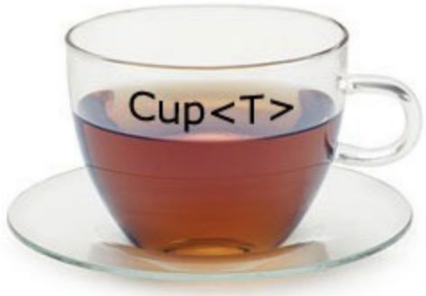
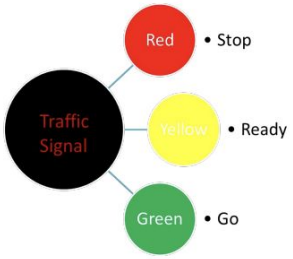
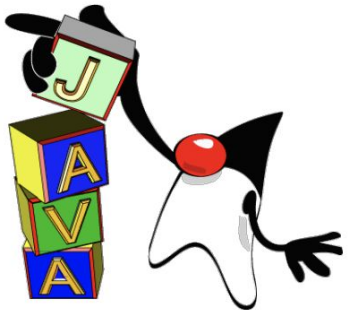
Einführung in die Programmiersprache Java IV







Inhalt

- Innere klassen
- Ausnahmebehandlung
- JSON

Innere Klassen

- Java 1+ :-)
- eine Innere Klasse (Inner/Nested Class) wird innerhalb des Codeblocks einer anderen Klasse implementiert
- die bisher diskutierten Klassen werden auch Top Level genannt
- Vorteile
  - Elegante
  - Nützlich

Innere Klassen

- Hilfsklassen
  - klassisches Beispiel: Klasse **List** und Klasse **Node** oder **Iterator**
- erlauben die Definition möglichst nahe an der Stelle, wo sie gebraucht werden

```
public class TopLevelClass1 {

}
```

TopLevelClass.java

```
public class OuterClass {

      ...

      public class InsideClass {

      }

}
```

OuterClass.java

The Zoo of inner classes

- die Theorie ist einfach
  - scheinbar!

- Inner Classes (Member-/Element-Klassen)
  - innere Klassen, die in anderen Klassen definiert wurden
- Nested Classes (Verschachtelte Top Level Klassen)
  - sind statische Klassen, die innerhalb anderer Klassen definiert sind

- Lokale Klassen
  - Klassen, die innerhalb einer Methode oder eines Blocks definiert werden
- Anonyme Klassen
  - Lokale und namenlose Klassen

Nested Classes


```
class EnclosingClass{

    ...

    static class StaticNestedClass {

        ...

    }

    class InnerClass {

        ...

    }

} // end of enclosing class
```

Nested Classes

```
public class A {
    static int i = 4711;
    public static class B {
        int my_i = i;
        public static class C { ... }//end of class C
    }// end of class B
}// end of class C
//in Main-Methode
 A a = new A();
 A.B ab = new A.B();
 A.B.C abc = new A.B.C();
```

Nested Classes

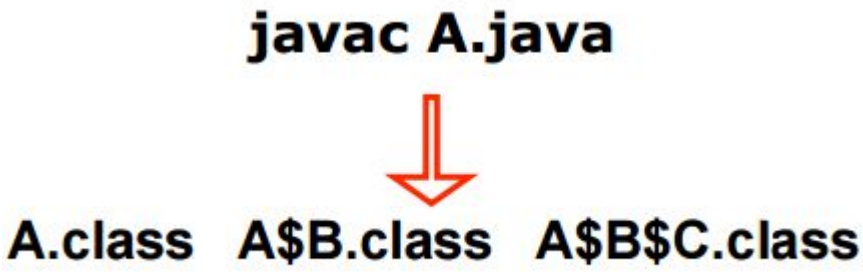
```
public class A {
    static String a = "A";
    String b = "B";
    public static class B {
        void m() {
            System.out.println(a);
        }
    } // end of class B
} // end of class A
```

Inner Classes

- Member-Klassen sind echte innere Klassen im Gegensatz zu den geschachtelten Klassen
  - die nur zur Strukturierung dienen
- solche Klassen haben Zugriff auf alle Attribute und Methoden der umgebenden Klasse
  - erhalten einen impliziten zusätzlichen Member, nämlich eine Referenz auf das umschließende Objekt
- solche Klassen werden analog benutzt genau wie *normale* Klassen
- ein Objekt der inneren Klasse ist mit dem Objekt der umgebenden Klasse gebunden

Inner Classes

```
public class A {

    ...

    public class B {

        ...

        public class C {

            ...

        }

    }

}
```



Inner Classes

Objekte von Member-Klassen sind immer mit einem Objekt der umgebenden Klasse verbunden

```
public class A {

    public static int i = 30;

    public class B {

        int j = 4;

        public class C {

            int k = i;

        }

    }

}
```

```
A a = new A();
A.B b = a.new B();
A.B.C c = b.new C();
```

Inner Classes

- Jeder Instanz einer Elementklasse ist ein Objekt der umgebenden Klassen zugeordnet
- Damit kann das Objekt der Elementklasse implizit auf die Instanzvariablen der umgebenden Klasse zugreifen
  - `this` Zeiger
- Elementklassen dürfen keine statischen Elemente

Inner Classes

```
public class H {
    static String t = "text";
    String at = "another text";
    public class B {
        public void print() {
            System.out.println(at);
            System.out.println(t);
        }
    }// end of class B
}// end of class H
```

Lokale Klassen

- lokale Klassen sind innere Klassen, die nur lokal innerhalb von Blöcken bzw. Methoden.

```
public class C {
    ...
    public void doSomething() {
        int i = 0;
        class X implements Runnable {
            public X() {...}
            public void run() {...}
        }
        new X().run();
    } // end of doSomething
}
```

Lokale Klassen

- Lokale Klassen dürfen nicht als public, protected, private oder static deklariert werden
- sind nur innerhalb des Blocks sichtbar, in dem sie definiert werden
- Lokale Klassen dürfen keine statischen Elemente haben
- Eine Lokale Klasse kann im umgebenden Codeblock nur die mit final markierten Variablen und Parameter benutzen
  - enthalten implizit eigene Kopien aller lokalen finalen Variablen

Lokale Klassen

```
public class T {
   String attr = “text”;
   public void f() {
      final String final_var = “in method”;
      int var = 10;
      class U {
         void u() {
            System.out.println(attr);
            System.out.println(final_var);
            System.out.println(var);
         }
      }
      U u = new U();
      u.u();
   }
}
```

```
class TestLocKlassen {
   public static void main(...) {
      T t = new T(); t.f();
   }
}
```

Anonyme Klassen

- haben keinen Namen
- haben keinen Konstruktor
- sie entstehen immer zusammen mit einem Objekt
- werden wie lokale Klassen innerhalb von Anweisungsblöcken definiert
- `new-expression class-body`

Anonyme Klassen

```
abstract class Person {
   abstract void eat();
}

class TestAnonKlassen {
   public static void main(String[] arr) {
      Person p = new Person() {
         void eat () {
            System.out.println(“eating fruit…”); }
      }
   }
}
```

Fehlerhafte Programme

- Ein Programm kann aus vielen Gründen unerwünschtes Verhalten zeigen
- Fehler beim Entwurf
- Fehler bei der Programmierung des Entwurfs
  - Algorithmen falsch implementiert
- Ungenügender Umgang mit außergewöhnlichen Situationen
  - Abbruch der Netzwerkverbindung
  - Dateien können nicht gefunden werden
  - fehlerhafte Benutzereingaben

Umgang mit außergewöhnlichen Situationen

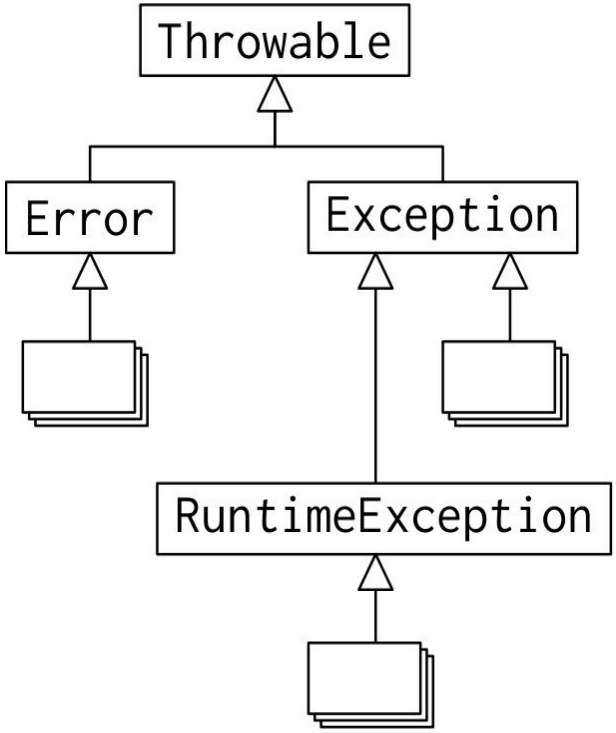
- Ausnahmesituationen unterscheiden sich von Programmierfehlern darin,
  - dass man sie nicht (zumindest prinzipiell) von vornherein ausschließen kann.

- Immer möglich sind zum Beispiel:
  - unerwartete oder ungültige Eingaben
  - Ein- und Ausgabe-Fehler beim Zugriff auf Dateien oder Netzwerk

Ausnahmen in Java

In Java werden verschiedene Arten von Ausnahmen durch verschiedene Unterklassen von `Throwable` repräsentiert.

- Instanzen von Error
- Instanzen von Exception
- Instanzen von RuntimeException



Error, Exception, RuntimeException

- `RuntimeException` Ausnahmen signalisieren ein Problem im Code
  - Wenn sie nicht behandelt werden, stürzt das Programm mit einer Fehlermeldung ab
  - unzulässige Umwandlungsoperationen
    - `ClassCastException`
  - oder unangemessene Verwendung eines Null-Pointer
    - `NullPointerException`
  - oder Zugriff auf ein Array-Element außerhalb der Grenzen
    - `IndexOutOfBoundsException`
- `Error` Ausnahmen signalisieren kritische Probleme, die normalerweise von der Anwendung nicht behandelt werden können
  - Speicherfehler
    - `OutOfMemorryError`
  - Stackoverflow
  - Ausfall der Java-VM
    - `VirtualMachineError`

Checked Exceptions

- `Exception` Ausnahmen werden als geprüfte Ausnahmen **(Checked Exceptions)** bezeichnet
  - der Compiler bestätigt zur Kompilierungszeit, dass die Methode Code enthält, der möglicherweise eine Ausnahme auslöst
  - der Compiler benötigt den Code, der eine solche Methode aufruft, um diesen Aufruf in einen try-Block aufzunehmen und einen geeigneten catch-Block bereitzustellen, um die Ausnahme abzufangen

- Geprüfte Ausnahmen repräsentieren Ausnahmesituationen, mit denen das Programm rechnen kann und auf die es reagieren sollte.
  - FileNotFoundException, IOException...
- Geprüfte Ausnahmen müssen entweder behandelt werden
  - oder als möglich deklariert werden.

Unchecked Exceptions

- runtime generierte Ausnahmen werden als ungeprüfte **Ausnahmen (Unchecked Exceptions)** bezeichnet, da der Compiler nicht feststellen kann, ob der Code die Ausnahme behandelt

- Ungeprüfte Ausnahmen repräsentieren Ausnahmesituationen, deren Ursache ein Programmierproblem ist

- Alle Ausnahmen, die von `RuntimeException` abgeleitet sind, sind ungeprüfte Ausnahmen
  - NullPointerException, IllegalArgumentException...

- Ungeprüfte Ausnahmen müssen weder behandelt noch deklariert werden.

Auslösen von Ausnahmen

- das Auslösen einer Ausnahme erfolgt mit der Anweisung

  ```
  throw exp;
  ```

  - wobei exp Ausdruck vom Typ Throwable ist.
- für Methoden kann man möglicherweise auftretende Ausnahmen deklarieren

```
public void m() throws IOException {
  if (...) {
    throw new IOException();
  }
}

// Annotationen sagen nur, dass eine Ausnahme moeglicher-
// weise aufritt. Tatsaechlich kann sie auch nie auftreten.
public void n() throws IOException {
  System.out.println();
}
```

Behandlung von Ausnahmen

```
try {
  // Block fuer "normalen" Code
} catch (Exception1 e) {
  // Ausnahmebehandlung fuer Instanzen von Exception1
} catch (Exception2 e) {
  // Ausnahmebehandlung fuer Instanzen von Exception2
} finally {
  // Code, der in jedem Fall nach normalem
  // Ausnahmebehandlung ausgefuehrt werden
}
```

Behandlung von Ausnahmen

```
try {
   ...
} catch (IOException e) {
   ...
} catch (JSONException e) {
   ...
}
```

besser als catch everything

```
try {
   ...
} catch (Exception e) { }
```

Behandlung von Ausnahmen

```
try {
  Iterator<String> i = list.iterator();
  while (true) {
    String s = i.next();
    ...
  }
} catch (NoSuchElementException e) { }
```

```
for (String s: list) {
  ...
}
```

Hinweise

- ein Programm sollte nie mit einer Exception abbrechen
- Geprüfte Exceptions sind an einer geeigneten Stelle mit try abzufangen und zu behandeln
- Ausnahmesituationen müssen sinnvoll behandelt werden
  - falsche Benutzereingabe -> neue Eingabeaufforderung
  - IO-Fehler -> nochmal versuchen
- nicht sinnvoll behandelbarer Fehler
  - Benutzerdaten speichern, Programm beenden

Hinweise

- Ungeprüfte Ausnahmen, die Programmierfehler repräsentieren, werden nicht abgefangen.
  - NullPointerException
  - IllegalArgumentException
  - ClassCastException

- Die einzige sinnvolle Reaktion auf solche Exceptions ist das Programm zu korrigieren.
  - kein Abfangen solcher Exceptions mit try

Konvention

öffentliche (public) Methoden überprüfen eventuelle Annahmen an ihre Parameter und lösen gegebenenfalls eine Exception aus

```java
/**
 * Konstruiere eine neue Node mit den gegebenen Daten
 *
 * @param id eindeutiger Name der Node, nicht null
 * @param latitude Koordinate
 * @param longitude Koordinate
 */
public Node(String id, double latitude, double longitude) {
  if (id == null) {
    throw new NullPointerException();
  }
  this.id = id;
  this.longitude = longitude;
  this.latitude = latitude;
}
```

Dokumentation

- ungeprüfte Exceptions werden üblicherweise nicht mit throws deklariert
- die möglichen ungeprüften Exceptions sollten jedoch im Javadoc dokumentiert werden

```
/**
 * Returns the element at the specified position in this list.
 *
 * @param index index of the element to return
 * @return the element at the specified position in this list
 * @throws IndexOutOfBoundsException {@inheritDoc}
 */
```

Fehlerklassen

```
1   class TestException extends Exception {
2    public TestException(String s) {super(s);}
3   }
4
5   class Test {
6     public void hello() throws TestException {
7       throw new TestException("Test::hello() Exception");
8     }
9   }
10
11  class Main {
12    public static void main(String[] args) {
13      try {
14        Test t = new Test();
15        t.hello();
16      }
17      catch (TestException t) {
18        System.out.println("Exception raised:" + t.getMessage());
19      }
20    }
21  }
```

```
javac -classpath .:/run_dir/junit-4.12.jar:target/dependency/* -d . Main.java
java -classpath .:/run_dir/junit-4.12.jar:target/dependency/* Main
Exception raised:Test::hello() Exception
```

Ausnahmen und Vererbung

- Unterklassen dürfen keine zusätzlichen Exception werfen
- Unterklassen können Unterklassen von den deklarierten Ausnahmen der Basisklasse werfen

```
class E1 extends Exception {}

class E2 extends Exception {}

class E3 extends E2 {}

class A { void m() throws E2 {} }

class B extends A { void m() throws E3 oder E2 {} }
```

Beispiel

Wir wollen unsere Stack Klasse mit einer benutzerdefinierten Fehlerklasse StackFullException, die wird ausgelöst, wenn der Stack voll ist und man die push() methode aufruft.

```java
public class FullStackException extends RuntimeException{ //extends Ex
    public FullStackException(String message) {
        super(message);
    }
}
```

```java
import java.util.EmptyStackException;

public class BasicStack<T> implements Stack<T>{
    private int max_size;
    private T[] data;
    private int top;

    public BasicStack(int max_size) {
        this.max_size = max_size;
        this.top = -1;
        this.data = (T[]) new Object[max_size];
    }

    @Override
    public void push(T element) {
        if (top < max_size - 1)
            data[++top] = element;
        else
            throw new FullStackException();
    }

    @Override
    public T pop() {
        if (! isEmpty())
            return data[top--];
        throw new EmptyStackException();
    }

    @Override
    public T peek() {
        if (! isEmpty())
            return data[top];
        throw new EmptyStackException();
    }

    @Override
    public boolean isEmpty() {
        return this.top == -1;
    }
}
```

```java
public interface Stack<T> {
    void push (T element);
    T pop();
    T peek();

    boolean isEmpty();
}
```

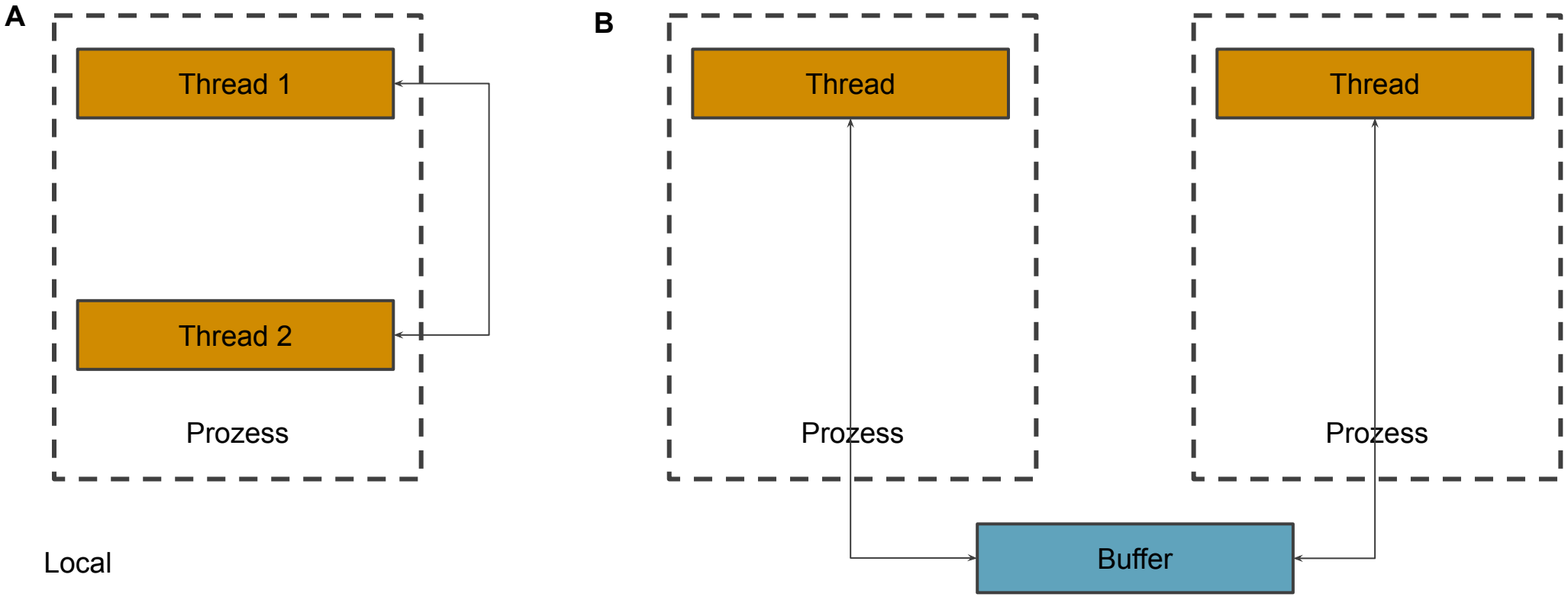
Hinweise

- Ungeprüfte Ausnahmen, die Programmierfehler repräsentieren, nicht abfangen
- Argumente in öffentlichen Methoden überprüfen
- Ausnahmen möglichst spezifisch behandeln
  - Ausnahmen nicht ignorieren
- Ausnahmen nur in außergewöhnlichen Situationen verwenden
- Ausnahmen dokumentieren

Datenaustausch

- Kodierung von Daten
- Binärformate (PNG, MP4, Word, . . .)
  - effizient, aufwändig, nicht menschenlesbar
- Textformate (Java, . . .):
  - menschenlesbar, Aufwand für Ein- und Ausgabe
- generische Formate (XML, JSON, . . .):
  - Datenaustausch, implementiert in Bibliotheken

Datenaustausch



Datenaustausch

- JSON (JavaScript Object Notation)
- einfaches textbasiertes Datenaustauschformat
- menschenlesbar
- Standardisiert in RFC 4627

JSON

- “Objekte” mit Attribut:Wert-Zuordnungen
- Leerzeichen außerhalb von Strings, Zeilenumbrüche nicht relevant

```
{
  "type":"node",
  "id":"363179",
   "lat":48.1408871, "long":11.5615991
},
{
  "type":"way", "id":"372802991",
  "nd":  ["3763512880","3763512881","1545920068"],
  "tags": {
       "bus":"yes",
       "name":"Herkomerplatz",
       "highway":"platform"
  }
}
```

Datentypen

- Standard-Datentypen
  - Strings in Anführungszeichen, Escaping mit \ (wie in Java)
  - Zahlen (z.B. -12, 12E9, 12.9)
  - Boolesche Werte (true, false)
  - Null-Wert durch Schlüsselwort null
- Arrays
  - in eckigen Klammern (z.B. [1,2,3,4])
- Objekte
  - in geschweiften Klammern
  - Attribute durch Strings benannt

Verarbeitung von JSON-Daten

- Es gibt viel Bibliotheken zur Ein- und Ausgabe von JSON
- org.json
  - für Java auf http://json.org verfügbar
- Jackson
- GSON

Parsing

```
String s = { "type":"way",
"nd": ["3763512880","3763512881","1545920068"],
"tags": { "name":"Herkomerplatz", "highway": "platform" }
}

JSONObject json = new JSONObject (s);

String t = json.getString ("type"); // " way "

JSONArray nd = json.getJSONArray ("nd");
double d1 = nd.GetDouble (1); //3763512881
```

Parsing

```
String s = { "type":"way", "nd":
["3763512880","3763512881","1545920068"],
"tags": { "name":"Herkomerplatz", "highway": "platform" }
}
JSONObject json = new JSONObject (s);

JSONObject tags = json.getJSONObject ("tags");
String n = tag.getStrings ("name ");

for (String k: json.keySet ())
   System.out.println (k);
```

Ausgabe

```
JSONObject json = new JSONObject ();
json.put ("type" , "node");
json.put ("id", "34");
json.put ("lat" , 31.3);
json.put ("long" , 12.8);
System.out.println (json);
```

Ausgabe:

```
{"id":"34","type":"node","lat":31.3,"long":12.8}
```

JSON

- JSON dient nur zum Datenaustausch
- z.B. JSONObject nicht zur Datenrepräsentation
- Beim Austausch von Text ist auf die Textkodierung zu achten
- Behandeln Sie bei der Programmierung alle möglichen Fehlerfälle. JSON-Daten, die aus einer Datei gelesen werden, können nicht als wohlgeformt angenommen werden

Beispiel

JSON Parsing mit Jackson