Javas Stromklassen III

Die Writer-Klassen unterstützen:

- das Schreiben einzelner Zeichen:

```
void write( int );
```

- das Schreiben mehrerer Zeichen eines char-Arrays:

```
void write(char[])
```

u. ä.;

- das Schreiben mehrerer Zeichen eines String:

```
void write(String)
```

u. ä.;

- die Ausgabe ggf. im Strom gepufferter Zeichen:

```
void flush() ;
```

Javas Stromklassen IV

- das Schließen des Ausgabestroms:

```
void close()
```

Die genannten Methoden lösen möglicherweise eine `IOException` aus.

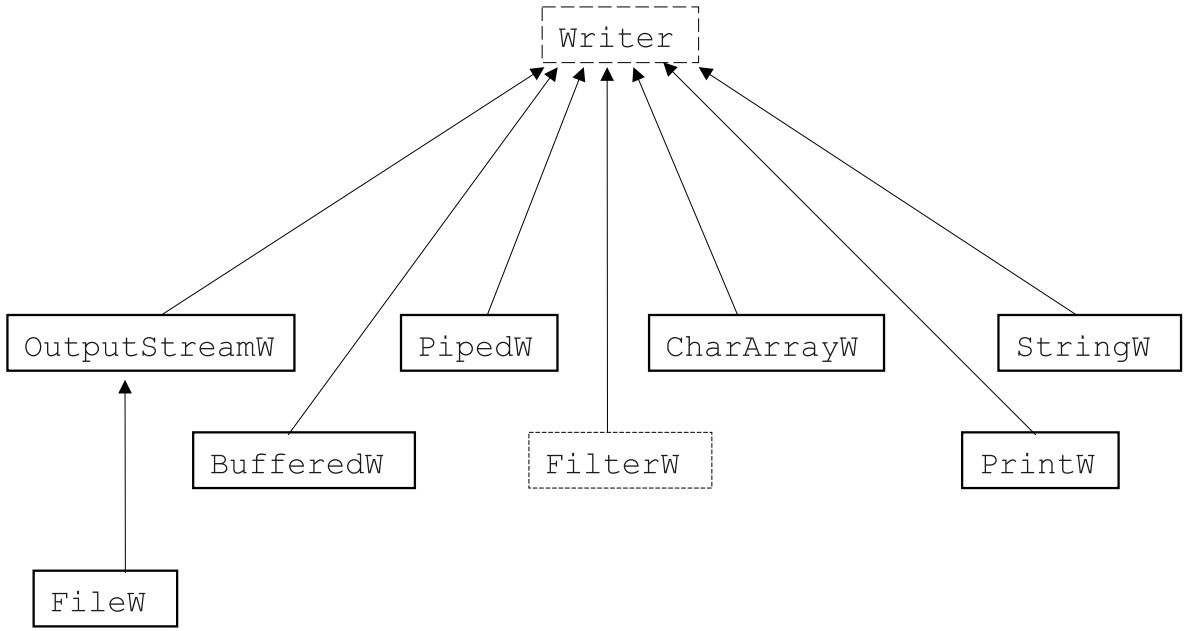
Die von `InputStream` bzw. `OutputStream` abgeleiteten Klassen leisten Entsprechendes für Daten vom Typ `byte`.

Reader-/Writer-Klassen I

Die Reader-Klassen unterscheiden sich im Wesentlichen durch ihre Quelle:

Reader-Klasse	Quelle	Bemerkung
InputStreamReader	InputStream	
FileReader	byte-Strom aus Datei	
BufferedReader	Reader	*puffernd; können*
LineNumberReader	Reader	*zeilenweise lesen*
PipedReader	PipedWriter	
FilterReader	Reader	
PushBackReader	Reader	Methode `unread`
CharArrayReader	char[]	
StringReader	String	

Reader-/Writer-Klassen II



Writer arbeiten analog zu Reader-Klassen, nur in umgekehrter Richtung.

`PrintWriter` unterstützen die formatierte Ausgabe von Daten durch die Methoden `print` und `println`, die alle Standarddatentypen als Parameter nehmen.

Bemerkung

Die Konstruktoren ermöglichen das Zusammenhängen von Strömen; hier am Beispiel eines Konstruktors der Klasse `PrintWriter`:

```
public PrintWriter(OutputStream o, boolean af) {
    this(new BufferedWriter(
            new OutputStreamWriter(o)), af);
}
```

Schließen von Strömen I

Ströme müssen geschlossen werden, wenn sie nicht mehr verwendet werden. Um dies sicherzustellen, wenn während der Verwendung des Stroms eine Exception auftritt, muss die Anweisung in einem `try-finally` Block verwendet werden.

```
BufferedReader reader = null;
try {
    reader = new BufferedReader(...);
    // Strom verwenden
} finally {
    if (reader != null) {
        reader.close();
    }
}
```

Der `finally`-Block wird immer nach dem `try`-Block ausgeführt.

Schließen von Strömen II

Java 7 bietet eine alternative Syntax für die Deklaration und Verwendung von Strömen an:

```
try (BufferedReader reader = new BufferedReader(...)) {
    // Strom verwenden
}
```

Dieses Konstrukt sorgt dafür, dass der Strom automatisch am Ende des `try`-Blocks geschlossen wird.

Beispiel: Suchen und Ersetzen

- Ersetzt zeilenweise alle Vorkommen von String X durch Y

Ein- und Ausgabe von Objekten

Ein- und Ausgabe von Objekten I

Wie wandelt man Objekte in `byte`- und `char`-Werte um?

Serialisieren bedeutet einen Wert eins Typs in Bytes umzuwandeln.

Deserialisieren bezeichnet den umgekehrten Prozess.

Bisweilen schwierig und komplex:

- Nicht alle Attribute eines Objekts müssen gespeichert werden (zum Beispiel Zwischenspeicher, Indexstrukturen, etc).
- Manchmal ist es nicht ausreichend alle Attribute eines Objekts zu speichern (der Zustand könnte von externen Ressourcen oder dem aktuellen Systemzustand abhängig sein).
- Objektreferenzen besitzen nur innerhalb des aktuellen Prozesses eine Gültigkeit.
- Bei Objekten ist häufig ihre Rolle im Objektgeflecht von entscheidender Bedeutung.

Ein- und Ausgabe von Objekten II

Andererseits ist Ein- und Ausgabe von Objekten wichtig, um

- Objekte zwischen Prozessen auszutauschen;
- Objekte für nachfolgende Programmläufe zu speichern, d.h. **persistent** zu machen.

Beispiel: Ausgabe von Objekten I

```
public class TodoList {
    private List<Task> tasks;
    public TodoList() {
        this.tasks = new LinkedList<>();
    }
    public void addTask(String desc, Date da) {...}
}

public class Task {
    private String description;
    private Date date;
    ... // Getter und Setter
}

TodoList tdl = new TodoList();
Date heute = new Date(16, 1, 2018);
tdl.add("Lernen", heute);
tdl.add("Einkaufen", heute);
tdl.add("Lernen", new Date(17, 1, 2018));
```

Beispiel: Ausgabe von Objekten II

Was bedeutet es, das von `tdl` referenzierte Objekt auszugeben?

- nur das `TodoList`-Objekt ausgeben?
- das `LinkedList`-Objekt (`tasks`) ausgeben?
- die `Entry`-Objekte der `LinkedList` ausgeben?
- die zugehörigen `Task`-Objekte ausgeben?
- die `Date`-Objekte in den `Task`-Objekten ausgeben?
- das von `heute` referenzierte Objekt einmal oder zweimal ausgeben?

Ausgabe von Objektgeflechten

Um Objekte in ihrem Zusammenwirken mit anderen Objekten wieder einlesen zu können, müssen sie gemeinsam mit allen erreichbaren Objekten ausgegeben werden.

Wegen möglicher Zyklen ist die Implementierung der Ausgabe und des Einlesens von Geflechten nicht einfach.

Bemerkung

- Gibt man ein Objekt mit den erreichbaren Objekten aus und liest es wieder ein, entsteht eine Kopie.
- Referenziert man von mehreren Variablen Teile des gleichen Geflechts, kommt es beim Einlesen ggf. zu mehreren Kopien eines Objekts des ursprünglichen Geflechts.

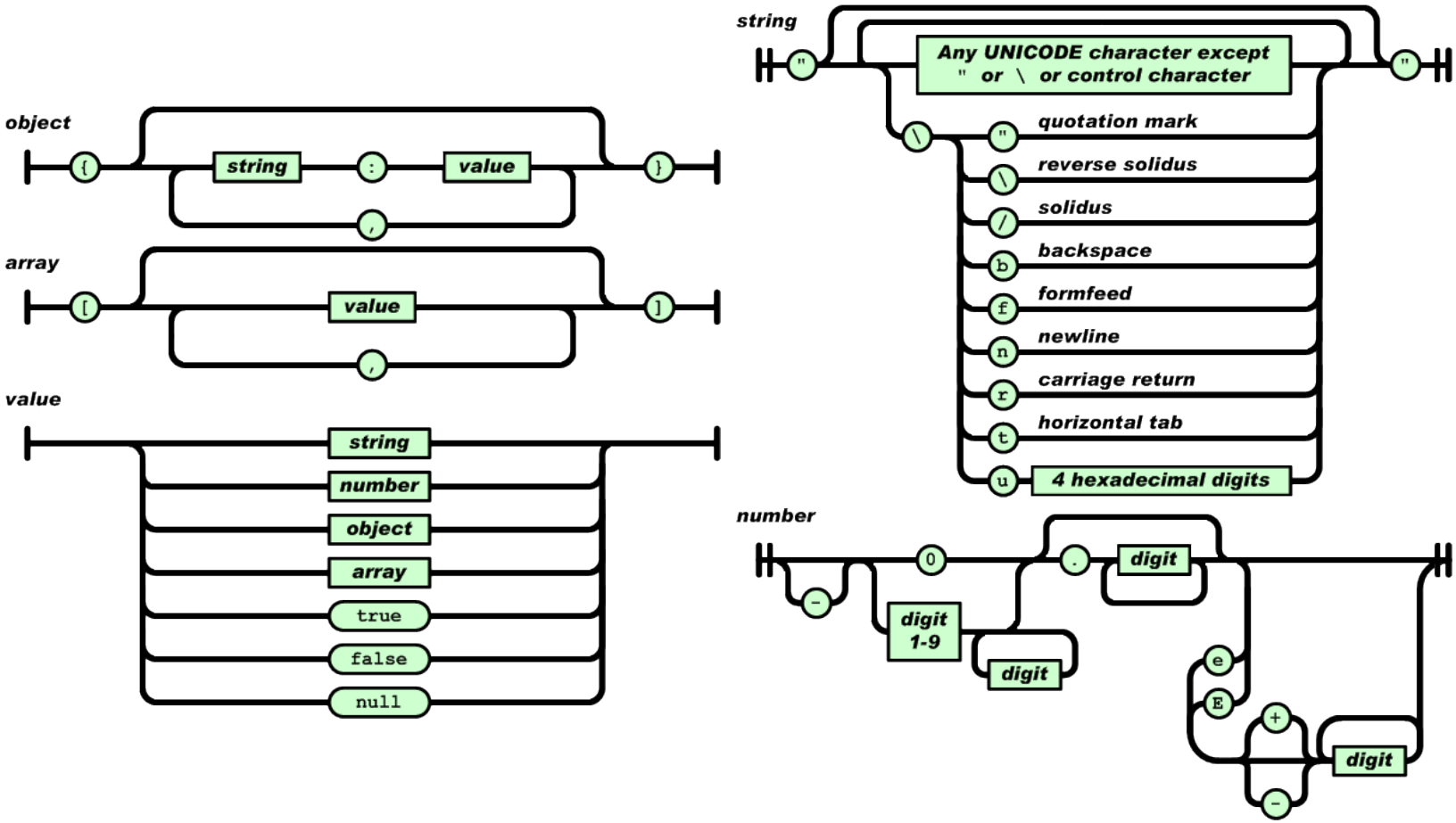
JSON

JSON (JavaScript Object Notation) ist ein leichtgewichtiges Datenaustauschformat. Es ist einfach für Menschen zu lesen und gleichzeitig einfach für Maschinen zu parsen und generieren.

JSON baut auf zwei Arten von Strukturen auf:

- Eine Sammlung von Name-Wert-Paaren (ähnlich einer Map); JSON object
- Eine geordnete Liste von Werten; JSON array

JSON Syntax


object
{
string
:
value
}
,
array
[
value
]
,
value
string
number
object
array
true
false
null
string
"
Any UNICODE character except
" or \ or control character
"
\
quotation mark
reverse solidus
solidus
backspace
formfeed
newline
carriage return
horizontal tab
4 hexadecimal digits
b
f
n
r
t
u
/
number
-
0
digit
1-9
digit
.
digit
e
E
+
-
digit

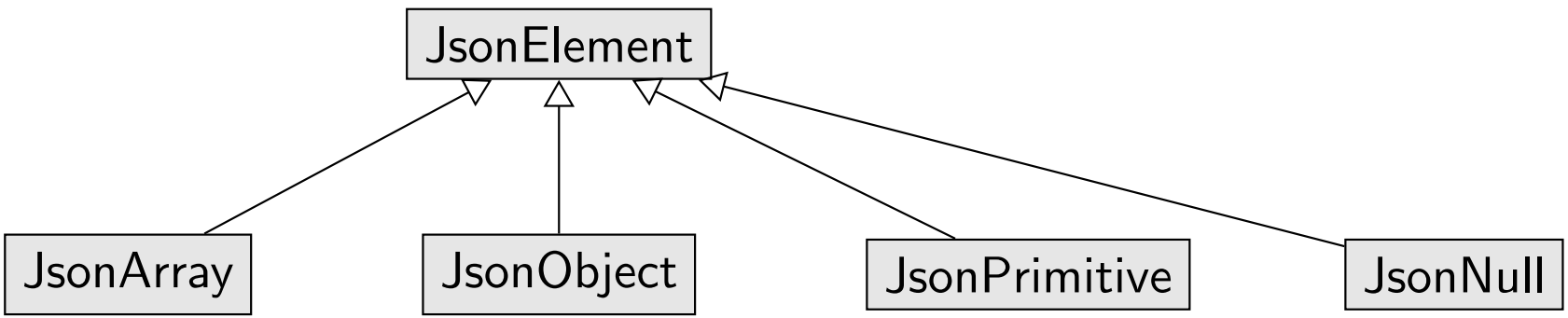
Gson

Gson ist ein Java-Bibliothek zur Erzeugung von JSON Repräsentationen von Objekten. Die Bibliothek enthält Unterstützung für

- die direkte Beschreibung der zu erzeugenden JSON Ausgabe,
- die Umwandlung von Java-Objekten in JSON, sowie
- entsprechende Methoden zum Parsen von JSON.

Direkte Beschreibung von JSON I

Für jedes Element in der JSON-Syntax gibt es eine Java-Klasse, die dieses Element repräsentiert.



Mit Objekten dieser Klassen können JSON-Ausgaben beschrieben werden.

Direkte Beschreibung von JSON II

```
Gson gson = new Gson();
JsonObject university =
  new JsonObject();
university.addProperty("name",
                       "TU KL");
JsonArray courses =
  new JsonArray();
courses.add("SE 1");
courses.add("SE 2");
courses.add("SE 3");
courses.add("Insy");
university.add("courses",
               courses);

gson.toJson(university);
```

```
{"name":"TU KL",
 "courses":
   ["SE 1",
    "SE 2",
    "SE 3",
    "Insy"]
}
```

Direkte Beschreibung von JSON III

```
Gson gson = new Gson();
JsonObject parsedUni = gson.fromJson(jsonString,
    JsonObject.class);
assertEquals(parsedUni.get("name").getAsString(),
             "TU KL");
JsonArray parsedCourses =
    parsedUni.get("courses").getAsJsonArray();
assertEquals(parsedCourses.get(0).getAsString(), "SE 1");
assertEquals(parsedCourses.get(1).getAsString(), "SE 2");
assertEquals(parsedCourses.get(2).getAsString(), "SE 3");
assertEquals(parsedCourses.get(3).getAsString(), "Insy");
```

Direkte Beschreibung von JSON IV

- Code stimmt mit ausgegebenem JSON überein
- Struktur direkt beeinflussbar
- Volle Kontrolle, somit auch bei
  - genauen Vorgaben des Ausgabeformats
  - Zyklen im Objektgeflecht

Umwandlung von Java-Objekten in JSON I

Für viele Java-Objekte ist die Relation zwischen dem Objekt im Speicher und der Ausgabe als JSON klar:

- Objekte werden als JSON-Objekt,
- Attribute von Objekten werden als Eigenschaften (properties) des JSON Objects,
- Listen und Arrays werden als JSON arrays und
- Maps werden ebenfalls als JSON object ausgegeben.

Reichen diese Konventionen aus, kann Gson die Konvertierung in der Regel automatisch durchführen.

Umwandlung von Java-Objekten in JSON II

```
public class University {
    private String name;
    private List<String> courses;

    public University(String name) {
        this.name = name;
        this.courses = new ArrayList<>();
    }

    public void addCourse(String course) {
        this.courses.add(course);
    }
    public boolean equals(Object o) {...}
}
```

Umwandlung von Java-Objekten in JSON III

```
Gson gson = new Gson();
University unikl = new University("TU KL");
unikl.addCourse("SE 1");
unikl.addCourse("SE 2");
unikl.addCourse("SE 3");
unikl.addCourse("Insy");
String json = gson.toJson(unikl);

University parsedUni =
    gson.fromJson(json, University.class);
assertEquals(parsedUni, unikl);
```

Weitergabe an Writer und Reader

JSON kann direkt auf einen `Writer` geschrieben oder von einem `Reader` gelesen werden:

```
// schreibe das University Objekt unikl in Datei unikl.
    json
gson.toJson(unikl, new JsonWriter(
                       new FileWriter("unikl.json")))

// lies das University Objekt aus der Datei unikl.json
gson.fromJson(new FileReader("unikl.json"),
              University.class)
```

Subtyping und Gson

```
abstract class Course {
    protected String name;
    protected int etcs;
    // ...
}

class Seminar extends Course {
    private int numPlaces;
    // ...
}

class Lecture extends Course {
    private boolean writtenExam;
    // ...
}

public class University2 {
    private String name;
    private List<Course> courses;
}
```

```
University2 unikl = new University2("TU KL");
unikl.addCourse(new Lecture("SE 1", 10, true));
unikl.addCourse(new Seminar("SE-Seminar", 4, 15));
Gson gson = new Gson();
String json = gson.toJson(unikl);
// Ergibt Json:
{
  "name": "TU KL",
  "courses": [
    {
      "writtenExam": true,
      "name": "SE 1",
      "etcs": 10
    },
    {
      "numPlaces": 15,
      "name": "SE-Seminar",
      "etcs": 4
    }
  ]
}
```

Serialisierung anpassen

```
public class CourseSerialization
      implements JsonSerializer<Course>, JsonDeserializer<
    Course> {
    @Override
    public JsonElement serialize(Course course, Type typeOfSrc,
          JsonSerializationContext context) {
        ...
    }

    @Override
    public Course deserialize(JsonElement json, Type typeOfT,
          JsonDeserializationContext context) throws
    JsonParseException {
        ...
    }

}
```

Serialisierung anpassen

```
@Override
public JsonElement serialize(Course course, Type typeOfSrc,
      JsonSerializationContext context) {
    String type;
    if (course instanceof Seminar) {
        type = "seminar";
    } else if (course instanceof Lecture) {
        type = "lecture";
    } else {
        throw new RuntimeException("Unknown type: " + course);
    }
    JsonObject obj = (JsonObject) context.serialize(course);
    obj.add("type", new JsonPrimitive(type));
    return obj;
}
```

Serialisierung anpassen

```
@Override
public Course deserialize(JsonElement json, Type typeOfT,
      JsonDeserializationContext context) throws
    JsonParseException {
    JsonObject obj = json.getAsJsonObject();
    String type = obj.get("type").getAsString();
    switch (type) {
        case "seminar":
            return context.deserialize(json, Seminar.class);
        case "lecture":
            return context.deserialize(json, Lecture.class);
        default:
            throw new JsonParseException("Unknown type: " +
    type);
    }
}
```

```
GsonBuilder builder = new GsonBuilder();
builder.registerTypeAdapter(Course.class, new
    CourseSerialization());
Gson gson = builder.create();
String json = gson.toJson(unikl);
// Ergibt Json:
{
  "name": "TU KL",
  "courses": [
    {
      "writtenExam": true,
      "name": "SE 1",
      "etcs": 10,
      "type": "lecture"
    },
    {
      "numPlaces": 15,
      "name": "SE-Seminar",
      "etcs": 4,
      "type": "seminar"
    }
  ]
}
```

Zusammenfassung

Checked und Unchecked Exceptions

Eingabe und Ausgabe von Dateien

Datenströme verwenden und implementieren

Ein- und Ausgabe von Objekten (Beispiel: Json)