Datenstrukturen mit XML/JSON, SQL und UML

Modelleriung von Daten und Datenstrukturen

XML

JSON

SQL

UML

Modelleriung von Daten und Datenstrukturen

Modellierung von Daten

Damit ein Computer versteht, wie er mit Daten umgehen kann, müssen die Strukturen von Daten festgelegt werden.

Dabei wird unter anderem angegeben,

- welche Typen die einzelnen Daten haben
- wie sie miteinander in Beziehung stehen
- welche Operationen mit ihnen erlaubt sind

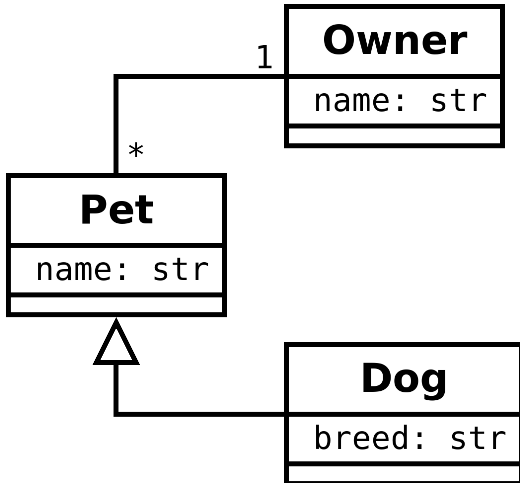
Die eigentlichen Daten werden dann als Instanzen dieser Strukturen modelliert.

Datenstruktur: Klasse

```python
class Pet:
    def __init__(self, name: str):
        self.name: str = name

class Dog(Pet):
    def __init__(self, name: str,
            breed: str):
        super().__init__(name)
        self.breed: str = breed

class Owner:
    def __init__(self, name: str,
            pets: List[Pet]):
        self.name: str = name
        self.pets: List[Pet] = pets
```

Klassen: Graphische Modellierung als Uml-Klassendiagramm

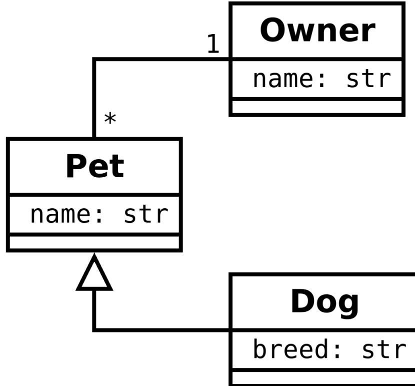
```
class Pet:
    def __init__(self, name: str):
        self.name: str = name

class Dog(Pet):
    def __init__(self, name: str,
            breed: str):
        super().__init__(name)
        self.breed: str = breed

class Owner:
    def __init__(self, name: str,
            pets: List[Pet]):
        self.name: str = name
        self.pets: List[Pet] = pets
```

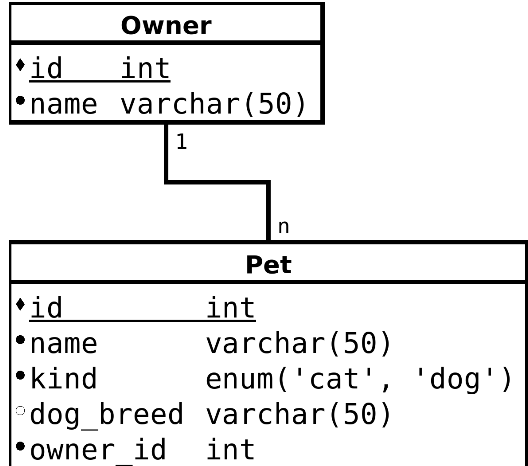


Klassen: Übertragung Xml Grammatik (DTD)

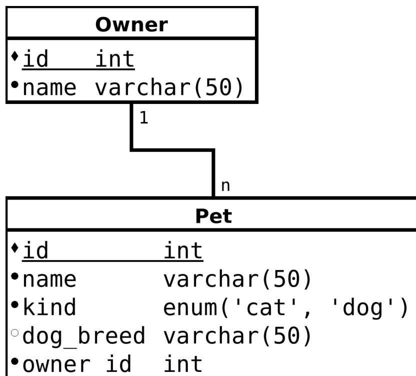
```python
class Pet:
    def __init__(self, name: str):
        self.name: str = name

class Dog(Pet):
    def __init__(self, name: str,
            breed: str):
        super().__init__(name)
        self.breed: str = breed

class Owner:
    def __init__(self, name: str,
            pets: List[Pet]):
        self.name: str = name
        self.pets: List[Pet] = pets
```

```
<!ELEMENT owner (name, pet*)>
<!ELEMENT pet (name, (dog | cat))>
<!ELEMENT dog (breed)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT breed (#PCDATA)>
<!ELEMENT cat EMPTY>
```


Owner
name: str
1
*
Pet
name: str
Dog
breed: str


Owner
id int
name varchar(50)
1
n
Pet
id int
name varchar(50)
kind enum('cat', 'dog')
dog_breed varchar(50)
owner_id int

Klassen: CREATE-Statements für Datenbankschema



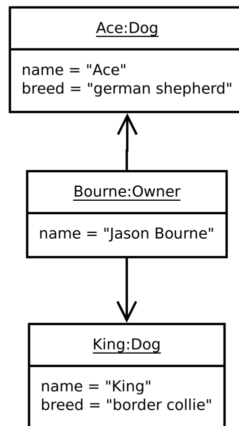

```sql
create table owner (
  id   int primary key,
  name varchar(50)
);

create table pets (
  id        int primary key,
  name      varchar(50),
  kind      enum ('cat', 'dog'),
  dog_breed varchar(50),
  owner_id  int,
  foreign key (owner_id)
  references owner (id)
);
```

Instanzen von Klassen

```
dog1 = Dog('Ace',
           'german shepherd')

dog2 = Dog('King',
           'border collie')

owner = Owner('Jason Bourne',
              [dog1, dog2])
```

Instanzen: Modellierung als Objektdiagramm

```
dog1 = Dog('Ace',
           'german shepherd')

dog2 = Dog('King',
           'border collie')

owner = Owner('Jason Bourne',
              [dog1, dog2])
```



Instanzen: Modellierung als Xml

```python
dog1 = Dog('Ace',
           'german shepherd')

dog2 = Dog('King',
           'border collie')

owner = Owner('Jason Bourne',
              [dog1, dog2])
```

```xml
<owner>
    <name>Jason Bourne</name>
    <pet>
        <name>Ace</name>
        <dog>
            <breed>german shepherd</breed>
        </dog>
    </pet>
    <pet>
        <name>King</name>
        <dog>
            <breed>border collie</breed>
        </dog>
    </pet>
</owner>
```

Instanten: SQL-Insert-Statements

```python
dog1 = Dog('Ace',
           'german shepherd')

dog2 = Dog('King',
           'border collie')

owner = Owner('Jason Bourne',
              [dog1, dog2])
```

```sql
insert into owner
values (1, 'Jason Bourne');

insert into pets
values (1, 'Ace', 'dog', 'german shepherd', 1),
       (2, 'King', 'dog', 'border collie', 1);
```

XML

Xml Beispiel (1): Nachricht

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE note SYSTEM "note.dtd">
<note>
    <to id="max123">Max</to>
    <from>Moritz</from>
    <heading>Hallo, Moritz</heading>
    <!-- Kommentar -->
    <body>Schön, dich zu sehen!</body>
</note>
```

Xml Beispiel (2): Autohersteller

```xml
<?xml version="1.0"?>
<?xml-stylesheet type="text/css"
        href="style.css"?>
<manufacturerlist>
    <manufacturer>
        <name>Audi</name>
        <founded>1909</founded>
        <models>
            <model>A4</model>
            <model>A6</model>
        </models>
    </manufacturer>
    <manufacturer>
        <name>BMW</name>
        <founded>1916</founded>
        <models>
            <model>3er</model>
            <model>5er</model>
        </models>
    </manufacturer>
</manufacturerlist>
```

Xml: eXtensible Markup Language

...Auszeichnungssprache zur Darstellung hierarchisch strukturierter Dokumente in Form von Textdateien.

- Single Source Prinzip: *write once, read everytime*
- Automatische Extraktion von Inhalten mit Parser
- Trennung von
  - Inhalt
  - Struktur
  - Layout

Anwendungen von Xml

- Xml selbst hat keine Wirkung, sondern *beschreibt* nur Daten („Auszeichnung“)
- Zur Interpretation / Verarbeitung durch einen Computer wird ein sogenannter Parser benötigt
- Plattform- und implementations- **unabhängiger** Austausch von Daten, Nutzung unter anderem im Internet
  - Aktienkurse
  - Wetterdaten
  - ...

Syntaxregeln von Xml

- Xml-Dokumente bestehen aus hierarchisch verschachtelten Elementen
- Jedes Dokument hat **genau ein** Wurzelelement, das das Elternelement aller anderen Elemente ist (hier: `<note>`)
- Kommentare werden in `<!----` und `---->` eingefasst und vom Parser ignoriert

```xml
<note>
    <to id="max123">Max</to>
    <from>Moritz</from>
    <heading>Hallo, Moritz</heading>
    <!-- Kommentar -->
    <body>Schön, dich zu sehen!</body>
</note>
```

Xml: Prolog

- Optional kann am Anfang des Dokuments ein sogenannter Prolog angegeben werden. Dieser beinhaltet:
  - Die Xml-Version
  - Codierung (UTF-8 oder ISO 8859–1, empfohlen: UTF-8)
  - Beispiel: `<?xml version="1.0" encoding=" UTF-8" ?>`
- Außerdem kann eine Grammatik referenziert werden (DTD oder XSD, siehe später)
  - direkt nach Prolog
  - Beispiel: `<!DOCTYPE note SYSTEM "Note.dtd" >`

Elemente

- Elemente werden innerhalb von Tags notiert
- Ein Tag wird immer von „<“ gestartet und mit „>“ geschlossen
- Elemente müssen geöffnet und geschlossen werden
  - Öffnen: `<note>` (öffnender Tag, Starttag)
  - Schließen: `</note>` (schließender Tag, Endtag, Abschlusstag)
  - Sonderfall: leere Elemente (Starttag = Endtag), Beispiel: `<example/>`
- Name von Start- und Endtag muss übereinstimmen, kann aber sonst frei gewählt werden
- **Achtung**: Xml ist „Case Sensitive“, das heißt name $\neq$ Name

Attribute

- Attribute gehören zu Elementen und beinhalten Daten, die zu einem Objekt gehören
- Werden innerhalb des Starttags angegeben
- Notation:

```
<tag_name name="wert" >
```

- Verwendung von Attributen vs Verwendung von Kindelementen
  - Keine genaue Regel, aber Richtlinien
  - Wenn ein kompliziertes Objekt modelliert werden soll (Datum: Tag, Monat, Jahr), dann besser ein Element
  - Zu viele Attribute pro Element verschlechtern Leserlichkeit
  - Ansonsten meist gleiche Aussage

Verschachtelung von Elementen

- Durch Verschachtelung von Elementen kann eine Ist-Teil-von-Beziehung (Hierarchie) modelliert werden
- Einteilung in Eltern- und Kindelemente
- Hervorhebung durch Einrückung der Kindelemente (kein Teil der Syntax wie bei Python, nur für Menschen)
- Beispiel:

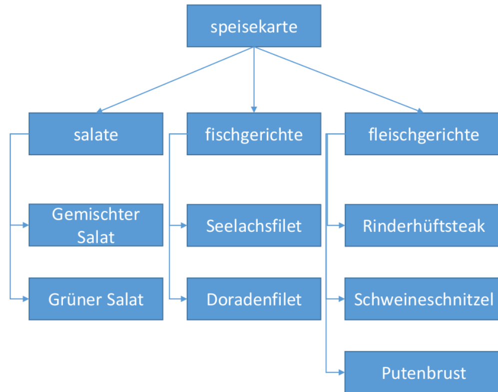
Eine Notiz besteht aus

- Adressat
- Absender
- Titel
- Rumpf

```xml
<note>
    <to id="max123">Max</to>
    <from>Moritz</from>
    <heading>Hallo, Moritz</heading>
    <!-- Kommentar -->
    <body>Schön, dich zu sehen!</body>
</note>
```

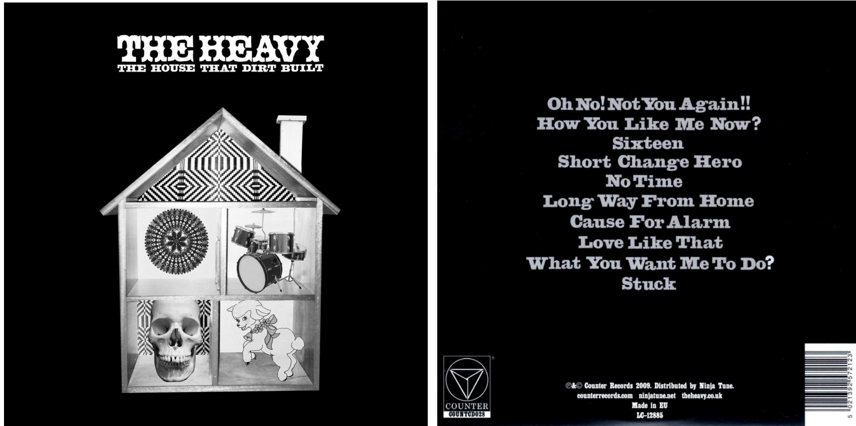
Visualisierung der Hierarchie

```xml
<?xml version="1.0" encoding="UTF-8"?>
<speisekarte>
    <salate>
        <salat>Gemischter Salat</salat>
        <salat>Grüner Salat</salat>
    </salate>
    <fischgerichte>
        <fisch>Seelachsfilet</fisch>
        <fisch>Doradenfilet</fisch>
    </fischgerichte>
    <fleischgerichte>
        <fleisch>Rinderhüftsteak</fleisch>
        <fleisch>Schweineschnitzel</fleisc
        <fleisch>Putenbrust</fleisch>
    </fleischgerichte>
</speisekarte>
```



Aufgabe 1: Erstellung eines Xml-Dokuments

Erstellen Sie für folgende CD ein Xml-Dokument:



Lösung (Beispiel)

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE cd SYSTEM "cd.dtd">
<cd songs="10">
    <artist>The Heavy</artist>
    <title>The House That Dirt Built</title>
    <songlist>
        <song>Oh No! Not You Again!</song>
        <song>How You Like Me Now?</song>
        ...
        <song>Stuck</song>
    </songlist>
</cd>
```

Xml Parser

- Xml-Parser existieren für viele Programmiersprachen, meistens sogar in der Standardbibliothek
- Beispiele
  - JAXP in Java
  - ElementTree in Python
  - DOMParser in Javascript
- Brauchen ein korrektes Dokument (sonst Fehler!)
- Anwendung eventuell später in Programmierteil der Vorlesung!
- Funktionen:
  - Einlesen des Dokuments
  - Zugriff auf Elemente („Iteration“ über Elemente)
  - Eventuell Manipulation des Dokuments

Validierung von Xml-Dokumenten

Wann ist ein Xml-Dokument korrekt? Man unterscheidet:

- Wohlgeformtheit: Dokument hält die Syntaxregeln von Xml ein
  $\Rightarrow$ Dokument kann von einem Parser ohne Fehler gelesen werden

- Gültigkeit: Dokument hält das durch die (referenzierte) Grammatik beschriebene Format ein (und ist wohlgeformt)
  $\Rightarrow$ Parser kann die Reihenfolge und den Inhalt der Element überprüfen
  $\Rightarrow$ Andere Anwendungen wissen, „wovon das Dokument handelt“

Wohlgeformtheit eines Dokuments

Wohlgeformtheit bedeutet, dass das Dokument unter anderem folgende Syntaxregeln einhält:

- Es hat genau ein Wurzelelement
- Alle Elemente haben ein schließendes Tag
- Tags sind korrekt „case sensitive“ geschrieben (name $\neq$ naMe)
- Elemente sind richtig verschachtelt
- Attributwerte sind in Anführungszeichen

Ein Test der Wohlgeformtheit ist online möglich:

`http://www.xmlvalidation.com/`

Beispiel: Verletzungen der Wohlgeformtheit

```
<!-- Kein Wurzelelement -->
<artist>The Heavy</artist>
<title>The House that dirt built</title>

<!-- Tag falsch geschrieben -->
<artist>The Heavy</arTist>

<!-- Element nicht geschlossen -->
<songlist>
    <song>How You Like Me Now?
</songlist>
```

Aufgabe: Wohlgeformtheit

Überarbeiten Sie Ihr Dokument aus der vorherigen Aufgabe:

- Versuchen Sie, mögliche Verletzungen der Wohlgeformtheit zu finden und auszubessern
- Testen Sie Ihr Dokument **danach** auf Wohlgeformtheit mit dem Tool von W3 und beheben Sie Ihre Fehler!
- `http://www.xmlvalidation.com/`

Xml Grammatiken

- Voraussetzung für Gültigkeit ist die Wohlgeformtheit des Dokuments
- Außerdem muss das Dokument der angegebenen Grammatik entsprechen

- Es gibt zwei Typen von Grammatiken:
  - DTD – Document Type Definition
  - Xml Schema
- Diese bestimmen
  - Reihenfolge, Verschachtelung und Typen von Elementen
  - Typ, Anzahl und Vorkommen von Attributen
- Definieren damit einen Standard für den Austausch von Daten

Beispiel: DTD

```
<!ELEMENT note (to, from, heading,
        (message | body))>
        <!ELEMENT to (#PCDATA)>
        <!ATTLIST to id CDATA #REQUIRED>
        <!ELEMENT from (#PCDATA)>
        <!ELEMENT heading (#PCDATA)>
        <!ELEMENT body ANY>
        <!ELEMENT message (#PCDATA)>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE note SYSTEM "note.dtd">
<note>
    <to id="max123">Max</to>
    <from>Moritz</from>
    <heading>Hallo, Moritz</heading>
    <!-- Kommentar -->
    <body>Schön, dich zu sehen!</body>
</note>
```

DTD: Definition von Elementen

`<!ELEMENT name (wert)>`

- *name* bezeichnet den Namen des definierten Elements
- *wert* definiert den möglichen Inhalt des Elements
- Mit einer Elementgruppierung (siehe Tabelle, letzte Zeile) kann eine kompliziertere Elementstruktur definiert werden

Elementwert	
#PCDATA	Element beinhaltet (beliebigen) Text
ANY	Beliebiger Inhalt
EMPTY	Leeres Element
(*name*, ...)	Andere Elemente

DTD: Definition von Eltern-Kind-Relation

Beispiele:

```
<!ELEMENT note (to, from, heading, (message | body))>
<!ELEMENT person (car*, favsong+)>
```

Elementgruppierungen im Elementwert		
(A | B | ...)	Alternative	A oder B oder...
(A, B, ...)	Sequenz	Erst A, dann B, dann ...
A*	Wiederholung 0 ...n	Beliebig oft A
A+	Wiederholung 1 ...n	Beliebig oft A, aber mindestens 1 mal
A?	Option 0 ...1	A oder nicht A / eventuell A

DTD: Definition von Attributen

Syntax: `<!ATTLIST` *element_name (name, typ, bedeutung)+*`>`

```
<!ATTLIST to name CDATA #REQUIRED>
<!ATTLIST student
    residence CDATA "Würzburg"
    matrikel CDATA #REQUIRED
    wach (ja | nein) #IMPLIED
>
```

Typ	
CDATA	(beliebiger) Text
(*enum1*, *enum2*, ...)	Aufzählung
ID(REF)	Attribut ist (Verweis auf) ID
...	...

Bedeutung	
#IMPLIED	Optionales Attribut
#REQUIRED	Nichtoptionales Attribut
value	Standardwert für Attribute
FIXED *value*	Fester Wert für Attribut

Aufgabe: Erstellung einer DTD

Erstellen Sie für folgendes Dokument eine DTD:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE cd SYSTEM "cd.dtd">
<cd songs="10">
    <artist>The Heavy</artist>
    <title>The House That Dirt Built</title>
    <songlist>
        <song>Oh No! Not You Again!</song>
        <song>How You Like Me Now?</song>
        ...
        <song>Stuck</song>
    </songlist>
</cd>
```

Lösung: Erstellung einer DTD

```
<!ELEMENT cd (artist, title, songlist)>
        <!ATTLIST cd songs CDATA #REQUIRED>
        <!ELEMENT artist (#PCDATA)>
        <!ELEMENT title (#PCDATA)>
        <!ELEMENT songlist (song+)>
        <!ELEMENT song (#PCDATA)>
```

Xml Schema: Beispiel

```xml
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xs="http://wwww.w3.org/2001/XMLSchema">
    <xs:element name="note">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="to" type="xs:string">
                    <xs:attribute name="id" type="xs:string" use="required"/>
                </xs:element>
                <xs:element name="from" type="xs:string"/>
                <xs:element name="heading" type="xs:string"/>
                <xs:element name="body" type="xs:string"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

Xml Schema versus DTD

- Vorteile Xml Schema:
  - Xml Schema ist wieder reines Xml, kann daher von jedem Parser geparst werden
  - Xml Schema erlaubt eine genauere Definition der Datentypen als eine DTD

- Vorteile DTD:
  - Xml Schema produziert mehr „Overhead“ als eine DTD
  - DTD ist leichter zu lesen und zu verstehen

Zusammenfassung Xml: Dokumente

- Xml ist eine Auszeichnungssprache für Daten
- Hat keine eigene Wirkung, sondern beschreibt nur Daten
- Durch Auszeichnung mit Tags wird die maschinelle Verarbeitung („Parsen“) vereinfacht
- Verarbeitung nur möglich, wenn Dokument wohlgeformt, ansonsten gibt es Fehler beim Parsen
- Trennung von
  - Inhalt (Daten)
  - Struktur (Tags, Grammatik)
  - Layout (z. B. mit CSS)

Zusammenfassung Xml: Grammatiken

- Grammatiken beschreiben die Struktur (Hierarchie) eines Dokuments
- Zwei Typen von Grammatiken: Xml Schema und Document Type Definition
  - DTD hat weniger Ausdruckskraft (Definition von Wertebereichen für Elemente und Attribute!), ist dafür einfacher zu verstehen
  - Xml Schema ist wieder Xml, hat aber mehr Overhead als eine DTD
- Grammatiken können nicht beschreiben, welchen Inhalt das Dokument hat, nur wie der Inhalt strukturiert ist!

JSON

JSON

Nachteile XML:

- Viel „Boilerplate“-Code (schließendes Tag)
- Keine optimale Grammatik
  - Eine DTD ist selbst kein XML
  - XML Schema ist zwar XML, aber kompliziert

Alternative: **JavaScript Object Notation (JSON)**

- Datenformat in einfach lesbarer Textform zum Zweck des Datenaustauschs zwischen Anwendungen
- Ist gültiges Javascript und kann daher (in JS) per `eval()` interpretiert werden ($\Rightarrow$ Parsing)
- Verwendung zum Beispiel bei sogenannten AJAX-Requests, um dynamisch Inhalte nachzuladen

Beispiel: JSON versus XML

```json
{
  "to": {
    "id": "max123",
    "name": "Max"
  },
  "from": "Moritz",
  "heading": "Hallo, Moritz",
  "body": "Schön, dich zu sehen!"
}
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE note SYSTEM "note.dtd">
<note>
    <to id="max123">Max</to>
    <from>Moritz</from>
    <heading>Hallo, Moritz</heading>
    <!-- Kommentar -->
    <body>Schön, dich zu sehen!</body>
</note>
```

Datentypen in JSON

- Null (`null`)
- Boolesche Werte (`boolean`)
- Zahlen (`number`)
- Zeichenketten (`string`)
- Arrays (`array`)
- Objekte (`object`)

Anmerkung: Vergleiche Python...

```json
{
  "null_value": null,
  "boolean": true,
  "number": {
    "integer": 1,
    "float": 3.14159
  },
  "string": "my_string",
  "array": [
    "this", "is", "an",
    "array", "of", "strings"
  ]
}
```

Beispiel: Kreditkarte

```json
{
  "herausgeber": "Xema",
  "nummer": "1234-5678-9012-3456",
  "deckung": 2e+6,
  "waehrung": "EURO",
  "deaktiviert": false,
  "inhaber": {
    "name": "Mustermann",
    "vorname": "Max",
    "geschlecht": "maennlich",
    "hobbies": [
      "Reiten",
      "Golfen"
    ],
    "alter": 42,
    "kinder": [
      {
        "name": "Mustermann",
        "vorname": "Sandra",
        "alter": 17
      },
      {
        "name": "Mustermann",
        "vorname": "Michael",
        "alter": 15
      }
    ],
    "partner": null
  }
}
```

JSON Schema

Mit JSON Schema kann - ähnlich zu XML - eine Grammatik für ein JSON-Dokument definiert werden.

Analog zu XML-Grammatiken ...

- ist JSON Schema auch wieder ein JSON-Dokument
- definiert JSON Schema
  - die hierarchischen Beziehungen zwischen den Werten ($\Rightarrow$ Struktur des Dokuments)
  - die Datentypen der einzelnen Werte
  - **nicht** die Semantik des Dokuments

JSON Schema ist noch kein fertiger Standard, sondern befindet sich noch in der Entwurfsphase.

Beispiel: JSON Schema

```
{
  "type": "object",
  "properties": {
    "to": {
      "type": "object",
      "properties": {
        "id": {"type": "string"},
        "name": {"type": "string"}
      }, "required": ["id", "name"]
    },
    "from": {"type": "string"},
    "heading": {"type": "string"},
    "body": {"type": "string"}
  }, "required": ["to", "from", "heading", "body"]
}
```

```
{
  "to": {
    "id": "max123",
    "name": "Max"
  },
  "from": "Moritz",
  "heading": "Hallo, Moritz",
  "body": "Schön, dich zu sehen!"
}
```

Kurzübersicht JSON Schema

- Eine JSON Schema Definition ist immer ein Objekt
- Der Wert `type` gibt des Typ (siehe Folie 42, Datentypen JSON) des Wurzelobjekts an.
- Bei einem Objekt gibt es zwei zusätzliche Felder:
  - `properties`: Gibt die Attribute des Objekts an (als Abbildung String auf JSON Schema Definition)
  - `required`: Gibt an, welche Attribute (als Array von Strings) angegeben werden müssen (alle anderen sind optional)
- Bei einem Array gibt es ein zusätzliches Feld:
  - `items`: Gibt des Typen der beinhalteten Werte an (als JSON Schema Definition)
- Mit dem Operator `$ref` kann eine Definition zu besseren Lesbarkeit ausgelagert werden

Beispiel JSON Schema, Kreditkarte (1)

```
{
  "type": "object",
  "required": ["herausgeber", "nummer", "deckung", "waehrung", "inhaber", "deaktiviert"],
  "properties": {
    "herausgeber": {"type": "string"}, "nummer": {"type": "string"},
    "deckung": {"type": "number"}, "deaktiviert": {"type": "boolean"},
    "waehrung": {"type": "string", "enum": ["DOLLAR", "EURO", "RUBEL"]},
    "inhaber": {"$ref": "#/definitions/Inhaber"}
  },
  "definitions": {
    "Person": {
      "type": "object", "required": ["name", "vorname", "alter"], "properties": {
        "name": {"type": "string"}, "vorname": {"type": "string"},
        "geschlecht": {"type": "string", "enum": ["maennlich", "weiblich", "divers"]},
        "alter": {"type": "number"}
      }
    },
```

Beispiel JSON Schema, Kreditkarte (3)

```
        "Inhaber": {
          "type": "object", "required": [
            "name", "vorname", "geschlecht", "hobbies", "alter", "kinder", "partner"
          ],
          "properties": {
            "name": {"type": "string"}, "vorname": {"type": "string"},
            "geschlecht": {"type": "string", "enum": ["maennlich", "weiblich", "divers"]},
            "alter": {"type": "number"},
            "hobbies": {"type": "array", "items": {"type": "string"}},
            "kinder": {"type": "array", "items": {"$ref": "#/definitions/Person"}},
            "partner": {
              "anyOf": [{"type": "null"}, {"$ref": "#/definitions/Person"}]
            }
          }
        }
      }
    }
```

JSON und Python

In Python existiert das Paket `json` zum Arbeiten mit JSON:

```python
import json

json_str: str = """{
    "key": "value",
    "array_key": ["string", "array"]
}"""

json_dict = json.loads(json_str)
print(type(json_dict))  # <class 'dict'>
print(json_dict['key'])  # value

json_dict['object_key'] = {
    'other_value': False
}
```

```python
json_str_new: str = json.dumps(
    json_dict, indent=2)
print(json_str_new)
# {
#   "key": "value",
#   "array_key": [
#     "string",
#     "array"
#   ],
#   "object_key": {
#     "other_value": false
#   }
# }
```

SQL

SQL

- Datenbanksprache zur Definition von Datenstrukturen und Manipulation von Daten in relationalen Datenbanken
- Verschiedene „Dialekte“ unter anderem MySQL, Microsoft SQL Server, Oracle SQL, SQLite, PostgreSQL, ...
- Häufig Nutzung der (kostenlosen) Open Source-Version von MySQL, Installation zum Beispiel mit XAMPP (Nutzung über phpmyadmin)
- Unterstützung für `CRUD`-Operationen
  - CREATE (Datensatz anlegen)
  - READ (Datensatz lesen)
  - UPDATE (Datensatz aktualisieren)
  - DELETE (Datensatz löschen)

Datenbanken (vereinfacht)

- Eine Datenbank ist eine Menge von Tabellen, die wiederum die Daten speichern
- Bereitstellung mehrerer Datenbanken auf einem SQL-Server möglich
- Graphische Modellierung der Tabellen und der Beziehungen zwischen den Tabellen mit Hilfe eines Datenbankdiagramms
- Definition und Veränderung der Datenbanken, Tabellen und Inhalte mittels einer (Programmier-)Sprache ($\Rightarrow$ SQL)

Tabellen

- Tabellen (*Relationen*) sind Mengen von Zeilen (*Tupeln*)
- Tupel sind wiederum Mengen von Werten
- Ein Datensatz entspricht einer Zeile

id	*first_name*	*family_name*	*birthday*
1	Joanne K.	Rowling	1965-07-31
2	George	Orwell	1903-06-25
3	John Ronald Reuel	Tolkien	1892-01-03
4	Antoine	de Saint-Exupery	1900-06-29
6	Robert	Ludlum	1927-05-25

Tabellen: Datentypen und Schlüssel

Jedem Wert in einem Tupel wird ein entsprechender Datentyp zugeordnet (z. B. `INT`, `VARCHAR(`*`laenge`*`)`, `DATETIME`, `TEXT`, `FLOAT`, …)

Normalerweise wird für jede Relation ein Wert als **Primärschlüssel** definiert, der ein Tupel *eindeutig* identifiziert

- Nutzung von „natürlichen“ Schlüssen, zum Beispiel Nutzername, ISBN bei Büchern, Matrikelnummer, …
- Falls kein natürlicher Schlüssel vorliegt, kann ein „künstlicher“ Schlüssel definiert werden (meist eine Zahl, siehe Beispiel)

Der Primarschlüssel kann bei Referenzierungen zwischen Tabellen verwendet werden

Aufteilung von Tabellen

Das Design von Tabellen sollte so gewählt werden, dass keine Information doppelt abgespeichert werden muss.

Beispiel:

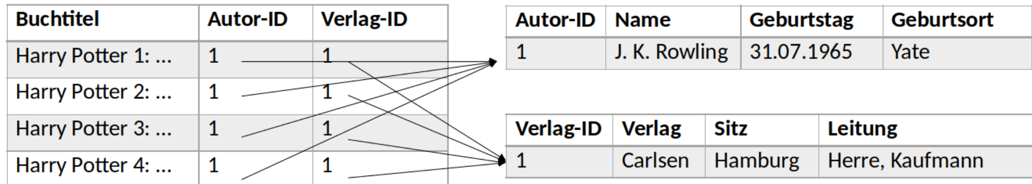
- Speicherung einer Liste von Büchern mit Preis, Autor, ...in einer Tabelle
- Wo speichert man Zusatzinformationen zum Autor, wie zum Beispiel Geburtsdatum, -ort, ...oder zum Verlag, wie Sitz, Leitung, ...?

Buchtitel	Autor	Geburtstag	Geburtsort	Verlag	Verlagssitz
Harry Potter 1: ...	J. K. Rowling	1965-07-31	Yate	Carlsen	Hamburg
Harry Potter 2: ...	J. K. Rowling	1965-07-31	Yate	Carlsen	Hamburg
Harry Potter 3: ...	J. K. Rowling	1965-07-31	Yate	Carlsen	Hamburg

Aufteilung von Tabellen

Lösung:

- Aufteilung der Informationen auf verschiedene Tabellen mit Referenzierungen zwischen den Tabellen
- Referenzierung des Primärschlüssels einer anderen Tabelle, um eine Verknüpfung zu erzeugen
- Durchführung mit Hilfe eines Algorithmus möglich



Buchtitel	Autor-ID	Verlag-ID
Harry Potter 1: ...	1	1
Harry Potter 2: ...	1	1
Harry Potter 3: ...	1	1
Harry Potter 4: ...	1	1

Autor-ID	Name	Geburtstag	Geburtsort
1	J. K. Rowling	31.07.1965	Yate

Verlag-ID	Verlag	Sitz	Leitung
1	Carlsen	Hamburg	Herre, Kaufmann

Datenbankdiagramm

- Beschreibt die Tabellen und die Inhalte der Tabellen grafisch
- Ähnelt Klassendiagrammen der UML, ist daher auch mit UML-Klassendiagramm-Notation durchführbar
- Darstellung einer Tabelle als Rechteck
  - Oberer Teil des Rechtecks: Name der Tabelle
  - Unterer Teil: Einzelne Spalten mit Spaltenname und -datentyp
- Fremdschlüsselbeziehung werden durch Verbindungen zwischen Tabellen dargestellt

Beispiel: Datenbankdiagramm

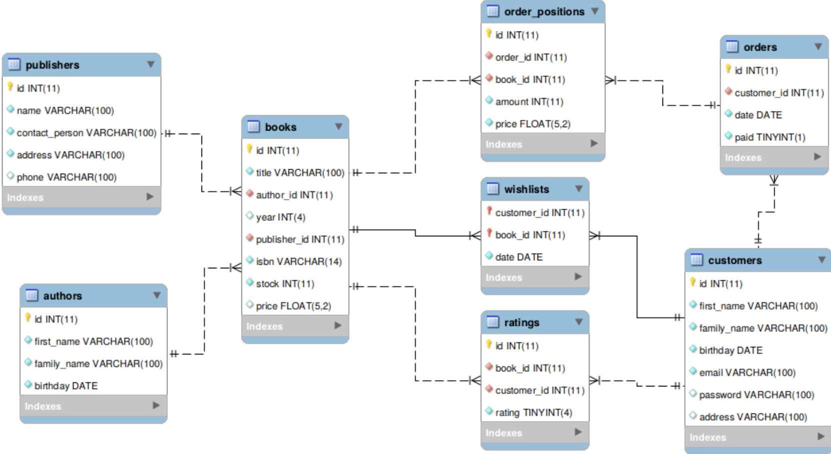


Abbildung 2: Beispiel Datenbankdiagramm

Datenbankzugriffe in SQL

- Erzeugen und Löschen von Datenbanken und Tabellen mit `CREATE` und `DROP`
- Verändern der Tabellen mit `INSERT`, `UPDATE` und `DELETE`
- Eigentlicher Datenzugriff mit `SELECT`, auch über mehrere Tabellen

- Zugriff aus Java oder anderen Programmiersprachen mit einer Schnittstellt (z. B. Java Database Connectivity, JDBC)
- Kommentare werden in SQL mit `----` eingeleitet

Beispiele für Anfragen

- Abfragen von Werten
  - Wie lautet der Name des Autors mit der ID 5?
  - Wie viele Bücher schrieb der Angestellte mit der ID 1?
  - Welcher ist der älteste Autor?
  - Was kosten alle Bücher von Orwell zusammen?
- Änderung der Datenbank
  - Ändere den Preis des Buches ...auf 9.99
  - Füge den neuen Autor ...in die Tabelle ein
  - Lösche das nicht mehr verfügbare Buch ...

SELECT-Statements

Mit Hilfe des `SELECT`-Statements können Anfragen auf eine Datenbank gemacht werden.

Dabei können folgende unter anderem Dinge spezifiziert werden:

- Tabelle(n), auf die die Anfrage gemacht wird
- Spalte(n), die ausgegeben werden soll(en)
- Bedingung(en), die die ausgewählten Zeilen erfüllen müssen
- Reihenfolge oder Gruppierungen der ausgewählten Zeilen

Minimale Anfrage (wählt alle Inhalte der Tabelle „*tabelle*“):

```
SELECT * FROM tabelle
```

SELECT: Syntax

```
SELECT spalten
        FROM tabellen
        WHERE bedingungen
        ORDER BY reihenfolge
        GROUP BY gruppierungen;
```

- Angabe der auszugebenden Spalten
- Angabe der verwendeten Tabellen
- **Optionale** Angabe der Bedingungen
- **Optionale** Angabe der Reihenfolge
- **Optionale** Angabe einer Gruppierung

SELECT: Grundlegende Syntax

Nach dem Schlüsselwort `SELECT` folgt die Angabe der auszugebenden Spalten (kommasepariert), das Schlüsselwort `FROM` und die Tabelle

Beispiele:

```sql
-- Alle Spalten einer Tabelle (hier: "books") aus
SELECT *                FROM books;

-- Gib bestimmte Spalte(n) einer Tabelle aus
SELECT family_name      FROM authors;
SELECT title, isbn      FROM books;
```

Aufgabe: Grundlegende SELECTs

Bearbeiten Sie folgende Aufgaben in `it4all`. Orientieren Sie Sich dabei am Datenbankdiagramm!

- Szenario: `Amazon`
- Typ: `SELECT`
- IDs: 1 bis 5

Bedingungen in SELECTs

Oft wollen wir nicht alle Datensätze einer Tabelle, sondern nur einen Teil. Dafür können wir Bedingungen angeben, die die Tupel erfüllen müssen, die wir auswählen

- Angabe nach Schlüsselwort `WHERE`
- Verknüpfung mehrerer Bedingungen mit `AND`, `OR` und `NOT` (siehe Allgemeiner Teil, Boolesche Algebra)
- Verschiedene Bedingungen, die wichtigsten sind:
  - Gleichheit: `spalte = wert` (`stock = 75848` oder `title = 1984`)
  - Ähnlichkeit: `spalte like wert` (`title like 'Harry Potter'`)

SELECT: Grundlegende Syntax WHERE

```sql
-- Alle Bücher, die in 2001 veröffentlich wurden
SELECT * FROM books
  WHERE year = 2001;

-- Das Buch mit dem Namen '1984'
SELECT * FROM books
  WHERE title = '1984';

-- Alle Bücher mit 'Harry Potter' im Namen
SELECT * FROM books
  WHERE title LIKE 'Harry Potter';
```

Aufgabe: SELECTs mit einer Bedingung

Bearbeiten Sie folgende Aufgaben in `it4all`. Orientieren Sie Sich dabei am Datenbankdiagramm!

- Szenario: `Amazon`
- Typ: `SELECT`
- IDs: 6 bis 10

Weitere Vergleichsoperatoren für Bedingungen

Operator	Beschreibung	Beispiel
=	Gleichheit	… `WHERE id = 5`
<>	Ungleichheit	… `WHERE id <> 5`
>	Größer	… `WHERE height > 120`
>=	Größer gleich	… `WHERE height >= 120`
<	Kleiner	… `WHERE height < 120`
<=	Kleiner gleich	… `WHERE height <= 120`
BETWEEN(x, y)	Zwischen zwei Werten	… `WHERE height BETWEEN(120, 210)`
LIKE	Ähnlichkeit	… `WHERE email LIKE '%@t-online.de'`
IN	Angabe verschiedener Werte	… `WHERE rating in (3, 4, 5)`

Erweiterte Bedingungen

Wie bereits erwähnt können Bedingungen mit AND, OR und NOT verknüpft werden.

Beispiele:

- Bücher-ID der Bewertungen des Nutzers mit ID 12 und 5 Sternen
- ID aller Bücher, die (in 2016) älter als 5 Jahre sind und weniger als 70.000 mal vorrätig sind
- IDs aller Bewertungen, die 4 oder 5 Sterne haben
- Nachnamen aller Autoren, die nicht „Rowling“ heißen

Die Klammerung kann die Verknüpfung der Bedingungen beeinflussen.

Beispiele: Erweiterte Bedingungen

```
SELECT book_id FROM ratings
    WHERE customer_id = 12 AND rating = 5;

SELECT id FROM books
    WHERE (2016 - year) > 5 AND stock < 70000;

SELECT id FROM ratings
    WHERE rating = 4 OR rating = 5;

SELECT family_name FROM authors
    WHERE NOT family_name = 'Rowling';
```

Aufgabe: SELECTs mit mehreren Bedingung

Bearbeiten Sie folgende Aufgaben in `it4all`. Orientieren Sie Sich dabei am Datenbankdiagramm!

- Szenario: `Amazon`
- Typ: `SELECT`
- IDs: 16 bis 20

SELECT: JOINs

- Wie finden wir Informationen, die in zwei Tabellen gespeichert ist?
- Beispiel: Alle Bücher, die die Autorin „J. K. Rowling " geschrieben hat?
- Naive Antwort: Zwei Abfragen
  1. Ermittle Autoren-ID von „J. K. Rowling " (Ergebnis: 1)
  2. Nutze ID in Tabelle `books`, um entsprechende Bücher auszugeben
- Neue Antwort: `JOIN` zweier Tabellen in einer Abfrage
  - In einer Abfrage wird die Tabelle books verwendet, um die Titel auszugeben
  - Mit einem JOIN wird diese Tabelle mit einer zweiten verbunden, um die Tupel über eine Bedingung zu korrelieren
  - Korrelation zum Beispiel über (Fremd-)Schlüssel

Beispiel: JOINs

```
-- Ordne den Büchern ihre Autoren zu
SELECT * FROM books
    JOIN authors ON books.author_id = authors.id;

-- Ordne den Büchern ihre Verlage zu
SELECT * FROM books
    JOIN publishers ON books.publisher_id = publishers.id;
```

JOIN-Bedingung

- Die Bedingung eines JOIN gibt an, welche Tupel der beiden „gejointen“ Tabellen zusammengefügt werden
- Wird keine Bedingung angegeben, werden einfach alle Zeilen miteinander verbunden
- Normalerweise werden die Spalten als JOIN-Bedingung angegeben, die sowieso die Abhängigkeit der beiden Tabellen modellieren (zum Beispiel die Spalte ID in der Tabelle „Autoren“ und die Autor-ID in der Tabelle „Buch“)

- Technisch gesehen werden alle Spalten aussortiert, in die die beiden Werte nicht übereinstimmen

JOINs und Bedingungen

Nach einem JOIN können wie bei einer normalen Query Bedingungen angegeben werden. Es ist hierbei darauf zu achten, dass gleichnamige Spalten (zum Beispiel `id`) mit dem Tabellennamen als Präfix identifiziert werden:

```
-- Alle Bücher, die von einem Autoren mit Vornamen Antoine geschrieben wurden
SELECT * FROM books
       JOIN authors ON books.author_id = authors.id
       WHERE first_name = 'Antoine';

-- ID in Bedingung ist nicht eindeutig, daher Angabe der Tabelle
SELECT * FROM books
       JOIN authors ON books.author_id = authors.id
       WHERE authors.id = 4;
```

Aufgabe: SELECTs mit einem JOIN

Bearbeiten Sie folgende Aufgaben in `it4all`. Orientieren Sie Sich dabei am Datenbankdiagramm!

- Szenario: `Amazon`
- Typ: `SELECT`
- IDs: 11 bis 15

DISTINCT

Um doppelte Werte in einer Spalte auszusortieren, kann das Schlüsselwort `DISTINCT` benutzt werden.

Beispiele:

```
-- Alle verschiedenen Preise
SELECT DISTINCT price FROM books;

-- Alle Kunden-IDs, für die bereits eine Bestellung existiert
SELECT DISTINCT customer_id FROM orders;
```

Aggregatsfunktionen

- Use-Case: Berechnung von Aggregationsfunktionen über Spalten
- Beispiele:
  - Durchschnitt der abgegebenen Bewertungen eines Nutzers
  - Durchschnitt der Bewertungen für ein Buch
  - Schlechteste/Beste Bewertung eines Buches
  - Anzahl der abgegebenen Bewertungen
  - Anzahl der Kunden
  - ...
- Auch in einer Programmiersprache möglich, aber einfacher in SQL

Beispiele: Aggregatsfunktionen

```sql
-- Anzahl aller Bewertungen
SELECT COUNT(rating) FROM ratings;

-- Durschnitt Bewertungen des Nutzers mit ID 5
SELECT AVG(rating) FROM ratings WHERE customer_id = 5;

-- Geburtstag des jüngsten Autoren
SELECT MAX(birthday) FROM authors;

-- Summe aller Bestände der Harry Potter Bücher
SELECT SUM(stock) FROM books
    WHERE title like 'Harry Potter%';
```

Aggregatsfunktionen

Funktion	Wirkung	Anmerkung
`AVG(`*spalte*`)`	Durchschnitt der Spalte	Numerischer Datentyp in der Spalte
`SUM(`*spalte*`)`	Summe aller Werte	Numerischer Datentyp in der Spalte
`COUNT(`*spalte*`)`	Anzahl der Werte in einer Spalte	Verschiedene Werte mit `DISTINCT`
`MAX(`*spalte*`)`	Maximalert Wert	Vergleichsrelation benötigt
`MIN(`*spalte*`)`	Minimaler Wert	Vergleichsrelation benötigt

Aliasse für Spalten

- Wenn eine Aggregatsfunktion benutzt wird, wird diese als Spaltenname im Ergebnis benutzt
- Meist möchte man den Spaltennamen aber passender wählen, wie zum Beispiel
  - „Anzahl_Bewertungen" statt `COUNT(rating)` oder
  - „Minimaler_Bestand" statt `MIN(stock)`
- Daher kann man beliebigen Spalten im Ergebnis ein sogenanntes *Alias* geben

- Syntax: `SELECT` ***`spalte`*** `AS` ***`alias`***`, …FROM …`

Beispiel: Aliasse für Spalten

```sql
-- Anzahl aller Bewertungen
SELECT COUNT(rating) AS Anzahl_Bewertungen
    FROM ratings;

-- Durschnitt Bewertungen des Nutzers mit ID 5
SELECT AVG(rating) AS Durchschn_Bewertungen
    FROM ratings WHERE customer_id = 5;

-- Spätester Geburtstag eines Autoren
SELECT MAX(birthday) AS Letzter_Geburtstag
    FROM authors;
```

Aufgabe: SELECTs mit mehreren Bedingung

Bearbeiten Sie folgende Aufgaben in `it4all`. Orientieren Sie Sich dabei am Datenbankdiagramm!

- Szenario: `Amazon`
- Typ: `SELECT`
- IDs: 21 bis 25

Mehrfache JOINs

- Bisher haben wir immer 2 Tabellen miteinander „gejoint“
- Es ist aber auch möglich, über mehrere Tabellen hinweg zu „joinen“

- Beispiele:
  - Ordne den Kunden die Titel der Bücher zu, die sie sich wünschen (Customers <-> Wishlists <-> Books)
  - Ordne den Kunden die Titel der Bücher zu, die sie bestellt haben (Customers <-> Orders <-> Order_positions <-> Books)
  - Ordne den Kunden die Verlage zu, von denen sie Bücher bestellt haben (Customers <-> Orders <-> Order_positions <-> Books <-> Publishers)

Beispiel (1): Mehrfache JOINs

```sql
-- Kunden <-> Bücher, die sie sich wünschen
SELECT first_name, family_name, title FROM customers
       JOIN wishlists ON customers.id = wishlists.customer_id
       JOIN books ON books.id = wishlists.book_id;

-- Kunden <-> Bücher, die sie gekauft haben
SELECT first_name, family_name, title FROM customers
       JOIN orders ON customers.id = orders.customer_id
       JOIN order_positions ON orders.id = order_positions.order_id
       JOIN books ON books.id = order_positions.book_id;
```

Beispiel (2): Mehrfache JOINs

```sql
-- Kunden <-> Verlage, deren Bücher sie gekauft haben
SELECT first_name, family_name, name FROM customers
       JOIN orders ON customers.id = orders.customer_id
       JOIN order_positions ON orders.id = order_positions.order_id
       JOIN books ON books.id = order_positions.book_id
       JOIN publishers ON books.publisher_id = publishers.id;
```

Sortierung mit ORDER BY

- Use-Case: Sortierung der Ergebnisse aufsteigend (`ASC`, ascending) oder absteigend (`DESC`, descending)
- Beispiele:
  - Sortierung der Bücher nach Bestand
  - Sortierung der Bücher nach Preis
  - Sortierung der Autoren nach Alter
  - ...
- In SQL ist der Standard für die Sortierung `ASC` (ascending, aufsteigend) und kann weggelassen werden
- Es können mehrere Spalten für die Ordnung angegeben werden, wobei die ersten Spalte stärker zählen als die letzten

Beispiel: Order By

```sql
-- Bücher absteigend nach Preis ordnen
SELECT * FROM books ORDER BY price DESC;

-- Bücher aufsteigend nach Bestand ordnen
SELECT * FROM books ORDER BY stock ASC;

-- oder so, da ASC der Standardwert ist
SELECT * FROM books ORDER BY stock;

-- Autoren alphabetisch nach Nachnamen ordnen, bei Gleichheit nach Vornamen
SELECT * FROM authors ORDER BY family_name, first_name;

-- Autoren alphabetisch nach Vornamen ordnen, bei Gleichheit nach Nachnamen
SELECT * FROM authors ORDER BY first_name, family_name;
```

Aufgabe: Order By

Bearbeiten Sie folgende Aufgaben in `it4all`. Orientieren Sie Sich dabei am Datenbankdiagramm!

- Szenario: `Amazon`
- Typ: `SELECT`
- IDs: 26 bis 30

Datenbanken anlegen

Um Tabellen erzeugen zu können, muss eine Datenbank vorhanden sein. Diese wird mit dem Befehl

CREATE DATABASE *datenbank_name*

erzeugt. Danach muss noch die gerade erstelle Datenbank ausgewählt werden, damit man mit ihr arbeiten kann:

USE *datenbank_name*

Wenn wir also eine Datenbank mit Namen „amazon“ erstellen wollen, brauchen wir folgende zwei Befehle:

```sql
CREATE DATABASE amazon;
USE amazon;
```

Tabellen anlegen

- Auf die Erstellung der Datenbank folgt die Erstellung der einzelnen Tabellen
- Bei der Erstellung der Tabellen muss auf die Reihenfolge geachtet werden, da einige Tabellen durch Fremdschlüssel von anderen abhängig sind

- Beispiele (siehe Datenbankschema, Abbildung 2):
  - `order_positions`, `ratings` und `wishlists` sind abhängig von `orders` und `books`
  - `orders` wiederum von `customers`
  - `books` von `publishers` und `authors`

Erstellen einer Tabelle

Beim Erstellen einer Tabelle werden definiert:

- Der Name der Tabelle
- Die Spalten der Tabelle
- Die Primär- und Fremdschlüssel

Die Definition einer Spalte besteht aus

- Dem Namen der Spalte
- Dem Datentypen der Spalte
- Optional zusätzlichen Attributen wie `UNIQUE`, `NOT NULL`, `DEFAULT`, ...

Datentypen in SQL

Datentyp	Verwendung
`BOOLEAN`	Wahrheitswerte
`INT`	Ganzzahlen
`FLOAT`	Fließkommazahlen
`VARCHAR(`*length*`)`	Text mit Maximallänge *length*
`TEXT`	Text mit (unendlicher) Länge
`DATE`	Datumsangaben
`ENUM(`*val1*`, `*val2*`, …)`	Werte mit festem Wertebereich *val1*, *val2*, …

Schlüssel

Primärschlüssel

- Identifizieren ein Tupel eindeutig
- Verwendung eines
  - natürlichen (Nutzername, Email, …) oder
  - künstlichen Schlüssels (meist `INT`-Wert, siehe Matrikelnummer)

Fremdschlüssel

- Verwendet, wenn Tabellen sich auf andere Tabellen beziehen
- Verweisen auf den Primärschlüssel und damit auf den entsprechenden Eintrag in der Tabelle, auf die sich die gerade verwendete bezieht
- Beispiele:
  - Spalte `author_id` in Tabelle `books` bezieht sich auf die Spalte `id` in `authors`
  - Spalte `publisher_id` bezieht sich auf die Spalte `id` in `publishers`

Beispiel: CREATE-Statement

```sql
CREATE TABLE publishers (
  -- Definition des (künstlichen) Primärschlüssels
  id             INT PRIMARY KEY,
  -- Definition name als eindeutig
  name           VARCHAR(100) UNIQUE,
  -- Definition als NOT NULL (darf nicht leer sein)
  contact_person VARCHAR(100) NOT NULL,
  address        VARCHAR(100),
  -- Angabe eines Defaultwertes
  phone          VARCHAR(100) DEFAULT '--/--'
);
```

Beispiel: CREATE mit Fremdschlüsseln

```sql
CREATE TABLE books (
  id           INT PRIMARY KEY,
  author_id    INT,
  publisher_id INT,

  -- Andere Definitionen ausgelassen, ...

  -- Spalte author_id verweist auf Spalte id in Tabelle authors
  FOREIGN KEY (author_id) REFERENCES authors (id),
  -- Spalte publisher_id verweist auf Spalte id in Tabelle publishers
  FOREIGN KEY (publisher_id) REFERENCES publishers (id)
);
```

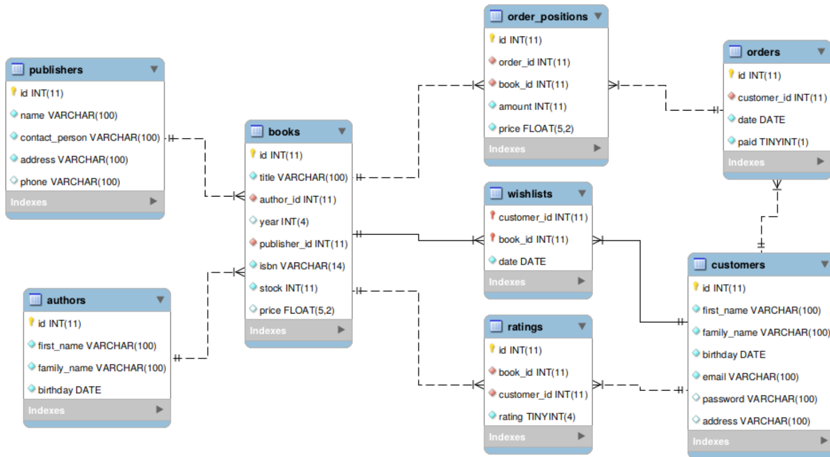
Composite Primary Keys

Manchmal ist ein Primärschlüssel nur eindeutig, wenn er mehrere Werte beinhaltet. Dann kann ein sogenannter *Composite* Primary Key angegeben werden:

```sql
CREATE TABLE wishlists (
  customer_id INT,
  book_id     INT,
  date        DATE,
  -- Primärschlüssel besteht aus beiden Spalten
  PRIMARY KEY (customer_id, book_id),
  -- Spalte customer_id ist gleichzeitig auch Fremdschlüssel
  FOREIGN KEY (customer_id) REFERENCES customers (id),
  FOREIGN KEY (book_id) REFERENCES books (id)
);
```

Aufgabe: CREATE TABLE

Erstellen Sie die `CREATE`-Statements für die Tabellen `ratings` und `customers`:



Lösung: CREATE TABLE ratings

```sql
CREATE TABLE ratings (
  id          INT PRIMARY KEY,
  book_id     INT NOT NULL,
  customer_id INT NOT NULL,
  rating      INT NOT NULL,
  FOREIGN KEY (book_id) REFERENCES books (id),
  FOREIGN KEY (customer_id) REFERENCES customers (id)
);
```

Lösung: CREATE TABLE customers

```sql
CREATE TABLE customers (
  id          INT PRIMARY KEY,
  first_name  VARCHAR(100) NOT NULL,
  family_name VARCHAR(100) NOT NULL,
  birthday    date         NOT NULL,
  email       VARCHAR(100) NOT NULL,
  password    VARCHAR(100) DEFAULT NULL,
  address     VARCHAR(100) DEFAULT NULL
);
```

INSERT

- Für das Einfügen von Tupeln in die Tabellen wird das `INSERT`-Statement benutzt.
- Dabei muss darauf geachtet werden, eventuelle Fremdschlüsselbeziehungen nicht zu verletzen (wenn der Wert, auf den verwiesen wird, nicht existiert, schlägt das Einfügen fehl).
- Für die Les- und Wartbarkeit empfiehlt es sich, die Namen der Spalten anzugeben, in die eingefügt wird.
- Wenn alle Spalten in der gleichen Reihenfolge wie in der Definition verwendet werden, können diese auch weggelassen werden.
- Es können auch mehrere Tupel (Zeilen) durch Komma getrennt eingefügt werden

Beispiel: INSERT

```sql
-- Einfügen mit Angabe der Spalten
INSERT INTO authors (`id`, `first_name`, `family_name`, `birthday`)
VALUES (1, 'Joanne K.', 'Rowling', '1965-07-31');

-- Mehrfaches Einfügen ohne Angabe der Spalten
INSERT INTO authors
VALUES (2, 'George', 'Orwell', '1903-06-25'),
       (3, 'John Ronald Reuel', 'Tolkien', '1892-01-03'),
       (4, 'Antoine', 'de Saint-Exupéry', '1900-06-29'),
       (5, 'Robert', 'Ludlum', '1927-05-25');
```

Aufgabe: INSERT

Erstellen Sie ein `INSERT`-Statement für folgende Kunden:

id	first_name	family_name	birthday	email	password
	address				
1	Ilrich	Horn	1982-11-04	ilrich_1570@aol.com	XXXX
	06779 Marke/Rudolfstraße 37c				
2	Hilma	Woll	1991-03-10	hilma_1411@web.de	XXXX
	01683 Nossen/Amselsteg 40				
3	Hilda	Endres	1999-07-08	hilda_1181@aol.com	XXXX
	27412 Tarmstedt/Johnsbacher Weg 17				
4	Sieghard	Jung	1990-04-02	sieghard_1474@yahoo.com	XXXX
	07570 Wünschendorf/Eichelseeweg 95b				
5	Mechthilde	Donath	1978-11-30	mechthilde_1442@uni.de	XXXX
	84032 Altdorf/Arnsberger Straße 39				

Lösung: INSERT

```sql
INSERT INTO customers
VALUES (1, 'Ilrich', 'Horn', '1982-11-04', 'ilrich_1570@aol.com',
         'XXXX', '06779 Marke/Rudolfstraße 37c'),
       (2, 'Hilma', 'Woll', '1991-03-10', 'hilma_1411@web.de',
         'XXXX', '01683 Nossen/Amselsteg 40'),
       (3, 'Hilda', 'Endres', '1999-07-08', 'hilda_1181@aol.com',
         'XXXX', '27412 Tarmstedt/Johnsbacher Weg 17'),
       (4, 'Sieghard', 'Jung', '1990-04-02', 'sieghard_1474@yahoo.com',
         'XXXX', '07570 Wünschendorf/Eichelseeweg 95b'),
       (5, 'Mechthilde', 'Donath', '1978-11-30', 'mechthilde_1442@uni.de',
         'XXXX', '84032 Altdorf/Arnsberger Straße 39');
```

UPDATE: Änderung Tabelleninhalt

Ein UPDATE-Statement besteht aus 3 Teilen:

```
-- Tabelle
UPDATE books
    -- Änderung
    SET stock = 100000
    -- Bedingung
    WHERE id = 4;
```

- Angabe der Tabelle, in der Änderungen durchgeführt werden
- Angabe der Änderungen, die durchgeführt werden
- Bedingungen, die die Spalten angeben, in denen die Änderungen durchgeführt werden

- Die Bedingungen funktionieren analog zu SELECTs
- *Achtung*: Wird keine Bedingung angegeben, werden alle Spalten der Tabelle geändert

Aufgabe: UPDATE

Schreiben Sie Statements für folgende Änderungen:

- Der Bestand des Buches „1984“ soll auf 54.000 geändert werden
- Der Kunde 12 hat seine Bewertung für das Buch 1 auf 3 Sterne aktualisiert
- Der Kunde mit der ID 13 hat sein Passwort auf „ZZZZ“ geändert
- Das Buch „Der kleine Hobbit“ kostet nur noch 7,00 €

Lösung: UPDATE

```sql
UPDATE books
  SET stock = 54000
  WHERE title = '1984';

UPDATE ratings
  SET rating = 3
  WHERE customer_id = 12
    AND book_id = 1;
```

```sql
UPDATE customers
  SET password = 'ZZZZ'
  WHERE id = 13;

UPDATE books
  SET price = 7.00
  WHERE title = 'Der kleine Hobbit';
```

DELETE: Löschen von Tabelleninhalt

Ein DELETE-Statement besteht aus 2 Teilen:

```
DELETE
  FROM author
  WHERE id = 2;
```

- Angabe der Tabelle, in der gelöscht wird
- Bedingungen, die die Spalten angeben, die gelöscht werden

- Die Bedingungen funktionieren analog zu SELECTs
- *Achtung*: Wird keine Bedingung angegeben, werden alle Spalten der Tabelle gelöscht
- Beim Löschen muss auf Fremdschlüsselbedingungen geachtet werden

DELETE: Weitere Beispiele

```sql
-- Buch mit dem Titel 1984 löschen
DELETE FROM books WHERE title = '1984';

-- Alle Kunden löschen, deren Vorname mit G beginnt
DELETE FROM customers WHERE first_name like 'G%';

-- Alle Bewertungen des Kunden 2 mit
-- einer schlechten Bewertung löschen
DELETE FROM ratings
  WHERE customer_id = 2
  AND rating < 3;
```

Aufgabe: DELETE

Schreiben Sie Statements für folgende Änderungen:

- Löschen Sie den / die Kunden mit dem Nachnamen „Fritz“
- Löschen Sie die Bücher des Autoren mit der ID 2
- Löschen Sie alle Bücher mit einem Bestand von weniger als 1000 Exemplaren

Lösung: DELETE

```sql
DELETE
  FROM customers
  WHERE family_name = 'Fritz';

DELETE
  FROM books
  WHERE author_id = 2;

DELETE
  FROM books
  WHERE stock < 1000;
```

UML

UML: Hierarchie der Diagramme

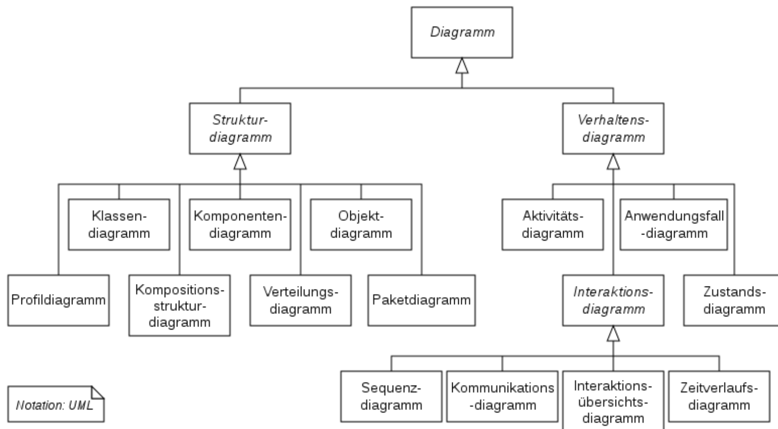


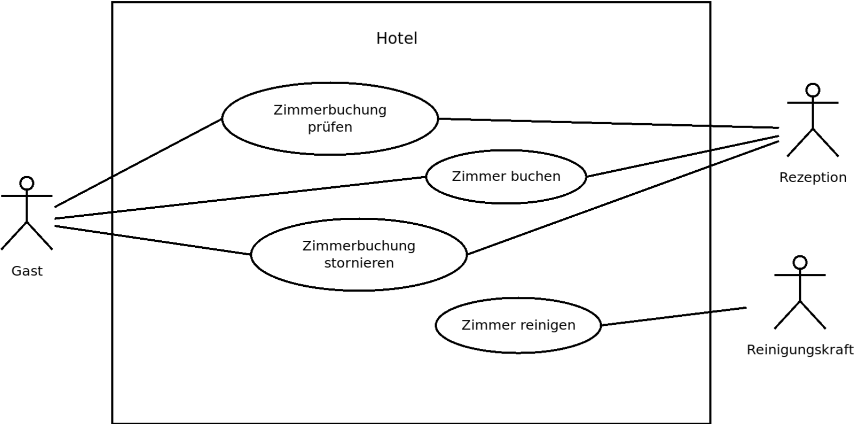
Abbildung 3: Hierarchie der Uml Diagramme (als Klassendiagramm)

Use-Case-Diagramm: Basics

… stellt Anwendungsfälle und Akteure mit ihren jeweiligen Abhängigkeiten und Beziehungen dar

- Akteure werden als Strichmännchen dargestellt
  - Reale Personen wie zum Beispiel Kunden oder Administratoren
  - Systeme ($\approx$ Computerprogramme, dann durch PC-Icon dargestellt)
- Die einzelnen Anwendungsfällen werden in Ellipsen mit einer passenden Beschreibung dargestellt
- Rechtecke geben „Systemgrenzen“ an

Beispiel: Use-Case-Diagramm



Klassendiagramme

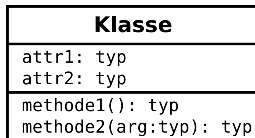
Diagramm zur Darstellung von Klassen, Schnittstellen sowie deren Beziehungen

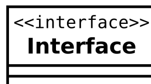
Notationselemente:

- Klassen
  - („Normale“) Klassen
  - Abstrakte Klassen
  - Schnittstellen
- Beziehungen zwischen Klassen
  - Vererbungen
  - Assoziationen
  - Aggregationen
  - Kompositionen

Klassen, Abstrakte Klassen und Schnittstellen

- Klassen und Schnittstellen werden in einem Rechteck mit 3 Teilrechtecken dargestellt.
- Im oberen Teil stehen
  - der Name der Klasse oder Schnittstelle und
  - der Modifikator `<<interface>>` oder `<<abstract>>`, falls es sich um eine Schnittstelle oder eine abstrakte Klasse handelt
- Im mittleren Teil stehen die Attribute der Klasse
- Im unteren Teil stehen die Methoden der Klasse





Assoziationen

- Eine Assoziation stellt eine Beziehung zwischen zwei Klassen dar.
- Assoziationen werden mit einer Linie ohne Pfeilspitzen dargestellt.
- An den Linienenden werden jedoch Multiplizitäten vermerkt, die angeben, wie viele Instanzen einer Klasse an einer Assoziation beteiligt sind.
- In den meisten Fällen reichen die Multiplizitäten 1 (genau *eine* Instanz) und * (beliebig viele Instanzen) aus.

Abbildung 4: Beispiele Assoziationen

Vererbungen

Eine Vererbung stellt dar, dass eine Klasse eine andere erweitert ($\Rightarrow$ von ihr erbt).

Eine Vererbungsbezeihung wird mit einer Linie mit einer geschlossenen, nicht ausgefüllten Pfeilspitze am Ende der *erweiterten* Klasse modelliert.

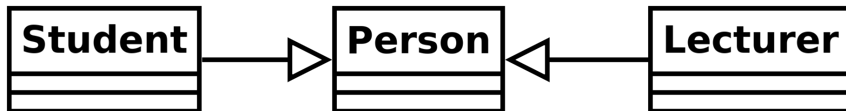


Abbildung 5: `Student` und `Lecturer` erweitern (erben von) `Person`

Aggregation und Komposition

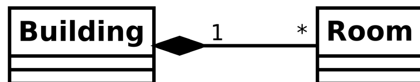
- Die Aggregation und die Komposition „erweitern“ jeweils die Assoziation und stellen ein *Teil-Ganzes-Relation* dar.
- Beide werden mit einer Raute an der Seite des „Ganzen“ dargestellt

Komposition

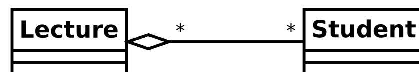
- Ausgefüllte Raute
- Teil existiert **nicht** ohne Ganzes

Aggregation

- **Nicht** ausgefüllte Raute
- Teil kann **ohne** Ganzes existieren



(a) Komposition



(b) Aggregation

Abbildung 6: Komposition und Aggregation

Klassendiagramme versus Python: Vererbung

```python
class Person:
    def __init__(self, name: str):
        self.name: str = name
```

```python
class Student(Person):
    def __init__(self, name: str):
        super().__init__(name)
```

```python
class Lecturer(Person):
    def __init__(self, name: str):
        super().__init__(name)
```

Klassendiagramme versus Python: Assoziationen

Assoziationen werden normalerweise als Attribut aufgelöst.

Bei einer 1:*-Abbildung (beziehngsweise *:1) gibt es zwei Möglichkeiten:

1. Abbildung als Attribut in Klasse auf der *-Seite
2. Abbildung als Listenattribute in Klasse auf der 1-Seite

Die Entscheidung, welche Möglichkeit verwendet wird, sollte je nach Anwendungsfall getroffen werden.

Eine *:*-Abbildung muss über eine zusätzliche Klasse modelliert werden, die jeweils die möglichen Paare speichert.

Beispiel: Auflösung mit Einzelattribut

```python
class Building:
    def __init__(self, height: float):
        self.height: float = height


class Room:
    def __init__(self, number: int, building: Building):
        self.number: int = number
        self.building: Building = building
```

Beispiel: Auflösung mit Listen

```python
from typing import List


class Room:
    def __init__(self, number: int):
        self.number: int = number


class Building:
    def __init__(self, height: float, rooms: List[Room]):
        self.height: float = height
        self.rooms: List[Room] = rooms
```

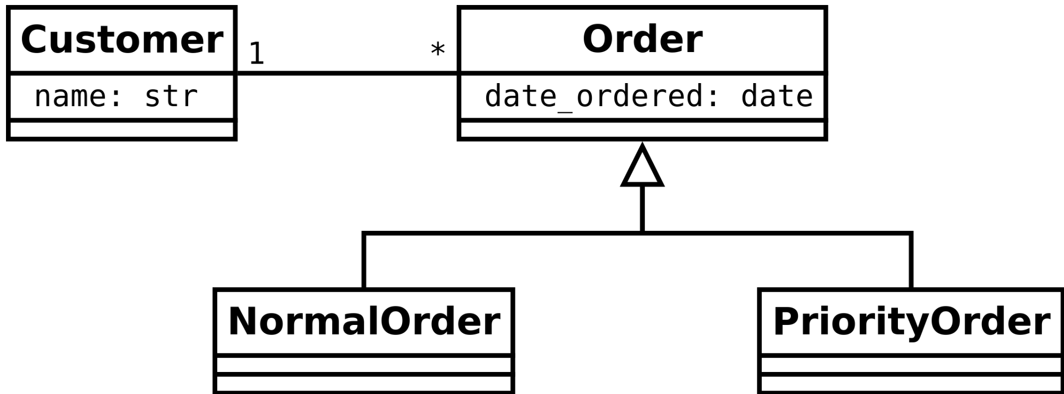
Aufgabe: Python nach UML

Erstellen Sie ein UML-Klassendiagramm, das folgendem Python-Code entspricht:

```
class Order:
    def __init__(self, date_ordered: date):
        # Auslassung...

class NormalOrder(Order):
    def __init__(self, date_ordered: date):
        # Auslassung...

class PriorityOrder(Order):
    def __init__(self, date_ordered: date):
        # Auslassung...
```

```
class Customer:
    def __init__(self, name: str,
                 orders: List[Order]):
        # Auslassung...
```

Lösung: Python nach UML



Aktivitätsdiagramme