Verteilte Systeme

Serverseitige Webentwicklung

Silko Kruse

Inhalt (1)

- Einleitung, Einführung in PHP, Variablen, Typen, Konstanten, Zuweisung und Operatoren
- Benutzerinteraktion
- Kontrollstrukturen
  - Programmverzweigungen
  - Schleifenanweisungen
- Dateien hochladen
- Felder, foreach-Kontrollstruktur, Zeichenketten
- Funktionen, Referenzparameter, Geltungsbereiche
- Dateioperationen

Inhalt (2)

- Objektorientierte Programmierung in PHP
  - Methoden und Variablen überladen
  - Konstruktoren
  - Komposition
  - Vererbung
  - Abstrakte Klassen
  - Interfaces
- Datenbanken - Grundlagen, Relationale Datenmodellierung
- PHP und MySQL
- Cookies, Sessions

Inhalt (3)

- Templates
- Normalformen
  - 1. Normalform
  - 2. Normalform, Anomalien
  - 3. Normalform
  - Dualitätsprinzip
  - Join
- Entity-Relationship-Modell
  - Datenbankdesign mit dem ERM
  - Relationale Datenmodellierung

Webauftritt: Früher

- HTML-Seiten
  - Überwiegend textuelle Information
  - Bilder
  - Formatierungsanweisungen
- Gestalterischer Aufwand vergleichsweise gering
- Inhalte statisch
- Keine Interaktivität
  - Kein zweiseitiger Datenaustausch mit Server

Webauftritt: Heute

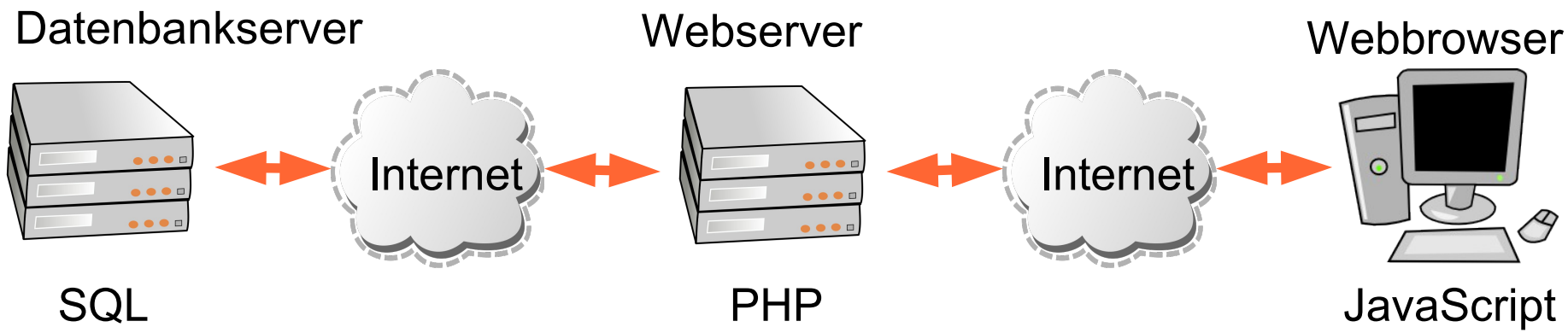
- Webauftritt oftmals Benutzeroberfläche komplexer Webanwendungen
- Hoher Grad an Interaktivität
- Inhalte dynamisch
  - Dynamisch generierte Seiteninhalte als Folge von z. B. Anwendungssteuerung und Benutzereingaben, Benutzeranfragen, Standort, Browser- oder Gerätetyp
  - Seiteninhalte werden dynamisch an Kundenbedürfnisse angepasst
- Dynamisierung sowohl client- (JavaScript) wie auch serverseitig (z. B. durch PHP)

Dynamische Webauftritte (1)

- HTML-Seiten für den Browser entstehen, indem Inhalte *programmatisch* zusammengefügt werden
  - Dabei werden ggf. vorhandene Vorlagen genutzt
- Ein Scriptinterpreter kooperiert in diesem Zusammenhang mit dem Webserver und einer Datenbank
- Datenbankgestützte programmierte Webauftritte bzw. -anwendungen
- Wenig umsetzungs- und wartungsfreundlich für die Anbieter
- Es entsteht der Bedarf für eine einfache Umsetzungs- und Wartungsmöglichkeit
- Dies ist die Aufgabe eines sogenannten **Content-Management-Systems**

Dynamische Webauftritte (2)

- Webauftritt verarbeitet im Hintergrund extern eingelesene Daten
  - Es gilt nicht das *Seitenwechselparadigma*: Der Inhalt der Seite ändert sich, ohne dass diese neu geladen werden muss
  - Es gilt das *Anwendungsparadigma*: Die Webseite verhält sich wie ein Desktopprogramm
    - Es ist möglich, mit der Anwendung weiter zu interagieren, während neue Daten vom Server geladen werden
- Die Anwendung kann vor dem Herunterladen individuell angepasst werden



Datenaustausch

- Client-Seite: AJAX-Technologie
  - Ursprünglich für: **A**synchronous **J**avaScript **A**nd **X**ML
  - Datenaustauschformate: XML, JSON
  - Laden der Daten mit Hilfe des **`XMLHttpRequest`**-Objekts
  - Ermöglicht das asynchrone Laden von Daten aus einem Webauftritt heraus, ohne dass hierfür ein Link erforderlich ist
- Server-Seite: PHP und SQL
  - Webseiteninhalte werden dynamisch mittels PHP-Interpreter generiert, inklusive JavaScript
  - Daten werden hierfür mittels SQL aus einer Datenbank ausgelesen und sofort Teil des Webseitencodes oder als XML oder JSON „verpackt“ versendet

Content-Management-System (1)

- Kurzform: CMS
- System zur Generierung und Verwaltung der Inhalte eines Webauftritts
- CMS basiert üblicherweise selbst auf einem dynamischen Webauftritt
- Der Webauftritt selbst wird zum System bzw. das System stellt den Webauftritt bereit
  - Erlaubt Erstellung und Aktualisierung der Inhalte mittels Webinterface
    - Expliziter Zugriff auf dahinter liegende Datenbank entfällt
    - Über die Bedienung des Systems hinaus sind i. d. R. keine speziellen Kenntnisse erforderlich
- Beispiele: Wordpress, TYPO3, Joomla, Drupal

Content-Management-System (2)

- Webinterface stellt i. d. R. eine Benutzerverwaltung mit verschiedenen Rollen und zugehörenden Kompetenzen zur Verfügung
  - Nur die zur jeweiligen Rolle gehörenden Modifikationen können dann am Webauftritt vorgenommen werden
- Begriff sehr weit gefasst
  - Bezeichnet z. T. auch schon Systeme, die über keine Webschnittstelle für die Wartung verfügen
- Es existieren zwei unterschiedliche Typen:
  - Statische Systeme
  - Dynamische Systeme

Statische CMS

- Scripte werden vor Aufruf der Seiten interpretiert und fertig generierte HTML-Seiten auf dem Server installiert
- Vorzuziehen für stark frequentierte Server und Webauftritte, die sich inhaltlich nur sporadisch ändern
- Verringert Last durch Scriptinterpreter
  - Webseiten werden nicht bei jedem Seitenaufruf wieder neu dynamisch generiert
- Webseiten können aber clientseitig über JavaScript dynamisiert sein

Dynamische CMS

- Scripte werden bei Aufruf der Webanwendung oder -seite jeweils neu ausgeführt
- Erlaubt Konstruktion und Aktualisierung der darzustellenden Inhalte in Echtzeit
- Uneingeschränkte Interaktion zwischen Client und Server
- Verursacht wesentlich höhere Interpreterlast
- Vorzugsweise für Webauftritte bzw. -anwendungen geeignet, die sich inhaltlich kontinuierlich ändern

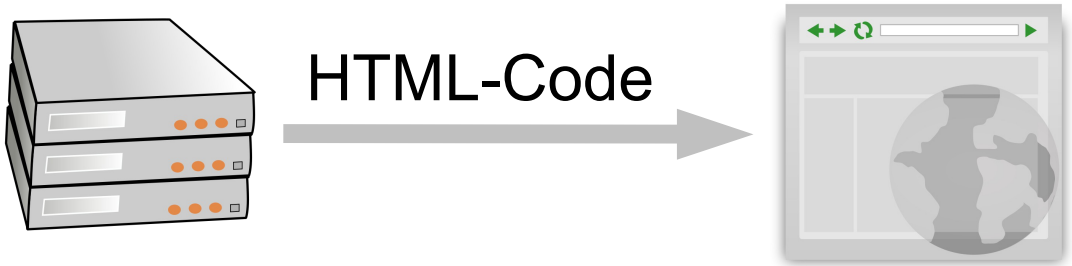
Dynamische Webseiten

- Wie lassen sich Inhalte auf der Serverseite dynamisch in eine Webseite bzw. -anwendung einfügen?
- Traditionelle Methode:
  - Common Gateway Interface (CGI)

Common Gateway Interface (CGI)

- CGI-Schnittstelle
  - Schnittstelle zwischen Webserver und externen Programmen
- Webbrowser fordert Scriptdatei an
- Webserver interpretiert Scriptdatei nicht selbst, sondern übergibt diese an Scriptinterpreter
  - Scriptinterpreter ist ein unabhängiges Programm
- Beispiel: Perl-Script und Perl-Interpreter
  - Scripts zumeist in eigenem Verzeichnis auf dem Server gespeichert
- Ausgabe des Interpreters wird vom Webserver entgegengenommen und an den Webbrowser geschickt

CGI versus JavaScript



Webserver

CGI-Schnittstelle:
Datenverarbeitung
auf dem Server-Rechner
(z. B. Gästebuch- oder
Blogeinträge einfügen)

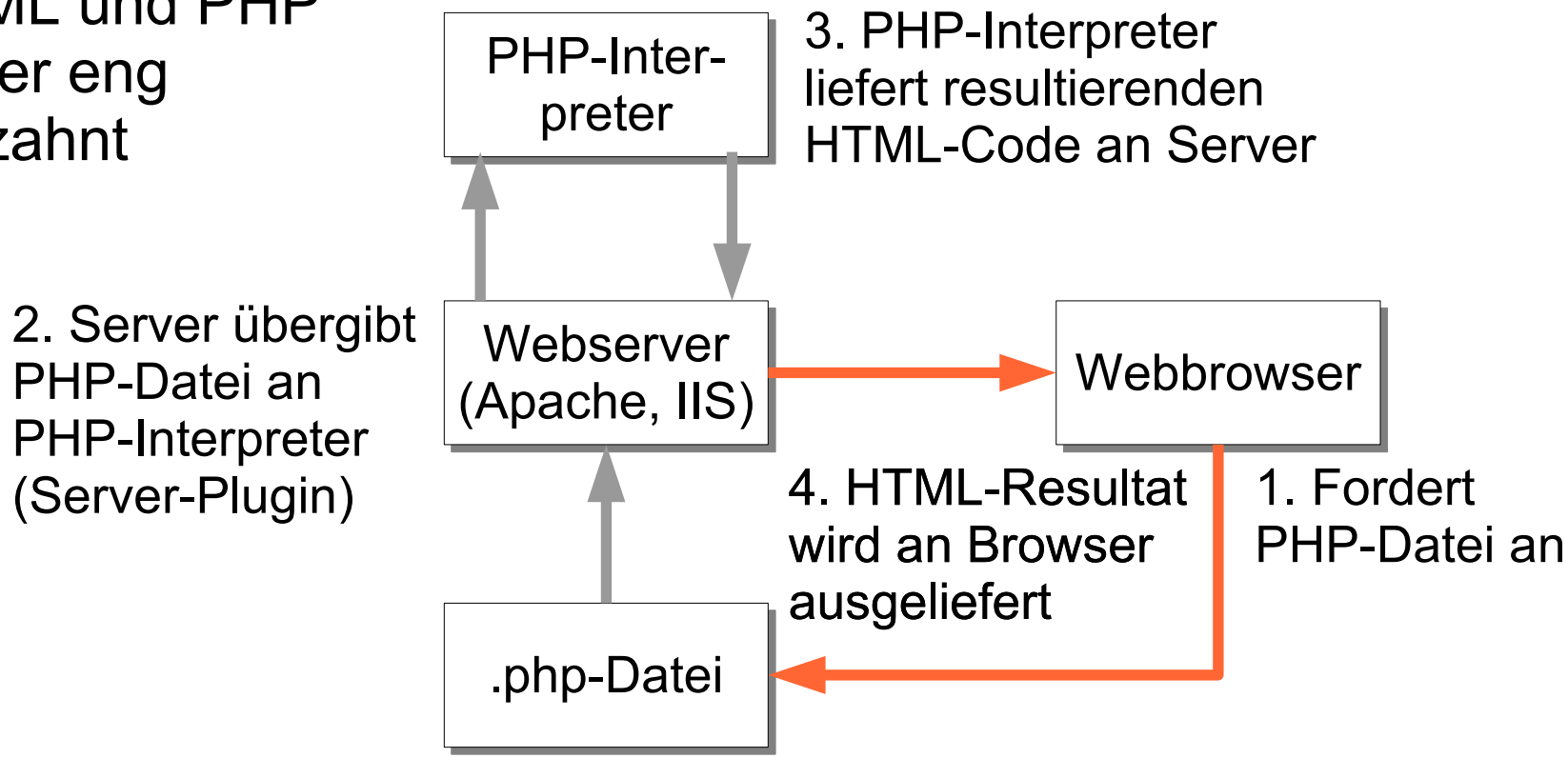
Webbrowser

Datenverarbeitung mittels JavaScript
auf dem Client-Rechner
(z. B. AJAX-Anwendung)

- CGI/Perl: Scripts in eigener Datei
- JavaScript: Scripts als Teil der Webseite oder inkludiert

PHP

- **Hypertext Preprocessor**
  - Scripts sind integraler Bestandteil der Webseite. Durch spezielle Auszeichnungsbefehle im HTML-Code markiert
  - HTML und PHP daher eng verzahnt



PHP versus Perl (1)

- Perl:
  - Universelle Scriptsprache
    - Nicht speziell für dynamische Webseiten entwickelt
    - Kann unabhängig von HTML benutzt werden
  - Perl-Interpreter verursacht externen Prozess
    - Folglich höhere Serverlast
  - Plattformunabhängig

PHP versus Perl (2)

- PHP:
  - Auf der Grundlage der Erfahrungen mit Perl beim Webdesign entwickelt
  - Syntax erbt von Perl und Java bzw. C-Familie
  - Wird als Server-Plug-In gestartet:
    - Kein externer Prozess: Daher ressourcenschonender
    - Unter Windows: DLL
  - Plattformunabhängig
  - Inzwischen universelle Scriptsprache
    - Kann unabhängig von HTML benutzt werden

PHP: Installation

- Kostenlos verfügbar, z. B. als Teil eines XAMPP-Pakets
- Beispiel: Apache unter Windows
  - In der Regel als Modul vorinstalliert
  - Sonst in Konfigurationsdatei **`httpd-xampp.conf`** eintragen

Einführung in PHP

Ein einfaches PHP-Script (1)

Beispiel: **helloWorld.php**

```
<html>
<head>
<title>Hello World</title>
</head>
<body>
<p>Ein Gruß vom PHP-Interpreter:
<?php
  echo("<b>Hello World!</b>");
?>
</p>
</body>
</html>
```

Anfang des PHP-Codes

PHP-Code eingebettet in HTML-Code

Ende des PHP-Codes

Auszeichnung von PHP-Code

```
<?php

    // PHP-Code

?>
```

- XML-Stil
- Häufigste Variante
- Empfohlen wegen der XML-Kompatibilität

Andere Varianten

- SGML-Stil (Standard Generalized Markup Language)

```
<? // PHP-Code ?>
```

- ASP-Stil (Active Server Pages)

```
<% // PHP-Code %>
```

  - Setzt ASP_TAGS gleich ON in PHP.INI voraus

- JavaScript-Stil

```
<Script language="PHP">
   // PHP-Code
</Script>
```

Kommentare

```
//  Dies ist ein einzeiliger Kommentar

#   Dies ist auch ein einzeiliger Kommentar

/*
    Dies ist ein mehrzeiliger Kommentar.
    Das Ende des Kommentars wird gesondert
    angezeigt:
*/
```

- // bzw. /* */: An C, C++, Java angelehnt
- Kommentare zum Verständnis des Codes wichtig!

Installation

- PHP-Datei muss auf dem Webserver installiert werden
  - Üblicherweise im `htdocs` Verzeichnis (Apache)
- Einfaches Anschauen im Browser (wie bei HTML-Dateien) nicht möglich!
  - PHP-Datei muss zuvor interpretiert werden
- PHP-Datei wird durch Endung `.php` kenntlich gemacht

Ein einfaches PHP-Script (2)

Beispiel: **helloWorld.php**

```
<html>
<head>
<title>Hello World</title>
</head>
<body>
<p>Ein Gruß vom PHP-Interpreter:
<?php
  echo("<b>Hello World!</b>");
?>
</p>
</body>
</html>
```

- PHP-Scripts bestehen aus einer Reihe von Befehlen
  - Werden sukzessive vom Interpreter abgearbeitet
- Befehle werden mit Semikolon abgeschlossen

- Dieser Befehl ruft den PHP-Befehl echo auf
- echo wird die Zeichenkette "Hello World!" übergeben
- echo übernimmt die Zeichenkette und platziert diese an der aktuellen Position in der HTML-Datei
- echo ist keine Funktion und kann auch ohne Klammern benutzt werden

Resultat der Interpretation

Beispiel: **helloWorld.php**

```
<html>
<head>
<title>Hello World</title>
</head>
<body>
<p>Ein Gruß vom PHP-
Interpreter:
<?php
  echo("<b>Hello World!</b>");
?>
</p>
</body>
</html>
```

```
<html>
<head>
<title>Hello World</title>
</head>
<body>
<p>Ein Gruß vom PHP-
Interpreter:
 <b>Hello World!</b>
</p>
</body>
</html>
```

Möglichkeiten der Einbettung in HTML

- Integration in den HTML-Code
  - Besonders geeignet für Webseiten, die über viel HTML-Code verfügen, weil die Präsentation im Vordergrund steht
  - PHP-Modus wird sofort nach Ausgabe wieder beendet
  - Vermeidet Erzeugung langer HTML-Passagen durch PHP-Befehle
- Vollständige Erzeugung des HTML-Codes mittels PHP
  - Besonders geeignet für Webseiten, bei denen der Programmcharakter im Vordergrund steht

Beispiel für Integration in den HTML-Code

```
<!doctype html>
<html>
<head>
<meta charset="utf-8">
<title>Hello World</title>
</head>
<body text="#000000" bgcolor="#FFFFFF">

<p>Ein Gruß vom PHP-Interpreter:
<?php echo("<b>Hello World!</b>");?></p>

</body>
</html>
```

Bsp. für vollständige Erzeugung mittels PHP

```
<?php
echo("<!doctype html">
<html>
<head>
<meta charset="utf-8">
<title>Hello World</title>
</head>
<body text=\"#000000\" bgcolor=\"#FFFFFF\">
<p>Ein Gruß vom PHP-Interpreter:
<b>Hello World!</b></p>
</body>
</html>
");
?>
```

- " ist PHP-Symbol!
- Muss daher durch \" ersetzt werden
  - \ bedeutet: Interpretiere das folgende Symbol nicht als PHP-Symbol

Variablen, Typen, Konstanten

Variablen

- Variablen beginnen mit einem `$`, gefolgt vom Variablennamen
- Variablenname:
  - Beginnt mit einem Unterstrich (_) oder mit einem Buchstaben
  - Besteht aus einer beliebigen Anzahl von Buchstaben, Zahlen oder Unterstrichen
  - Groß- und Kleinschreibung wird berücksichtigt (case-sensitive)
  - Variablennamen sollten wie üblich selbsterklärend sein
- Beispiel: `$variable`
- Es gibt selbst definierte und in der Entwicklungsumgebung vordefinierte Variablen
  - Letztere werden zu den Themen passend erläutert

Typen

- PHP ist eine locker typisierte Sprache
  - Variablen werden nicht als zu einem bestimmten Typ gehörend deklariert
  - Typen können aber auch explizit für Funktionsargumente, Rückgabewerte und für Klasseneigenschaften benutzt werden
- Jede Variable kann jeglichen Datentyp enthalten
  - Zahl, Zeichenkette etc.
- Eine Variable darf während ihrer Lebenszeit für Werte unterschiedlicher Datentypen benutzt werden

Grundlegende Typen

- Vier skalare Typen (Grundtypen, primitive Datentypen):
  - Boolean (bool), Integer (int), Gleitkommazahl (float), Zeichenkette (string)
- Zwei abstrakte, d. h. zusammengesetzte Typen:
  - array, object
- Typ Ressource (resource)
  - Repräsentiert z. B. Datenbank- oder FTP-Verbindungen

Typ: `bool`

- Folgende Werte werden als FALSE interpretiert:
  - der Wert **`FALSE`** des Typs **`bool`**
  - die Integerzahl 0
  - die Gleitkommazahl 0.0
  - die leere Zeichenkette und die Zeichenkette "0"
  - ein Feld ohne Elemente
  - der spezielle Wert **`NULL`**
    (einschließlich nicht definierter Variablen)
- Jeder andere Wert wird als TRUE interpretiert

Typ: `string`

- In PHP entspricht ein Zeichen einem Byte
  - D. h., es sind 256 Zeichen möglich
  - Unterstützung für Unicode über HTML-Head-Element
- Strings können beliebig lang sein (max. 2GB unter 32-bit)
- Schreibweisen
  - Einfache Anführungszeichen (single quoted)
    - `echo('Das ist ein einfacher String');`
    - Variablen werden in single-quoted Strings nicht ausgewertet!
  - Doppelte Anführungszeichen (double quoted)
    - `echo("Das ist ein einfacher String");`
  - Heredoc Syntax für mehrzeilige Strings (siehe hierzu z. B. https://code-basics.com/languages/php/lessons/heredoc)

Konstanten

- Wert einer Konstanten zur Laufzeit eines Skripts nicht änderbar
- Namen folgen gleichen Regeln wie Variablennamen
  - Konvention: Nur GROSSBUCHSTABEN verwenden
- Konstante wird mittels Funktion **`define()`** definiert
- Wert einer Konstanten durch Angabe ihres Namens auslesbar
- Kein **`$`** erforderlich
- Beispiel:

```
define("MEINE_KONSTANTE", "hat einen Wert");
echo(MEINE_KONSTANTE);
```

  - Ausgabe: **`hat einen Wert`**

Zuweisung und Operatoren

Zuweisung

- Variablen erhalten einen Wert mittels einer Zuweisung:

```
$variable = 3;

$variable = "Drei";

$var1 = $var2;

define("MYNUM", 1);

$var = MYNUM+1;

echo($var); // Ausgabe: 2
```

- Das Gleichheitszeichen ist der **Zuweisungsoperator**

Variablenname zur Laufzeit festlegen

- Wert der Variable als Variablenname einer neuen Variable
- Beispiel:

```
$var = "Hallo";
```

- Variablenwert wird als Bezeichner einer neuen Variable genommen

```
$$var = "Welt";
```

- Nun existieren zwei Variablen:
  - **`$var`** enthält **`"Hallo"`**
  - **`$Hallo`** enthält **`"Welt"`**

Zuweisung durch Referenzierung

- Der Wert der neuen Variable ist Referenz zur Ursprungsvariable
- Beide Variablen benennen dieselbe Speicherstelle
  - Zuweisungen an neue Variable ändern auch Wert der ersten Variable und umgekehrt
- Für die Zuweisung per Referenz wird ein **&** vorangestellt
- Beispiel:

```
$var1 = 25;

$var2 = &$var1; // var2 referenziert Wert von var1

$var3 = &(24 * 7); // NICHT erlaubt, da (24 * 7)
                   // keine Variable ist!
```

Geschachtelte Zuweisung

- Der Wert einer Zuweisung ist der zugewiesene Wert
- Dieser kann in einem übergeordneten Ausdruck verwendet werden
- Beispiel:
- `$a = ($b = 4) + 5;`
  - Nach Auswertung ist `$a` gleich `9` und `$b` gleich `4`

NULL

- Der spezielle Wert **`NULL`** steht dafür, dass eine Variable keinen Wert hat

- Eine Variable wird als **`NULL`** interpretiert, wenn

  - ihr bis jetzt kein Wert zugewiesen wurde,

  - ihr die Konstante **`NULL`** als Wert zugewiesen wurde,

    - Beispiel: **`$var = NULL;`**

  - sie mit **`unset()`** gelöscht wurde

Arithmetische Operatoren

`$a + $b`	Addition:	Summe von $a und $b
`$a - $b`	Subtraktion:	Differenz von $a und $b
`$a * $b`	Multiplikation:	Produkt von $a und $b
`$a / $b`	Division:	Quotient von $a und $b
`$a % $b`	Modulus:	Rest von $a geteilt durch $b

Beispiele:

```
$var1 = 1+1;
$var2 = 100.5 - 2;
$var3 = $var1 + $var2;
$var4 = $var3 / 27.2;
$var4 = $var4 * 2;
```

Logische Operatoren

$a && $b	Und
$a and $b	Und (niedrigerer Rang)
$a || $b	Oder
$a or $b	Oder (niedrigerer Rang)
$a xor $b	Entweder Oder
! $a	Nicht

Vergleichsoperatoren (1)

- $a == $b Gleich: TRUE, wenn $a gleich $b
- $a != $b Ungleich: TRUE, wenn $a nicht gleich $b
- $a <> $b Ungleich: TRUE, wenn $a nicht gleich $b
- $a < $b Kleiner: TRUE, wenn $a kleiner als $b
- $a > $b Größer: TRUE, wenn $a größer als $b
- $a <= $b Kleiner-Gleich: TRUE, wenn $a kleiner oder gleich $b
- $a >= $b Größer-Gleich: TRUE, wenn $a größer oder gleich $b

Vergleichsoperatoren (2)

- $a === $b Identischer Typ- und Wert
  - TRUE, wenn $a gleich $b ist und beide vom gleichen Typ sind
- $a !== $b Nicht identischer Typ- und/oder Wert
  - TRUE, wenn $a nicht gleich $b ist, oder wenn beide nicht vom gleichen Typ sind

Zeichenketten verbinden

- Verkettungsoperator (Konkatenationsoperator): .
- Beispiele:

```
$var = "Hello " . "World";  // var enthält
                            // "Hello World"
```

- Variablenauswertung in Abhängigkeit vom Anführungszeichen:

```
echo("<p>var: " . $var . "</p>"); // Auswertung
echo("<p>var: $var </p>");        //      "
echo("<p>var: " . "$var </p>");   //      "
echo('<p>var: $var </p>'); // keine Auswertung!
```

Fehlermeldungen unterdrücken

- @-Operator
- Funktioniert bei Ausdrücken
  - Führt zum Ignorieren aller Fehlermeldungen, die von diesem Ausdruck erzeugt werden könnten
  - Z. B. Funktionen, Variablen, normale Ausdrücke
- Beispiele:
- ```
  $my_file = @file ('nicht_vorhandene_Datei') or die ("Datei konnte nicht geöffnet werden: Fehler war:'" . error_get_last()['message'] . "'");
  ```
- ```
  @$var2 = 1/0; // Hier funktioniert es nicht
  ```
  - Kritischer Fehler: Diese werden nicht ignoriert

Benutzerinteraktion

Benutzerinteraktion

- Durchgeführt in Form von Benutzeranfragen, die an den Webserver geschickt werden
- Webserver bearbeitet Anfrage und antwortet mit dynamisch erzeugtem HTML-Code
- Anfragen werden mittels *URL-Querystring* übermittelt

Anfrage mittels URL-Querystring

- anfrage.html:

```
<a href="antwort.php?name=Browser"> Webserver
bitte melden! </a>
```

- antwort.php:

```
<?php
  echo("Habe verstanden, $_GET[name]!");
?>
```

- Link in *anfrage.html* wird benutzt, um *antwort.php* zu laden
- Wert der Variablen **`name`** im Querystring wird automatisch in PHP-Variable **`$_GET[name]`** geschrieben (Feldelement)

Mehrere Variablen

- anfrage2.html:

```
<a href="antwort2.php?client=Browser&
   ort=Labor3">Webserver bitte melden!</a>
```

- antwort2.php:

```
<?php
  echo("Habe verstanden, $_GET[client] in
        $_GET[ort]!");
?>
```

- Variablen werden im Querystring durch & voneinander getrennt

Dateneingabe

- Bisher noch keine echte Interaktion zwischen Webserver und Browser
  - Daten fest codiert im Querystring
- Benutzer soll Daten im Browser eingeben können
  - Suchanfragen (Google etc.)
  - Adressinformationen
  - Bestellung mit Formular
  - Kundendaten etc.
- Hierfür können HTML-Formulare benutzt werden

Formular definieren

- Formular kann an beliebiger Stelle innerhalb eines HTML-Body definiert werden:

```
<form action="antwort2.php" method="get">

<!-- hier folgen die Formularelemente -->

</form>
```

- `<form>...</form>` definiert das Formular
- `action:`
  - Script, E-Mail-Adresse (mit mailto:) oder HTML-Datei mit JavaScript
  - Wird durch Abschicken des Formulars aufgerufen

Formularelement: <input type="text"...>

- Definiert ein einzeiliges Eingabefeld
  - Dient zur Aufnahme einer Zeichenkette (Wörter, Zahlen)
  - Standalone-Tag (d. h. kein Ende-Tag erforderlich)
- Beispiel:
- `<input name="str" type="text" size="30" maxlength="40">`

Bezeichner bzw. Variablenname unter welchem auf Eingabe-string zugegriffen wird	Eingabetyp, hier: Textfeld (anstelle z. B. eines Buttons)	Anzeigebreite des Feldes in Zeichen (automatisches Scrolling, falls **`size< maxlength`**)	Maximale Anzahl einzugebender Zeichen

- Vorbelegungstext kann mit **`value="..."`** eingefügt werden

Formularelement: <textarea ...>

- Definiert ein mehrzeiliges Eingabefeld
  - Dient zur Aufnahme von längeren Textabschnitten
  - Erfordert abschließendes **</textarea>**-Tag
    - Ermöglicht Eingabe von Vorbelegungstext
- Beispiel:

```
<textarea name="textfeld" cols="50" rows="20"></textarea>
```

Bezeichner bzw. Variablenname unter welchem auf Eingabetext zugegriffen wird

Zeichen pro Zeile

Anzahl der angezeigten Zeilen

Formularelement: <input type="radio"...>

- Zur Definition von Radio Buttons
  - Beschriftete Knöpfe, von denen einer ausgewählt sein kann
- Beispiel:

```
<input name="Card" type="radio" value="Mastercard">Mastercard
<input name="Card" type="radio" value="Eurocard">Eurocard
<input name="Card" type="radio" value="Visa" checked>Visa
```

Bezeichner: Alle Radio Buttons mit dem gleichen Bezeichner gehören zur selben Gruppe

Typ: Radio Button

Zu übertragener Wert: Definiert selektierten Knopf

`checked`: Vorselektierter Knopf

Formularelement: <input type="submit"...>

- Zum Abschicken des ausgefüllten Formulars
- Beispiel:

```
<input type="submit" value="Abschicken">
```

Definiert Absende-knopf

Beschriftung des Knopfes

- Werte aus Eingabefeldern werden aneinandergehängt und an das unter `"action"` angegebene Ziel geschickt:

```
<form action="antwort2.php" method="get">
```

Formularelement: <input type="reset"...>

- Zum Abbrechen der Eingabe
  - Eingegebene Daten werden verworfen und nicht abgeschickt
- Beispiel:

```
<input type="reset" value="Abbrechen">
```

Definiert Knopf zum Abbrechen der Eingabe

Beschriftung des Knopfes

Anfrage mittels Formular

- Anfrage3.html:

```
<html><head><title>...</title></head>
<body>
  <form action="antwort3.php" method="get">
    Klient: <input type="text" name="client"><br>
    Ort: <input type="text" name="ort"><br>
    <input type="submit" value="Anfragen">
  </form>
</body>
</html>
```

- antwort3.php: identisch zu antwort2.php

Kontrollstrukturen

Kontrollstrukturen

- Programmverzweigungen
  - **`if, if-else, elseif`**
  - **`switch`**
- Schleifenkonstrukte
  - **`while`**
  - **`for`**

Programmverzweigungen

if-else

- Erlaubt Verzweigungen im Programmablauf:

```
if ( Bedingung )
  // Anweisung, die ausgeführt werden, wenn
  // die Bedingung erfüllt, d. h. wahr, ist
else
  // Anweisung, die ausgeführt
  // wird, wenn die Bedingung nicht
  // erfüllt, d. h. falsch, ist
```

- Der **`else`**-Zweig ist optional

`if-else` mit Block

- Block: In Klammern eingeschlossene Reihe von Anweisungen

```
if ( Bedingung ) {
  // Anweisungen, die ausgeführt werden, wenn
  // die Bedingung erfüllt, d. h. wahr, ist
}
else {
  // Optionale Anweisungen, die ausgeführt
  // werden, wenn die Bedingung nicht
  // erfüllt, d. h. falsch, ist
}
```

if-else: Beispiele (1)

```
if ($a > $b)
  echo("$a ist größer als $b");

if ($a > $b) {
  echo("$a ist größer als $b");
  $max = $a;
}

if ($a > $b) {
  echo("$a ist größer als $b");
  $max = $a;
}
else {
  echo("$a ist kleiner-gleich $b");
  $max = $b;
}
```

`if-else`: Beispiele (2)

- Abfrage des Browser-Typs durch Auswertung einer vordefinierten Variable

```
if (strstr($_SERVER["HTTP_USER_AGENT"], "Gecko")){
  echo("Sie benutzen Firefox <br>");
}
```

- `strstr()`: In PHP eingebaute Funktion, die nach einem String in einem anderen String sucht
  - Wird String gefunden, gibt die Funktion TRUE zurück, sonst FALSE
- Aber: Chrome schickt z. B. „Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:129.0) Gecko/20100101 Firefox/129.0“
- Heutzutage keine sichere Möglichkeit mehr für die Browser-Unterscheidung

Anfrage mittels URL-Querystring

- Zusammenfassen der Dialogdateien zu einer Datei möglich durch Auswerten der Querystringvariablen
  - Hierfür wird deren Existenz geprüft
  - Dies erfolgt mit Hilfe der PHP-Funktion **isset()**
- anfrage.php:

```
<?php
  if (isset($_GET["name"]))
    echo("Habe verstanden, $_GET[name]!");
  else
    echo("<a href='$_SERVER[PHP_SELF]?name=Browser'>
          Webserver bitte melden! </a>");
?>
```

elseif

- Verbindung von **if** und **else**

```
if ($a > $b) {
    echo("$a ist größer als $b");
} elseif ($a == $b) {
    echo("$a ist gleich $b");
} else {
    echo("$a ist kleiner als $b");
}
```

- Mehrere **elseif** innerhalb derselben **if**-Anweisung möglich
- Die erste **if**- oder **elseif**-Bedingung, die TRUE ist, wird ausgeführt

switch

- Alternative Schreibweise für eine Reihe von **if**-Anweisungen mit dem gleichen Parameter

```
if ($i == 0) {
  echo("i ist gleich 0");
}
elseif ($i == 1) {
  echo("i ist gleich 1");
}
elseif ($i == 2) {
  echo("i ist gleich 2");
}
else {
  echo("Wert unbekannt");
}
```

```
switch ($i) {
  case 0:
    echo("i ist gleich 0");
    break;
  case 1:
    echo("i ist gleich 1");
    break;
  case 2:
    echo("i ist gleich 2");
    break;
  default:
    echo("Wert unbekannt");
}
```

Beispiel: Kreditkartenfirma abfragen

- Schreibweise mit einer Reihe von **`if`**-Anweisungen

```
if ($Card == "Mastercard") {
  echo("Betrag wird von Mastercard abgebucht");
}
elseif ($Card == "Visa") {
  echo("Betrag wird von Visa-Karte abgebucht");
}
elseif ($Card == "Eurocard") {
  echo("Betrag wird von Eurocard abgebucht");
}
else {
  echo("Kreditkartenfirma unbekannt");
}
```

Beispiel: Kreditkartenfirma abfragen

- Schreibweise mit **`switch`**-Anweisung

```
switch ($Card) {
  case "Mastercard":
    echo("Betrag wird von Mastercard abgebucht");
    break;
  case "Visa":
    echo("Betrag wird von Visa-Karte abgebucht");
    break;
  case "Eurocard":
    echo("Betrag wird von Eurocard abgebucht");
    break;
  default:
    echo("Kreditkartenfirma unbekannt");
}
```

Schleifenanweisungen

`while`-Schleife

- Konstrukt für die wiederholte Ausführung einer Anweisung
- Anweisung kann auch ein Block sein
- Grundformen einer **`while`**-Anweisung:

1. **`while (ausdr) Anweisung;`**
2.
```
while (ausdr) {
    //Anweisungen;
}
```

Alternative Syntax zu 2.:

3. **`while (ausdr): Anweisung; ... endwhile;`**

while-Schleife: Beispiele

```
// Beispiel 1
$i = 1;
while ($i <= 10)
  echo($i++); // erst wird $i ausgegeben, dann der
              // Wert erhöht (Post-Inkrement)

// Beispiel 2: Äquivalent zu Beispiel 1
$i = 1;
while ($i <= 10) {
    echo($i);
    $i++;
}
```

while-Schleife: Alternative Syntax

```
// Beispiel 1: Syntax mit geschweiften Klammern
$i = 1;
while ($i <= 10) {
    echo($i);
    $i++;
}
// Beispiel 2: Äquivalent zu Beispiel 1
$i = 1;
while ($i <= 10):
    echo($i);
    $i++;
endwhile;
```

if-Anweisung: Alternative Syntax

```
if ($a == 5):
    echo("a ist gleich 5");
    echo("...");
elseif ($a == 6):
    echo("a ist gleich 6");
    echo("!!!");
else:
    echo("a ist weder 5 noch 6");
endif;
```

Prä- und Post-Inkrement und -Dekrement

- Post-Inkrement: $i++
- Post-Dekrement: $i--
- Prä-Inkrement: ++$i
- Prä-Dekrement: --$i
- Beispiel:

```
$i = 1;
echo($i++); // Ausgabe: ?
echo(++$i); // Ausgabe: ?
echo($i--); // Ausgabe: ?
echo(--$i); // Ausgabe: ?
```

Beispiel für Prä-Inkrement

```
// Beispiel 1
$i = 0;
while ($i < 10)
  echo(++$i); // erst wird der Wert erhöht,
              // dann $i ausgegeben

// Beispiel 2: Äquivalent zu Beispiel 1
$i = 0;
while ($i < 10) {
    $i++;
    echo($i);
}
```

Weitere Abkürzungen

```
$a = 3;
$a += 5;          // setzt $a auf den Wert 8
                  // entspricht: $a = $a + 5;

$b = "Hello ";
$b .= "all";      // setzt $b auf den Wert
                  // "Hello all", äquivalent
                  // zu $b = $b . "all";
```

- Auch: `-=`, `*=`, `/=`, `%=` etc.

do-while-Schleife

- Wie **while**-Schleife
- Wahrheitswert des Ausdrucks wird indes erst am Ende jeder Iteration geprüft
- Schleife wird in jedem Fall einmal durchlaufen
- Grundformen einer **do-while**-Anweisung:

1.
```
do Anweisung while (ausdr);
```

2.
```
do {
    // Anweisungen;
} while (ausdr);
```

do-while-Schleife: Beispiele

```
// Beispiel 1
$i = 1;
do
  echo($i++); // erst wird $i ausgegeben, dann der
              // Wert erhöht (Post-Inkrement)
while ($i <= 10);

// Beispiel 2: Äquivalent zu Beispiel 1
$i = 1;
do {
    echo($i);
    $i++;
} while ($i <= 10);
```

for-Schleife

1. `for (Initial.; Bedingung; Aktualisier.) Anweisung`
2. `for (Initial.; Bedingung; Aktualisier.) {`

```
    // Anweisungen;
  }
```

Initialisierung:	**Laufvariable initialisieren**
Bedingung:	**Prüfen, ob Laufvariable im zulässigen Bereich**
Aktualisierung:	**Wert der Laufvariable erhöhen oder verringern**

for-Schleife: Beispiele

```
for ($i = 1; $i <= 10; $i++) echo($i);
```

```
for ($i = 1; $i <= 10; $i++) {
  echo($i);
}
```

Alternative Syntax:

```
for ($i = 1; $i <= 10; $i++):
  echo($i);
endfor;
```

break

- **break** erlaubt den vorzeitigen Abbruch einer **for, while, do..while, foreach** oder **switch** Anweisung

- Beispiel:

```
1. for ($i = 1; $i <= 10; $i++) echo($i);

2. for ($i = 1;;$i++) {
       if ($i > 10)
          break;
       echo($i);
   }
```

- Nur bei **switch** sinnvoll, sonst führt es zu unsauberer Programmierung

continue

- Abbruch des gegenwärtigen Schleifendurchlaufs
- Beispiel:

```
$var = 0;
while ($var++ < 11) {
    if ($var % 2) // überspringe ungerade
      continue;   // Werte
    echo("$var ");
}
```

- Dito: Unsaubere Programmierung. Besser if-Anweisung im Schleifenrumpf oder Bedingung im Schleifenkopf anpassen

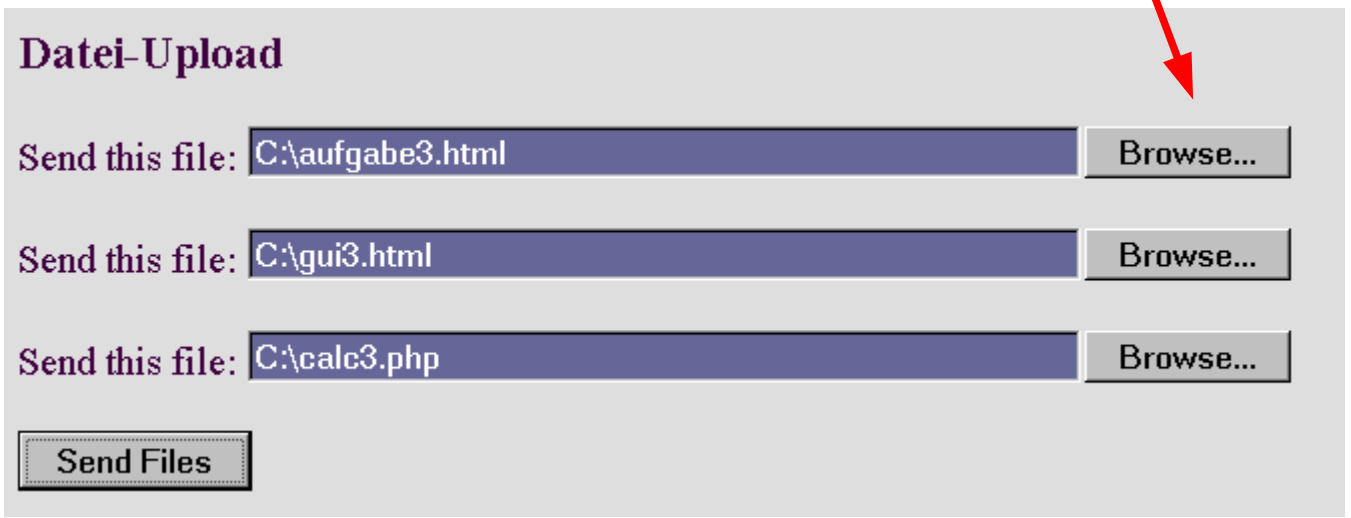
Dateien auf den Webserver hochladen

Datei-Upload

- Realisiert mittels:
  - Eines HTML-Formulars zum Eingeben von Dateinamen
  - Einer PHP-Datei zum Verschieben der übertragenen Dateien in das Zielverzeichnis

Prinzipieller Ablauf (1)

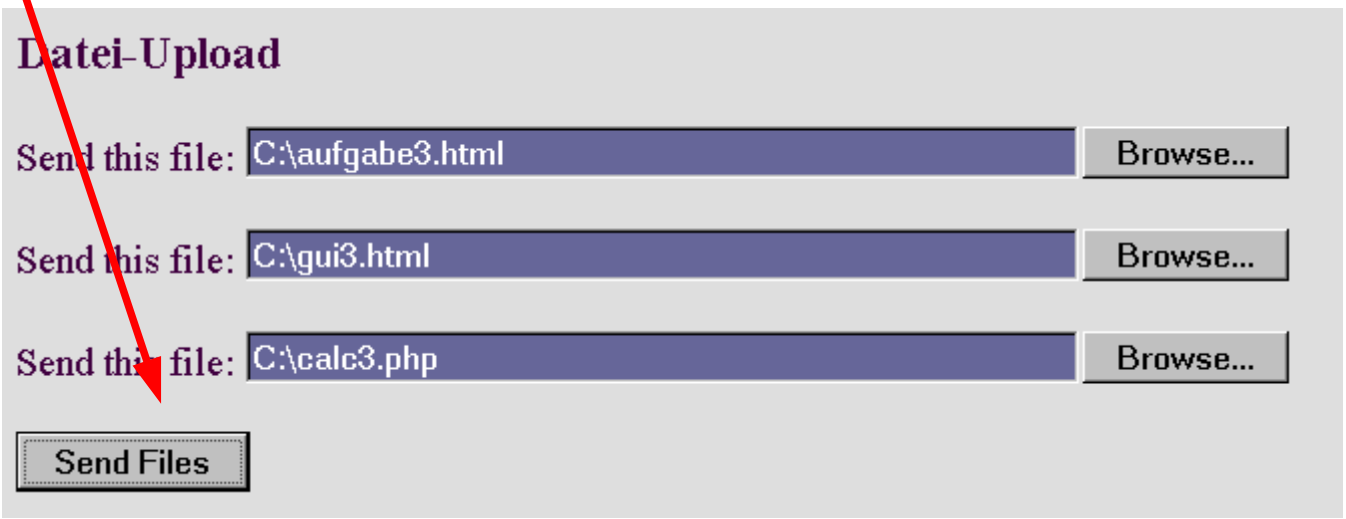
- Benutzer wählt Dateien im Webbrowser mittels HTML-Formular
- HTML-Formular enthält Element für die Dateiauswahl:
  - Eingabefeld plus Auswahl-Button
- `<p>Send this file: <input name="userfile" type="file"></p>`



Prinzipieller Ablauf (2)

- Nach Auswahl der Datei wird Formular mit Send-Button abgeschickt:

```
<input type="submit" value="Send Files"></form>
```



- Mit einem einfachen Formular wird nur der *Name* der gewählten Dateien übertragen!

Prinzipieller Ablauf (3)

- Damit nicht nur die Dateinamen, sondern auch die zugehörenden Dateien übertragen werden, muss das Formular aus mehreren Teilen bestehend definiert werden:

```
<form enctype="multipart/form-data"
action="upload.php" method="post">
```

- Zum Vergleich: Einfaches Formular

```
<form action="form.php" method="get">
```

HTML-Formular

```
<form enctype="multipart/form-data"
  action="upload.php"
  method="post">
```

- Formular für Datei-Upload

```
<input type="hidden"
  name="MAX_FILE_SIZE"
  value="1000000">
```

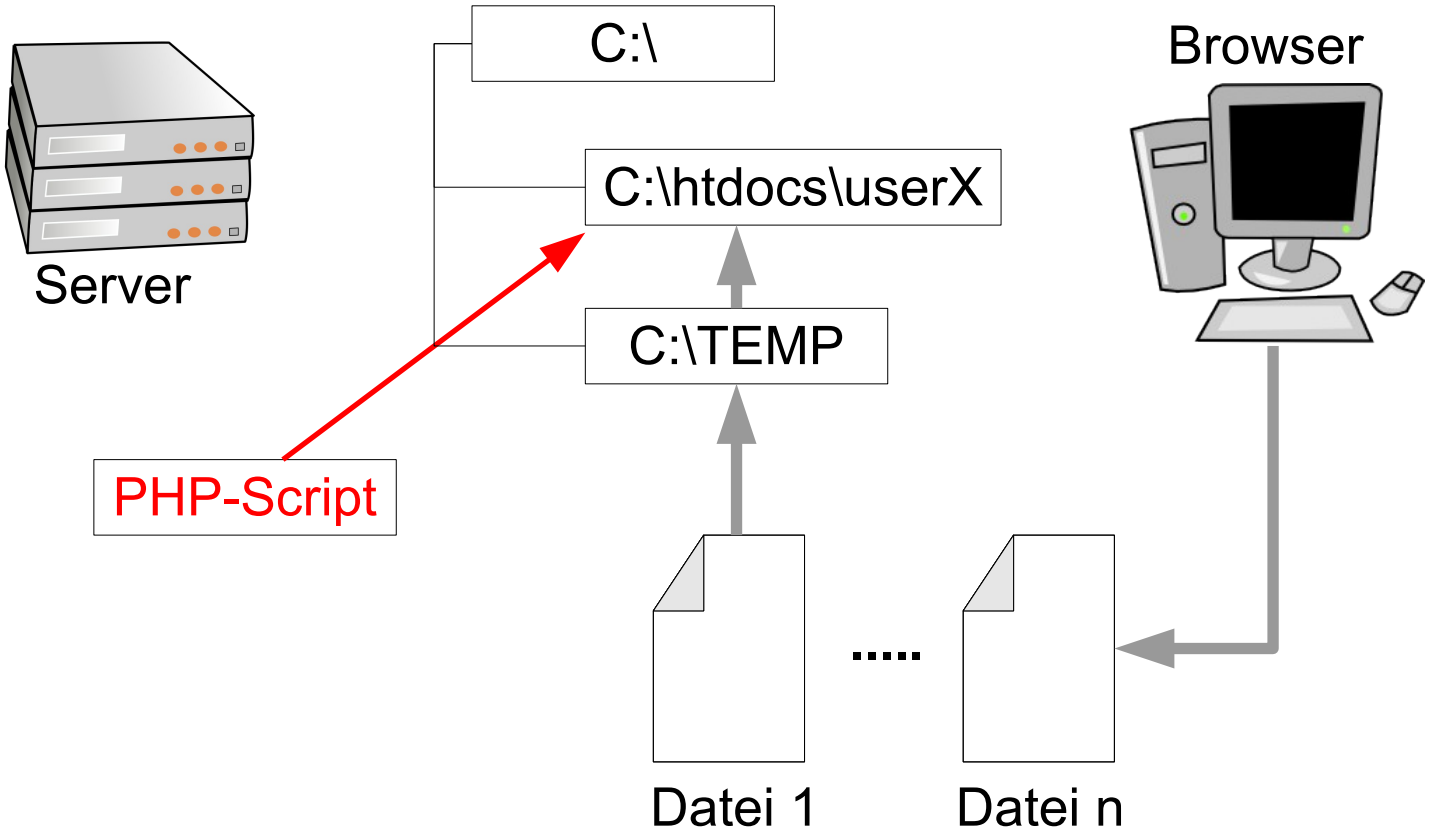
- Versteckes Element: Legt erlaubte Maximalgröße in Bytes der zu übertragenen Dateien fest

```
<p>Send this file: <input
  name="userfile" type="file"
  size="50"></p>

<input type="submit"
  value="Send File"></form>
```

- Definiert Element für Datei-Upload
- Webbrowser zeigt Eingabefeld an, das die Eingabe eines Dateinamens erlaubt

Prinzipieller Ablauf (4)

- Dateien werden nicht direkt in Variablen des PHP-Interpreters übertragen, sondern in das temporäre Verzeichnis des Webservers gelegt (z. B. C:\temp oder C:\xampp\tmp)!



Prinzipieller Ablauf (5)

- Das PHP-Script, welches die Formulardaten verarbeiten soll, wie üblich im einleitenden **<form>**-Tag eintragen:

```
<form enctype="multipart/form-data"
action="upload.php" method="post">
```

- Die Aufgabe des PHP-Scripts besteht darin, die übertragenen Dateien aus dem temporären Verzeichnis in das Zielverzeichnis zu verschieben

- Zu diesem Zweck kann es über den im jeweiligen **<input>**-Tag definierten Namen auf die temporäre Datei zugreifen:

```
<p>Send this file: <input name="userfile"
type="file" size="50"></p>
```

PHP-Script (1)

- Temporärer Name der Datei inkl. Verzeichnis steht in der Variable

  ```
  $_FILES["userfile"]["tmp_name"]
  ```

  - Name, unter dem Datei im temporärem Verzeichnis liegt
- Wurzelverzeichnis des HTML-Servers steht in Variable

  ```
  $_SERVER["DOCUMENT_ROOT"]
  ```

- Ursprünglicher Dateiname steht in:

  ```
  $_FILES["userfile"]["name"]
  ```

PHP-Script (2)

- Größe der hochgeladenen Datei in Bytes

```
$_FILES["userfile"]["size"]
```

  - Kann benutzt werden, um zu überprüfen, ob die Datei die vorgeschriebene Maximalgröße auch nicht überschreitet

- Der MIME-Type der Datei, z. B. "image/gif"

```
$_FILES["userfile"]["type"]
```

  - Kann benutzt werden, um zu überprüfen, ob die hochgeladene Datei dem erwarteten Dateityp entspricht

PHP-Script (3)

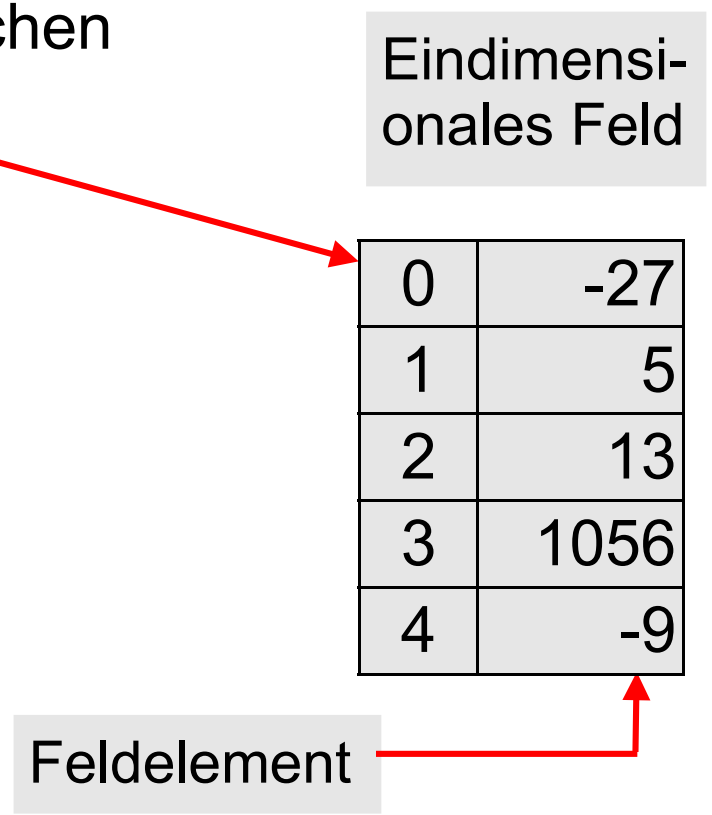
```
$dest = $_SERVER["DOCUMENT_ROOT"] . "/" .
        $_FILES["userfile"]["name"];

if (move_uploaded_file(
      $_FILES["userfile"]["tmp_name"], $dest)
   )
  echo("<br>File successfully saved to: $dest");
else
  echo("<br>Error during uploading");
```

- **move_uploaded_file**: PHP-Funktion zum Verschieben einer hochgeladenen Datei in ein Zielverzeichnis

Felder

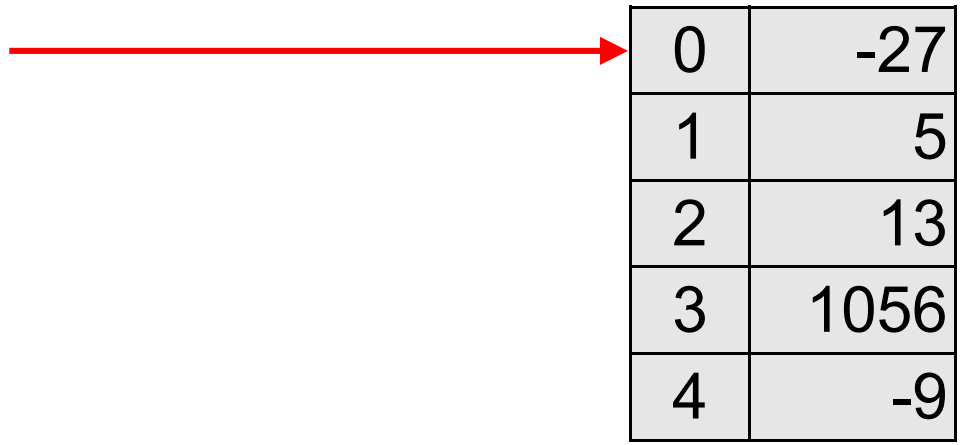
Felder

- (engl.: *array*)
- Datentyp, welcher die Speicherung einer fortlaufenden Liste von Werten erlaubt
- Werte werden über *Schlüssel* angesprochen
- Ein- oder mehrdimensional
- In PHP gibt es zwei Arten von Feldern:
  - Indizierte Felder
  - Assoziative Felder



Indizierte Felder

- Bei diesem Feldtyp werden die Feldelemente wie in üblichen Programmiersprachen über nicht-negative ganze Zahlen angesprochen
- Schlüssel sind daher *Indizes*



0	-27
1	5
2	13
3	1056
4	-9

Indizierte Felder: Beispiele

- Feld wird durch explizites Zuweisen von beliebig vielen Werten erzeugt

```
$data[0] = -27;
$data[1] =  5;
$data[2] =  13;
$data[3] =  1056;
$data[4] = -9;
```

```
$data[] = -27;
$data[] =  5;
$data[] =  13;
$data[] =  1056;
$data[] = -9;
```

```
for ($i=0; $i<5; $i++)
    echo("data[$i]: $data[$i] <br>");
```

Indizierte Felder: Zeichenketten

```
$farbe[0] = "rot";
$farbe[1] = "grün";
$farbe[2] = "blau";

for ($i=0; $i<3; $i++)
    echo("farbe[$i]: $farbe[$i] <br>");
```

array()

- Alternative Funktion für die Erzeugung von Feldern

```
$data[0] = -27;
$data[1] =  5;
$data[2] =  13;
$data[3] =  1056;
$data[4] = -9;
```

```
$data = array(-27,5,13,1056,-9);
```

```
$farbe[0] = "rot";
$farbe[1] = "grün";
$farbe[2] = "blau";
```

```
$farbe = array("rot","grün","blau");
```

Assoziative Felder (1)

- Schlüssel sind *Zeichenketten*

Nr. 1	-27
Nr. 2	5
Nr. 3	13
Nr. 4	1056
Nr. 5	-9

```
$data["Nr. 1"] = -27;
$data["Nr. 2"] =  5;
$data["Nr. 3"] =  13;
$data["Nr. 4"] =  1056;
$data["Nr. 5"] = -9;

for ($i=1; $i<6; $i++){
  $index = "Nr. " . $i;
  echo("data[$index]: $data[$index] <br>");
}
```

Ausgabe:

```
data[Nr. 1]: -27
data[Nr. 2]: 5
data[Nr. 3]: 13
data[Nr. 4]: 1056
data[Nr. 5]: -9
```

Assoziative Felder (2)

```
$farbe["rot"]  = "255 0 0";
$farbe["grün"] = "0 255 0";
$farbe["blau"] = "0 0 255";
```

Schlüssel funktionieren bei der Ausgabe auch ohne Anführungszeichen

```
echo("farbe[rot]: $farbe[rot] <br>");
echo("farbe[grün]: $farbe[grün] <br>");
echo("farbe[blau]: $farbe[blau] <br>");
```

Mehrdimensionale indizierte Felder

- Zusätzliche Dimensionen werden einfach durch Hinzufügen weiterer Schlüssel erzeugt

```
$farbe[0][0]   = 255;
$farbe[0][1]   = 0;
$farbe[0][2]   = 0;

$farbe[1][0]   = 0;
$farbe[1][1]   = 255;
$farbe[1][2]   = 0;

$farbe[2][0]   = 0;
$farbe[2][1]   = 0;
$farbe[2][2]   = 255;
```

```
echo("farbe['rot']: "   .
     $farbe[0][0] . " " .
     $farbe[0][1] . " " .
     $farbe[0][2] . "<br>");

// Verkürzte Ausgabe nur mit
// geschweiften Klammern:

echo("farbe['rot']:
  {$farbe[0][0]} {$farbe[0]
  [1]} {$farbe[0][2]} <br>");

// 0: rot, 1: grün, 2: blau
```

`array()` und mehrdimensionale Felder (1)

- Zuweisung des Teilfeldes erfolgt mit Hilfe des "**=>**"-Operators
- Indiziertes Feld:

```
$farbe = array ( 0 => array(255,0,0),
                 1 => array(0,255,0),
                 2 => array(0,0,255));
```

`array()` und mehrdimensionale Felder (2)

- Auch Mischformen sind möglich
- Indiziert-assoziatives Feld:

```
$farbe = array ("rot"  => array(255,0,0),
                "grün" => array(0,255,0),
                "blau" => array(0,0,255));

echo("farbe['rot']: "   .
     $farbe["rot"][0] . " " .
     $farbe["rot"][1] . " " .
     $farbe["rot"][2] . "<br>");
//etc.
```

`array()` und mehrdimensionale Felder (3)

- Rein assoziatives Feld:

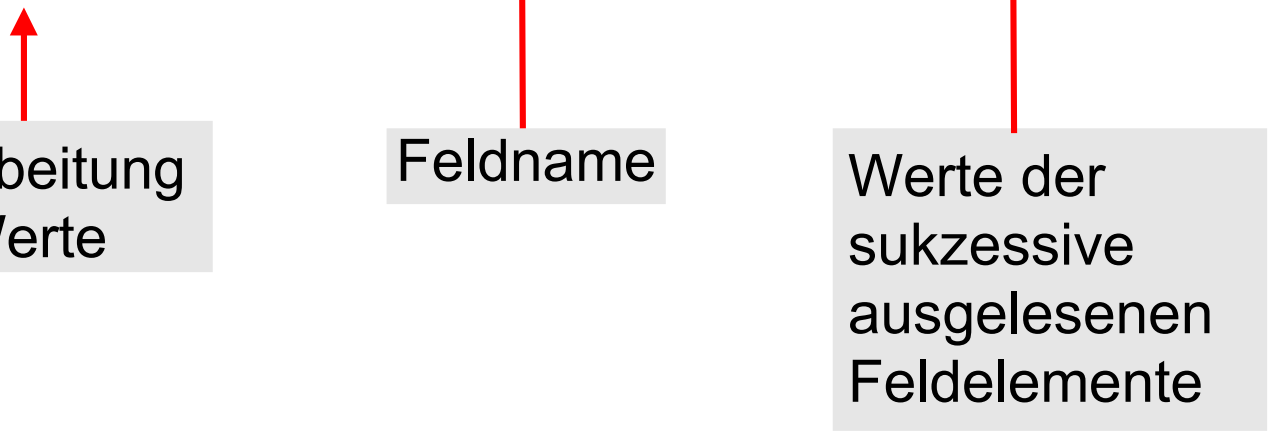
```
$farbe = array ( "rot"  => array("r" => 255,
                                 "g" => 0,
                                 "b" => 0
                                 )
                        //, etc.
);
echo("farbe['rot']: "   .
     $farbe["rot"]["r"] . " " .
     $farbe["rot"]["g"] . " " .
     $farbe["rot"]["b"] . "<br>");
//etc.
```

`foreach`-Kontrollstruktur

foreach-Kontrollstruktur (1)

- Ermöglicht es, die Elemente eines Feldes in sehr einfacher Weise zu durchlaufen
- Syntax:

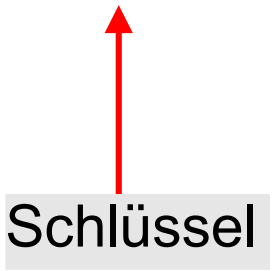
```
foreach(array_expression as $value)
   Anweisung
```



foreach-Kontrollstruktur (2)

- Soll während des Durchlaufens der Feldelemente ebenfalls der Schlüssel ausgelesen werden, so ist dies über eine erweiterte Syntax mit "=>"-Operator möglich:

```
foreach(array_expression as $key => $value)
   Anweisung
```



foreach-Kontrollstruktur (3)

```
$a = array (1, 2, 3, 17);
```

- Mittels **foreach** kann diese **for**-Schleife

```
for($i=0; $i<count($a); $i++) {
    echo("a[$i]: $a[$i]<br>");
    $sum += $a[$i];
} // count() liefert die Anzahl der Feldelemente
```

- wie folgt geschrieben werden:

```
foreach ($a as $key => $val) {
    echo("a[$key]: $val<br>");
    $sum += $val;
}
```

- Ausgabe des Schlüssels mittels Erweiterung **'$key =>'**

foreach-Kontrollstruktur (4)

- **foreach** insbesondere nützlich bei assoziativen Feldern:

```
$farbe["rot"]  = "255 0 0";
$farbe["grün"] = "0 255 0";
$farbe["blau"] = "0 0 255";

echo("farbe[rot]:  $farbe[rot] <br>");
echo("farbe[grün]: $farbe[grün] <br>");
echo("farbe[blau]: $farbe[blau] <br>");

// echo()-Anweisungsfolge kann abgekürzt werden zu:

foreach ($farbe as $key => $val)
  echo("farbe[$key]: $val <br>");
```

- Auch hier Ausgabe des Schlüssels mittels Erweiterung '**$key =>**'

`foreach` und mehrdimensionale Felder

```
$farbe = array ("rot"   => array(255,0,0),
                "grün"  => array(0,255,0),
                "blau"  => array(0,0,255));

echo("<br>farbe[rot]: ");
for ($i=0; $i<3; $i++) echo($farbe["rot"][$i] . " ");
echo("<br>farbe[grün]: ");
for ($i=0; $i<3; $i++) echo($farbe["grün"][$i] . " ");
echo("<br>farbe[blau]: ");
for ($i=0; $i<3; $i++) echo($farbe["blau"][$i] . " ");
```

- **Anweisungsfolge kann ersetzt werden durch:**

```
foreach ($farbe as $key1 => $val1)
  foreach ($val1 as $key2 => $val2)
      echo("farbe[$key1][$key2]: $val2<br>");
```

+ Operator

- Ermöglicht Konkatenation von Feldern
- Beispiel:

```
$a = array( "a" => "Apfel", "b" => "Banane");
$b = array( "a" => "Birne", "b" => "Erdbeere",
            "c" => "Kirsche");
$c = $a + $b;
```

- Liefert mit:

```
foreach ($c as $key => $val)
  echo("c[$key]: $val, ");
```

- folgendes Ergebnis:

```
c[a]: Apfel, c[b]: Banane, c[c]: Kirsche,
```

- Doppelte Schlüssel werden nicht überschrieben!

Zeichenketten

Zugriff auf Zeichen

- Auf die Zeichen einer Zeichenkette (engl.: *string*) kann mittels Schlüssel in eckigen Klammern zugegriffen werden
  - Erstes Zeichen einer Zeichenkette besitzt den Schlüssel 0
- Beispiel:

```
$str = "Dies ist ein String.";

$first = $str[0];           // $first enthält "D"

$second = $str[1];          // $second enthält "i"

$last = $str[strlen($str)-1]; // $last enthält "."
```

- Die Länge einer Zeichenkette ist mit `int strlen(string str)` ermittelbar

Umwandlung von Zeichenketten

- Eine Zeichenkette wird in einem Ausdruck als Zahlenwert angesehen, sofern diese mit gültigen numerischen Daten beginnt
  - Automatische Typ-Konvertierung
- Die Zeichenkette wird als Fließkommazahl ausgewertet, wenn sie eines der Zeichen
  "`.`", "`e`" oder "`E`" enthält
- Alles andere wird als Integer-Wert interpretiert
- Nur die erste zusammenhängende Zahl in einer Zeichenkette wird ausgewertet!
- Mehr als eine Zahl oder zusätzliche Zeichen in einer Zeichenkette vermeiden: Führt zu Warning-Ausgabe in das Browser-Fenster

Umwandlung von Zeichenketten: Beispiele

```
$res = 1 + "10.5";           // $res: float (11.5)
$res = 1 + "-1.3e3";         // $res: float (-1299)
$res = 1 + "10 Tomaten";     // $res: int (11) + Warning
$res = 4 + "10.2kg Zucker";// $res: float (14.2) + Wrg.
$res = "10 100" + 1;         // $res: int (11) + Warning
$res = "10,4ms" + 1;         // $res: int (11) + Warning
$res = "10.0ms" + 1.0;       // $res: float (11) + Warng.
```

Suchen in Zeichenketten

`string strstr(string haystack, string needle)`

- Sucht erstes Vorkommen der Zeichenkette **`needle`** in **`haystack`**
- Gibt alles von **`haystack`** zurück ab der ersten Fundstelle von **`needle`** bis zum Ende
- Falls **`needle`** nicht gefunden wird, ist das Ergebnis FALSE
- Funktion unterscheidet zwischen Groß- und Kleinschreibung
  - Falls keine Unterscheidung gewünscht: **`stristr()`**

Suchen in Zeichenketten: Beispiele

```
$email = "KlausBrause@wasserundsaft-gmbh.de";
$domain = strstr($email, "@");
echo($domain); // gibt @wasserundsaft-gmbh.de zurück


$opStr = "12+3";

if (strstr($opStr,"+")) {
  echo("+ gefunden<br>");
  $Operation = "Addition";
}
```

Strings modifizieren

- Beispiel:

```
$strA = "string A";
$strB = str_replace('A', 'B', $strA);

echo($strA . "<br>");
echo($strB . "<br>");

// Ausgabe:
string A
string B
```

- Ersetzt alle Vorkommen der gesuchten Zeichenkette

Zerlegen einer Zeichenkette

- Zerteilen einer Zeichenkette an Hand einer Trennzeichenkette:

```
array explode(string separator, string input [, int limit])
```

  - **`separator`**: Zeichenkette **`input`** wird an den durch die Zeichenkette **`separator`** bezeichneten Positionen aufgetrennt
  - Zurückgegeben werden die "Bruchstücke" ohne die Zeichenkette **`separator`** als Feld von Zeichenketten
  - **`limit`**: Maximal **`limit`** Teile werden zurückgegeben

explode(): Beispiel

```
$opStr = "12+3";

$pieces = explode("+", $opStr);  // Verwende Plus-
                                 // Zeichen
                                 // für die Trennung
foreach ($pieces as $v) {
    echo("Substring: $v<br>");
}

/*
liefert:
Substring: 12
Substring: 3
*/
```

Array- und String-Funktionen

- Viele weitere vordefinierte PHP-Funktionen für die Verarbeitung von Feldern und Zeichenketten im:

  PHP Handbuch, PHP-Dokumentationsgruppe

  - https://www.php.net/manual/de/book.strings.php

Funktionen

Funktionen

- PHP bietet eine Vielzahl vordefinierter Funktionen
- Neben diesen können auch eigene Funktionen definiert werden
- Syntax:

```
function myFunction($par1, $par2,..., $parn) {
  // Funktionsrumpf
}
```

- Daten werden an die Funktion mittels einer *Parameterliste* übergeben
  - Die Parameterliste ist eine durch Kommata getrennte Liste von Variablen und/oder Konstanten
  - Parameter werden auch als *Argumente* bezeichnet
  - Funktionswert wird mit `return` Befehl zurückgegeben

Funktion: Beispiel

```
<?php
function parSum($par1, $par2, $par3) {
    echo("Berechnung<br>");
    $retval = $par1+$par2+$par3;
    return $retval;
}
echo("Argumentsumme: " . parSum(1,2,3) . "<br>");
?>
```

Gibt aus:

```
Berechnung
Argumentsumme: 6
```

Hier keine verkürzte Schreibweise als einzelne Zeichenkette ohne . möglich!

Vorgabewerte

- Funktionsparameter können Vorgabewerte zugewiesen werden
  - Die Ersetzung der Vorgabe- durch Parameterwerte beim Aufruf erfolgt dabei von links nach rechts

```
<?php
function parSum($par1=1, $par2=2, $par3=3) {
    $retval = $par1+$par2+$par3;
    return $retval;
}
echo("Sum1: " . parSum(2,4,6) . "<br>"); // = 12
echo("Sum2: " . parSum()      . "<br>"); // =  6
echo("Sum3: " . parSum(10)    . "<br>"); // = 15
echo("Sum4: " . parSum(10,20) . "<br>"); // = 33
?>
```

Variable Anzahl von Parameter

- Funktionen können eine variable Anzahl von Parameter besitzen
- Hierfür ist keine spezielle Syntax erforderlich
- Zugriff auf Parameter mittels spezieller Funktionen:
  - `func_num_args()`
    - Gibt die Anzahl der übergebenen Parameterwerte zurück
  - `func_get_arg(n)`
    - Gibt den *n*-ten Parameterwert zurück
  - `func_get_args()`
    - Gibt ein Feld (Array) der übergebenen Parameterwerte zurück

Variable Anzahl von Parameter: Beispiel

```
<?php
function parSum2() {
  $numpar = func_num_args();
  echo("Anzahl der Parameter: $numpar<br>");
  if ($numpar >= 1) {
     echo("1. Parameterwert: " . func_get_arg(0) . "<br>");
  }

  $par_list = func_get_args();
  for ($i = 0; $i < count($par_list); $i++) {
     echo("Parameterwert $i ist: $par_list[$i] <br>");
  }
}
parSum2(1, 2, 3);
?>
```

```
Anzahl der Parameter: 3
1. Parameterwert: 1
Parameterwert 0 ist: 1
Parameterwert 1 ist: 2
Parameterwert 2 ist: 3
```

Referenzparameter

- Normalerweise werden Werte an Funktionen übergeben
  - *Call by value*
  - Intern an die Parameter zugewiesene Werte verfallen
  - Nur ein Rückgabewert möglich
    - Dieser wird mit `return()` zurückgegeben
- PHP erlaubt auch **Referenzparameter**
  - *Call by reference*
  - An Referenzparameter zugewiesene Werte bleiben erhalten
  - Referenzparameter werden durch ein vorangestelltes **&** deklariert

Referenzparameter: Beispiel

```
<?php
function anfrage(&$string) {
    $string = "Was gibt's denn?<br>";
}
$str = "Ist da wer?<br>";
echo($str);
anfrage($str);
echo($str);
?>
```

Ausgabe:

```
Ist da wer?
Was gibt's denn?
```

Funktionen innerhalb von Funktionen

- Funktionsrümpfe können Definitionen von Funktionen enthalten

```
function f1() {
  function f2() {
    echo("Ich existiere erst nach Aufruf von f1()");
  }
}

// f2() noch nicht aufrufbar, da nicht existent
f1();

// Jetzt existiert auch f2()
f2();
```

Bedingte Funktionsdefinition

```
<?php
$makefunc = true;
f3();

if ($makefunc) {
  function condFunc () {
    echo("Ich existiere erst, wenn mich die
          Programmausführung erreicht hat.<br>");
  }
}

if ($makefunc) condFunc();
function f3() {
  echo "Ich existiere sofort nach Programmstart.<br>";
}
?>
```

Variablenfunktionen

- Variablenwert wird als Funktionsname interpretiert, wenn Klammern an den Variablennamen angehängt werden
- Funktioniert nicht mit vordefinierten Funktionen wie z. B. **`print()`**, **`unset()`**, **`isset()`**, **`empty()`**, **`include()`**, **`require()`** und dem Befehl **`echo`**
  - Hier Hilfsfunktion erforderlich
  - Beispielsweise eine Funktion **`_echo()`**, die **`echo`** aufruft (s. u.)

Variablenfunktionen: Beispiel

```
<?php
function func1() {
    echo("In func1()<br>");
}

function func2($par = "Ersatzwert") {
    echo("In func2(): Parameterwert = $par.<br>");
}

$func = "func1";
$func();

$func = "func2";
$func("Test");
?>
```

Ausgabe:

```
In func1()
In func2(): Parameterwert = Test.
```

Hilfsfunktion: Beispiel

```
<?php

// Dies ist eine Hilfsfunktion für den Aufruf
// von echo

 function _echo($string) {
     echo($string);
 }


 $func = "_echo";
 $func("echo auf _echo abgebildet.<br>");


 ?>
```

Geltungsbereiche

Geltungsbereiche

- Konstanten
  - Beispiel:

```
define("MYCONSTANT", "Hallo Welt.");
echo(MYCONSTANT);  // Ausgabe: "Hallo Welt."
```

  - Geltungsbereich ist global
  - Auf eine Konstante kann überall im Script zugegriffen werden
- Variablen
  - Beschränkt auf den Kontext, in welchem die Variable definiert wurde
  - Z. B. Rumpf einer Funktion

Variablen und Funktionen

- Jede in einer Funktion benutzte Variable ist auf den Rumpf der Funktion beschränkt:

```
$a = 1; // Globaler Geltungsbereich
function test() {
    echo($a); // Lokaler Geltungsbereich!
}
test();
```

- Echo-Anweisung bezieht sich auf lokale Variable **`$a`**
- Folge: Skript gibt Warnung aus („Warning: Undefined variable $a in ... on line ...“), da **`$a`** im Funktionsrumpf kein Wert zugewiesen wird

Globale Variablen

- Um globale Variablen in Funktionsrümpfen benutzen zu können, müssen diese explizit als global definiert werden
- Dies erfolgt mit dem Attribut **`global`**
- Beispiel:

```
$a = 1;
$b = 2;
function sum() {
  global $a, $b;
  $b = $a + $b;
}
sum();
echo($b); // Gibt "3" aus
```

Statische Variablen

- Der Wert einer statischen Variable bleibt zwischen den Funktionsaufrufen erhalten
- Attribut: **`static`**
- Beispiel:

```
function counter() {
   static $a = 0;
   echo("Aufruf: $a <br>");
   $a++;
}
for ($i=0; $i<3; $i++) counter();
```

Ausgabe:

```
Aufruf: 0
Aufruf: 1
Aufruf: 2
```

Vordefinierte Funktionen

- PHP stellt hunderte vordefinierter Funktionen für die verschiedensten Anwendungsbereiche zur Verfügung
  - Unterstützung für Strings, Streams, Kompression, Datenbankzugriffe, Flash, PDF, Datum und Zeit, Mathematik, Mail, Grafik...
- Siehe Funktionsreferenz im "PHP Handbuch"

Beispiel: Datums-Funktion

Beispiel: **date.php**

```
<html>
<body>
<p>Das heutige Datum ist:
<?php echo(date("l, F dS Y."));?>
</p>
</body>
</html>
```

- l – (kleines 'L') ausgeschriebener Tag der Woche, z. B. "Friday"
- F - Monat als ganzes Wort, z. B. "January"
- d – Tag des Monats, 2-stellig mit führender Null: "01" bis "31"
- S - Anhang der englischen Aufzählung, z. B. "st", "nd", "rd" oder "th"
- Y - Jahr als vierstellige Zahl, z.B. "1999"

Dateioperationen

`include()` und `require()` (1)

- Die **`include()`**- und **`require()`**-Anweisungen binden eine externe Datei in ein PHP-Script ein
  - Diese wird dann zusammen mit dem Script verarbeitet
- Die Anweisungen sind funktional identisch, Unterschiede bestehen nur im Fehlerfall, falls die externe Datei nicht gefunden wird
- Fehlerbehandlung:
  - **`include()`** erzeugt Warnung
    - PHP-Datei wird weiter abgearbeitet
  - **`require()`** erzeugt Abbruch
    - Bearbeitung der PHP-Datei wird abgebrochen

`include()` und `require()` (2)

- Beim Einbinden einer Datei wechselt der Parser vom PHP-Modus in den HTML-Modus
- Am Ende der Datei kehrt er wieder in den PHP-Modus zurück
- Deshalb muss PHP-Code innerhalb der eingebundenen Datei von PHP-Start- und Ende-Tags eingefasst sein

include() und require(): Beispiel

vars.php:

```
<?php
$color = "green";
$fruit = "apple";
?>
```

test.php:

```
<?php
echo("A $color $fruit");

include("vars.php");
echo("A $color $fruit");
?>
```

- Ausgabe:
- **A**
  - Zusätzlich Warnung, da Variablen nicht bekannt
- **A green apple**
  - Keine Warnung, da Variablen jetzt bekannt

Dateien lesen und schreiben

- Nützlich, um z. B. externe Informationsquellen auszulesen oder flüchtige Daten abzuspeichern
- Schritt 1: Öffnen der Datei

```
$datei = fopen("Dateiname","Zugriffmodus");
```

- Zugriffmodi:
  - "**r**": **Lesen**. Begonnen wird am Dateianfang.
  - "**w**": **Schreiben**. Existiert die Datei bereits, wird der Inhalt gelöscht. Existiert die Datei nicht, wird sie erzeugt.
  - "**a**": **Anfügen**. Begonnen wird am Ende der Datei. Existiert die Datei nicht, wird sie erzeugt.

Dateien gleichzeitig lesen und schreiben

- "+"-Zeichen an den Zugriffsmodus anhängen
  - "`w+`": **Schreiben und lesen**. Existiert die Datei bereits, wird der Inhalt gelöscht. Existiert die Datei nicht, wird sie erzeugt. Dateizeiger wird auf den Anfang der Datei gesetzt.
  - "`a+`": **Anfügen und lesen**. Neue Zeichen werden am Ende der Datei eingefügt. Existiert die Datei nicht, wird sie erzeugt. Lesen beginnt am Anfang der Datei.
  - "`r+`": **Lesen und anfügen**. Wie "a+", allerdings wird Datei nicht automatisch erzeugt, falls sie nicht existiert.

Lesen aus einer Datei

- Beispiel: Inhalt einer Datei zeilenweise lesen

```
while (!feof($datei)) {
  $zeile = fgets($datei);
  echo($zeile);
}
```

- Es wird solange gelesen, bis Ende der Datei erreicht ist
  - **`feof($datei)`** liefert TRUE, sobald Dateiende erreicht ist, sonst FALSE
- **`$zeile = fgets($datei)`** liest eine Zeile aus Datei
  - Terminiert, sobald eine neue Zeile beginnt, 1024 Bytes gelesen wurden oder das Ende der Datei erreicht ist

Schreiben in eine Datei

- **`fwrite ($datei, $str)`**
  - Schreibt den Inhalt der Zeichenkette **`$str`** in die Datei auf die **`$datei`** zeigt

- Beispiel:

```
$str = "Add this to the file\n";

$datei = fopen("test.txt","w");

fwrite($datei, $str);

fclose($datei);
```

Schließen der Datei

- Nach Abschluss der Lese- und/oder Schreiboperation muss die Datei geschlossen werden:

```
fclose($datei);
```

- Mehr über Dateioperationen im PHP Handbuch
  - Siehe: *Dateisystem*
    - https://www.php.net/manual/de/book.filesystem.php

Unterdrücken unerwünschter Fehlermeldungen

- Vordefinierte Funktionen produzieren im Fehlerfall ihre eigenen Fehlermeldungen
- Dies kann durch ein vorangestelltes @ unterbunden werden

```
$datei = @fopen("test.txt","r");
if ($datei != NULL) {
  while (!feof($datei)) {
    $zeile = fgets($datei);
    echo($zeile);
  }
  fclose($datei);
}
```

- Ohne @-Zeichen gibt **`fopen`** eine Warnung in das Browserfenster aus, wenn die Datei nicht existiert

Eigene Fehlerbehandlung

- Fehler bei Dateizugriff können im Script abgefangen werden
  - Dateioperationen liefern **`NULL`** oder **`FALSE`** im Fehlerfall
  - Auf diese Weise lassen sich die vorgegebenen Fehlermeldungen durch eine eigene, situationsabhängige Fehlerbehandlung ersetzen
  - Stichwort: Graceful Degradation

Objektorientierte Programmierung in PHP

Beispiel: Lightbulb-Klasse

- Modelliert werden soll eine Glühbirne
- Wir sind nicht am physischen Aufbau der Glühbirne interessiert
- Wichtig hinsichtlich der Modellierung soll nur der Zustand der Glühbirne sein
  - Ein- bzw. ausgeschaltet
- Die Glühbirne wird also sehr abstrakt modelliert
- Weiterhin möchten wir mittels Methoden die Glühbirne ein- und ausschalten und ihren Zustand abfragen können

Lightbulb-Klasse in PHP

```php
<?php
class Lightbulb {

  private $light = true;

  public function switchOn() {
    $this->light = true;
  }
  public function switchOff() {
    $this->light = false;
  }
  public function isOn() {
    return $this->light;
  }
}
?>
```

- Schlüsselwort: **`class`**
- Klassenvariable ohne Typangabe
- Methode ohne Typangabe
- Methodendeklaration mit Schlüsselwort **`function`**

Zugriffssteuerung

```php
<?php

class Lightbulb {

  private $light = true;

  public function switchOn() {
    $this->light = true;
  }
  public function switchOff() {
    $this->light = false;
  }
  public function isOn() {
    return $this->light;
  }
}
?>
```

- Realisiert mit Hilfe von Attributen
- Attribute: **`private`**, **`public`**, **`protected`**
- **`protected`** beschränkt den Zugriff auf die Klasse, die das Element definiert und auf die aus dieser Klasse abgeleiteten Klassen

Interner Zugriff auf Variablen und Methoden

```php
<?php

class Lightbulb {

  private $light = true;

  public function switchOn() {
    $this->light = true;
  }
  public function switchOff() {
    $this->light = false;
  }
  public function isOn() {
    return $this->light;
  }
}
?>
```

- Erfordert Referenz auf das Objekt selbst:

  `$this`

- Zugriff auf Element mit Hilfe von Pfeiloperator `->`

- Achtung: `$`-Zeichen nur bei `$this`, nicht beim Variablennamen!

Instanzen erzeugen

- Sobald eine Klasse definiert ist, können Objekte dieser Klasse erzeugt werden

- Schlüsselwort **new**

```
<?php

// Klassendefinition...

$lightbulb = new Lightbulb();
// auch möglich: $lightbulb = new Lightbulb;

if ($lightbulb->isOn())
  echo("Glühbirne ist eingeschaltet");
else
  echo("Glühbirne ist ausgeschaltet");

?>
```

- Programmcode kann direkt hinter der Klassendefinition stehen

Zugriff auf Element

- Zugriff möglich, sofern nicht **`private`**
  - Ansonsten Fehlermeldung vom Interpreter
- Zugriff auf Element erfolgt mit Pfeiloperator **`->`**

```
<?php
// Klassendefinition...
$lightbulb = new Lightbulb();

if ($lightbulb->isOn())
// if ($lightbulb->light)
  echo("Glühbirne ist eingeschaltet");
else
  echo("Glühbirne ist ausgeschaltet");
?>
```

- Achtung: **`$`**-Zeichen nur beim Objekt-namen, nicht beim Variablen-namen!

Mehr als ein Objekt erzeugen

```
<?php
// Klassendefinition...

$lightbulb1 = new Lightbulb();
$lightbulb2 = new Lightbulb();

$lightbulb1->switchOn();
$lightbulb2->switchOff();

if ($lightbulb1->isOn())
  echo("Glühbirne 1 ist eingeschaltet<br />");
else
  echo("Glühbirne 1 ist ausgeschaltet<br />");

if ($lightbulb2->isOn())
  echo("Glühbirne 2 ist eingeschaltet<br />");
else
  echo("Glühbirne 2 ist ausgeschaltet<br />");
?>
```

Methoden mit Parameter

- Beispiel Polynomauswertung:

- Auch Methodenparametern können Vorgabewerte zugewiesen werden
- Optionale Parameter müssen nach den erforderlichen Parametern angegeben werden

```
<?php
class Polynom {
  public $counter = 1;

  public function polynomEvaluation($pa3, $pa2, $pa1,
    $pa0, $px=3) {

    $this->counter++;

    return(((($pa3*$px + $pa2)*$px + $pa1)*$px + $pa0);
  }
}
$poly = new Polynom();
echo($poly->polynomEvaluation(9,8,7,6,3) . "<br />"); // = 342
echo($poly->polynomEvaluation(9,8,7,6) . "<br />"); // = 342
echo($poly->counter);
?>
```

Benannte Parameter

- Beispiel Polynomauswertung:

- Es können benannte Parameter verwendet werden, um mehrere optionale Parameter zu überspringen

```
<?php
class Polynom {
  public $counter = 1;
  public function polynomEvaluation($pa3=9, $pa2=8, $pa1=7,
    $pa0=6, $px=3) {
    $this->counter++;
    return((($pa3*$px + $pa2)*$px + $pa1)*$px + $pa0);
  }
}
$poly = new Polynom();
echo($poly->polynomEvaluation(pa0:5) . "<br />"); // = 341
echo($poly->counter); // = 2
?>
```

- Kein $-Zeichen bei der Namensangabe erlaubt

Statische Eigenschaften: Externer Zugriff

- Klasseneigenschaften als statisch zu deklarieren macht diese zugänglich, ohne dass man die Klasse instanzieren muss
- Das erforderliche Schlüsselwort **static** steht hinter dem Zugriffsattribut
- Zugriff erfolgt mit Klassennamen und **Bereichsoperator ::**

```
<?php

class Polynom {
  public $counter = 1;

  public static function polynomEvaluation($pa3, $pa2, $pa1,
    $pa0, $px) {

    // $this->counter++; Kein Zugriff im statischen Kontext!
    return(((($pa3*$px + $pa2)*$px + $pa1)*$px + $pa0);
  }
}
echo(Polynom::polynomEvaluation(9,8,7,6,3) . "<br />");
?>
```

Statische Eigenschaften: Interner Zugriff

```php
<?php

class Polynom {

  public static $counter = 1;

  public static function polynomEvaluation($pa3, $pa2, $pa1,
    $pa0, $px) {

    self::$counter++;

    return(((($pa3*$px + $pa2)*$px + $pa1)*$px + $pa0);
  }
}

echo(Polynom::polynomEvaluation(9,8,7,6,3) . "<br />");
echo(Polynom::$counter);
?>
```

- Zugriff innerhalb der Klasse mit Bereichsoperator **::** und Schlüsselwort **self**
- Dollarzeichen steht beim Variablennamen!

Klassen in externen Dateien

- Klassen können in eigenen Dateien angelegt werden
  - Unterstützt Wiederverwendbarkeit
  - Erleichtert das Auffinden der Klassendefinition
- Datei dann z. B. mit **`require`** einbinden
- Andere Möglichkeit: Autoloader definieren und registrieren
  - Automatisch aufgerufene "magische" Methode

Autoloading: Beispiel

- Lightbulb-Klasse in externer Datei definiert

```
<?php
function my_autoloader($class_name) {
    require_once($class_name . '.php');
}

spl_autoload_register('my_autoloader');

$lightbulb = new Lightbulb();

if ($lightbulb->isOn())
  echo("Glühbirne eingeschaltet<br />");
else
  echo("Glühbirne ausgeschaltet<br />");

?>
```

- **`my_autoloader()`** wird automatisch vom PHP-Interpreter mit dem Namen der gesuchten Klasse aufgerufen
- **`require_once()`** bindet eine Datei ein und wertet diese zur Laufzeit des Skripts aus
- Verhalten ähnlich zu **`require()`** mit dem Unterschied, dass einmal eingebundener Code nicht noch einmal eingebunden wird

Methoden und Variablen überladen

Überladen: Allgemeines

- Sofern in einer Programmiersprache von einer Methode mehrere Varianten mit unterschiedlicher Signatur definiert werden können, wird vom **Überladen** gesprochen
- Mehrfache Verwendung eines Methodennamens möglich über den impliziten Aufruf der Funktion
  `__call ( string name, array arguments )`
  - Wird immer dann aufgerufen, wenn eine nicht-existierende Methode aufgerufen wird
- Mehrfache Verwendung eines Klassenvariablennamens ebenfalls möglich über den impliziten Aufruf der Funktion
  `__set ( string name, mixed value )`
- `mixed:` Pseudo-Typ – mehrere Typen werden hier akzeptiert

Beispiel: Verschiedene Methodennamen

```
class Polynom {

  function __call($name, $pa) {

    if ($name == "polynomEvaluation1")
      return(((($pa[4]*$pa[0] + $pa[3])*$pa[0] + $pa[2])*
              $pa[0] + $pa[1]);

    elseif ($name == "polynomEvaluation2")
      return(($pa[3]*$pa[0] + $pa[2])*$pa[0] + $pa[1]);
  }
}

$poly = new Polynom();

echo($poly->polynomEvaluation1(3, 6, 7, 8, 9) . "<br />");
echo($poly->polynomEvaluation2(3, 6, 7, 8) . "<br />");
```

- Methodenname im Parameter **$name**
- Methodenparameter im Feld **$pa**

Beispiel: Unterschiedliche Signatur

```
class Polynom {

  function __call($name, $pa) {

    if (count($pa) == 5)
      return((($pa[4]*$pa[0] + $pa[3])*$pa[0] +
               $pa[2])*$pa[0] + $pa[1]);

    elseif (count($pa) == 4)
      return(($pa[3]*$pa[0] + $pa[2])*$pa[0] + $pa[1]);

    else
      return("Unknown function");
  }
}
$poly = new Polynom();
echo($poly->polynomEvaluation(3, 6, 7, 8, 9) . "<br />");
echo($poly->polynomEvaluation(3, 6, 7, 8) . "<br />");
echo($poly->polynomEvaluation(3, 6, 7) . "<br />");
```

- Selektion der passenden Variante erfolgt über die Anzahl der übergebenen Parameter

Parametertypen

- Weitere Unterscheidung über Bestimmung der Parametertypen möglich:
  - `is_int($pa)`
  - `is_float($pa)`
  - `is_bool($pa)`
  - `is_string($pa)`
  - `is_array($pa)`

Beispiel: Varianten einer Klassenvariable

```
class VarOverloading {

  public $myVarArr = array("i" => 0, "d" => 0.0, "s" => "0");

  function __set($vname, $val)
  {
    if ($vname == "myVar")
      if (is_int ($val)) $this->myVarArr["i"] = $val;
      elseif (is_double ($val)) $this->myVarArr["d"] = $val;
      elseif (is_string ($val)) $this->myVarArr["s"] = $val;
  }
}

$vol = new VarOverloading();
$vol->myVar = 1;
$vol->myVar = 2.0;
$vol->myVar = "drei";

var_dump($vol->myVarArr);
```

- Ausgabe:

```
array(3) {
  ["i"]=> int(1)
  ["d"]=> float(2)
  ["s"]=> string(4) "drei"
}
```

Auslesen der Klassenvariable

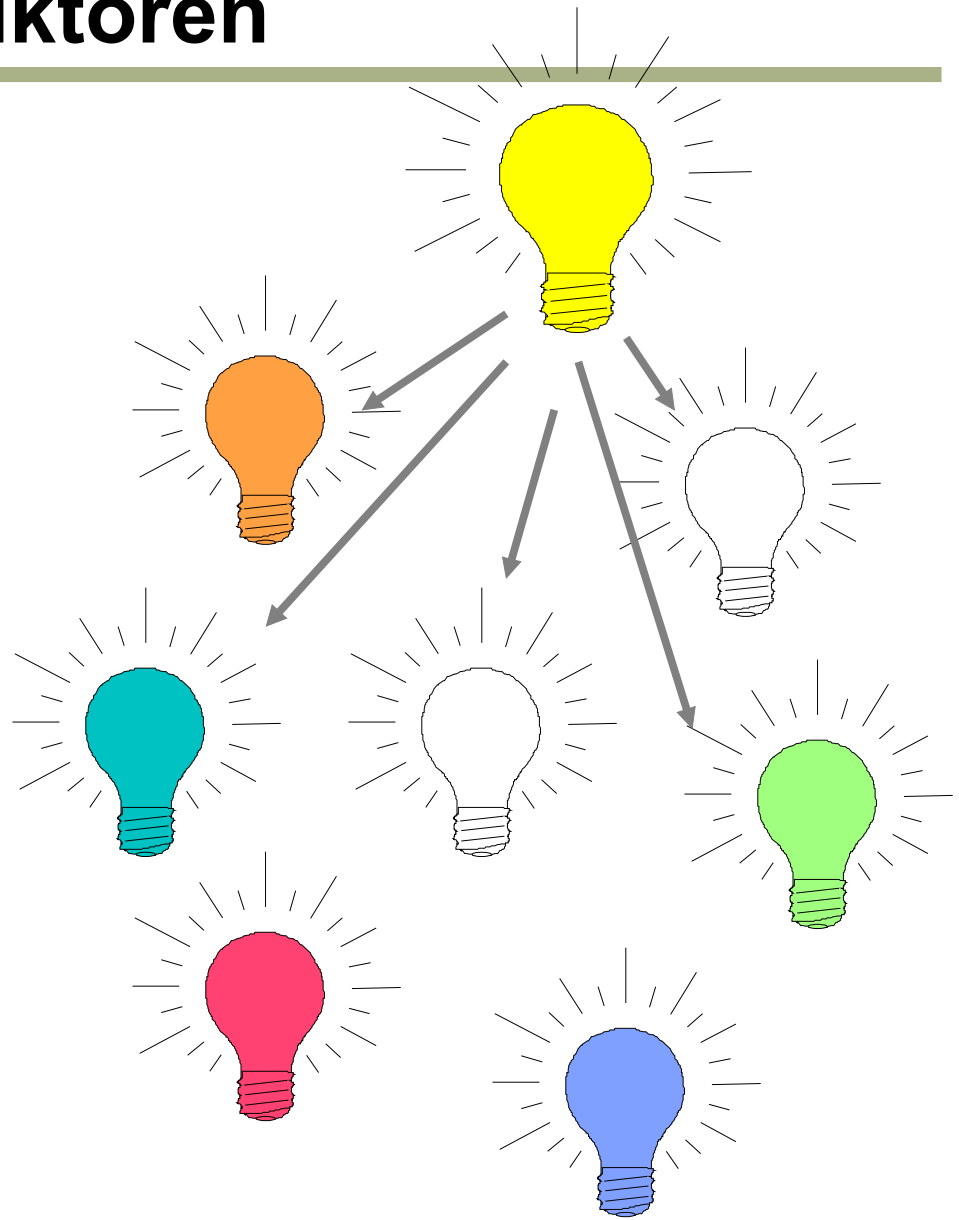
```
class VarOverloading {
  public $myVarArr = array("i" => 0, "d" => 0.0, "s" => "0");
  function __set($vname, $val) { // ... wie zuvor }
  function __get($vname){
    if (array_key_exists($vname, $this->myVarArr))
      return $this->myVarArr[$vname];
    else return null;
  }
}

$vol = new VarOverloading();
$vol->myVar = 1;
$vol->myVar = 2.0;
$vol->myVar = "drei";
echo("i: " . $vol->i . ", d: " . $vol->d . ", s: " . $vol->s);
```

- Ausgabe:
  `i: 1, d: 2, s: drei`

Konstruktoren

Konstruktoren

- Konstruktoren (engl.: *constructors*) erlauben es, bei der "Produktion" von Objekten einer Klasse zwischen verschiedenen Varianten auszuwählen



Konstruktor

```
class Lightbulb {

  function __construct() {
    $this->color = "white";
  }

  public function getColor() {
    return $this->color;
  }

  // more methods...
}

$lightbulb = new Lightbulb();

echo("Color: " .
      $lightbulb->getColor() .
     "<br />");
```

- Der Name des Konstruktors ist `__construct`
- Alte Methode: Name der Klasse
  - Geht aus Gründen der Abwärtskompatibilität ebenfalls noch
- Der Konstruktor wird automatisch aufgerufen, wenn ein Objekt der Klasse erzeugt wird

Konstruktor mit Parameter

```
class Lightbulb {

  function __construct($color) {

    $this->color = $color;

  }

  public function getColor() {
    return $this->color;
  }

  // more methods...
}

$lightbulb = new Lightbulb("red");

echo("Color: " . $lightbulb->getColor() . "<br />");

// Nicht möglich:
// echo("Color: $lightbulb->getColor() <br />");
```

Konstruktor mit variabler Parameterzahl

- Konstruktoren können nicht überladen werden
- Alternative: Variable Anzahl von Parametern analog zu Funktionen
-
```
class Lightbulb {

  function __construct() {
    $numpar = func_num_args();
    switch($numpar) {
      case 0: $this->color = "white"; break;
      case 1: $this->color = func_get_arg(0); break;
    }
  }

  // as before...
}

$lightbulb1 = new Lightbulb();
$lightbulb2 = new Lightbulb("red");
```

Komposition

Kompositionsbeispiel: Auto bauen

- Erstellt wird eine Klasse für ein Auto
- Das Auto besteht aus Teilen, die für sich selber wieder durch Klassen definiert sind:
- **`Car`**
  - **`Engine`**
  - **`Wheel`**
  - **`Door`**
    - **`Window`**

Motor

```
class Engine {
  function start() {
    echo("Engine started<br>");
  }
  function accelerator($rpm) {
    echo($rpm . "rpm<br>");
  }
  function stop() {
    echo("Engine stopped<br>");
  }
}
```

Rad

```
class Wheel {
  function inflate($pressure) {
    echo("Wheel pressure: " . $pressure . "<br>");
  }
}
```

Fenster

```
class Window {
   function rollup() {
      echo("Window up<br>");
   }
   function rolldown() {
      echo("Window down<br>");
   }
}
```

Türen: Komposition

```
class Door {
  function __construct() {
    $this->window = new Window();
  }
  function open() {
    echo("Door open<br>");
  }
  function close() {
    echo("Door closed<br>");
  }
}
```

Auto: Komposition

```
class Car {
  function __construct() {
    $this->engine = new Engine();
    $this->left = new Door();
    $this->right = new Door(); // 2-door
    for($i = 0; $i < 4; $i++)
      $this->wheel[$i] = new Wheel();
  }
}
```

Hauptprogramm

```
$car = new Car();
$car->left->window->rollup();
$car->wheel[0]->inflate(72);
$car->left->close();
$car->right->close();
```

```
Ausgabe:
Window up
Wheel pressure: 72.0
Door closed
Door closed
```

Klasse Door mit Konstruktorparameter

```
class Door {
  function __construct($whichDoor) {
    $this->window = new Window();
    $this->whichDoor = $whichDoor;
  }
  function open() {
    echo($this->whichDoor . " open<br>");
  }
  function close() {
    echo($this->whichDoor . " closed<br>");
  }
}
```

Car mit Parameterwerten

```
class Car {
  function __construct() {
    $this->engine = new Engine();
    $this->left = new Door("Left door");
    $this->right = new Door("Right door"); //
    for($i = 0; $i < 4; $i++)
      $this->wheel[$i] = new Wheel();
  }
}
```

Hauptprogramm: Neue Ausgabe

```
$car = new Car();
$car->left->window->rollup();
$car->wheel[0]->inflate(72);
$car->left->close();
$car->right->close();
```

Ausgabe:

```
Window up
Wheel pressure: 72.0
Left door closed
Right door closed
```

Vererbung

Einführung

- Vererbung ermöglicht die Definition einer neuen Klasse auf der Grundlage einer schon existierenden

- Dabei erbt die neu definierte Klasse die Datenfelder und Methoden der Oberklasse, aus der sie abgeleitet wurde
  - Grundfunktionen können weiterverwendet werden
  - Auf diese Weise wird Code-Duplikation vermieden

- Schlüsseltechnik zur Steigerung der Effizienz in der Programmierung

Vererbung in PHP

- Wird durch Angabe des Schlüsselworts **`extends`** eingeleitet

```
class Triangle extends Polygon {

    ...

}
```

- In PHP erben Unterklassen nur von **einer** Oberklasse
  - **Single-Inheritance-Prinzip**

Beispiel

```
<?php
class Polygon {
  public function setColor($col) {
    $this->col = $col;
  }
  public function draw() {
    echo("Drawing a $this->col polygon<br />");
  }
}

class Triangle extends Polygon {
   public function draw() {
     echo("Drawing a $this->col triangle<br />");
   }
}

$p = new Polygon(); $p->setColor("red");
$t = new Triangle(); $t->setColor("blue");
$p->draw(); $t->draw();
?>
```

```
Ausgabe:
Drawing a red polygon
Drawing a blue triangle
```

Konstruktor der Oberklasse

```
<?php

class Polygon {

  public function __construct($col) {
    $this->col = $col;
  }
  public function draw() {
    echo("Drawing a $this->col polygon<br />");
  }
}

class Triangle extends Polygon {
   public function draw() {
     echo("Drawing a $this->col triangle<br />");
   }
}

$p = new Polygon("red"); $t = new Triangle("blue");

$p->draw(); $t->draw();
?>
```

- Wird implizit aufgerufen, wenn Unterklasse keinen eigenen Konstruktor definiert

Konstruktor der Oberklasse: Expliziter Aufruf

```
<?php
class Polygon {
  // as before...
}

class Triangle extends Polygon {
  public function __construct($col, $x, $y) {
    parent::__construct($col);
    $this->x = $x; $this->y = $y;
  }

  public function draw() {
    echo("Drawing a $this->col triangle at
          $this->x, $this->y<br />");
  }
}

$t = new Triangle("blue", 10, 20);
$t->draw();
?>
```

- Expliziter Aufruf mit `parent::__construct(...)` möglich

```
Ausgabe:
Drawing a blue triangle at 10, 20
```

Zugriff auf Methoden und Variablen

```
<?php
class Polygon {
  // as before...
}

class Triangle extends Polygon {
  public function __construct($col, $x, $y) {
    parent::__construct($col);
    $this->x = $x; $this->y = $y;
  }
  public function draw() {
    parent::draw();
    echo("A triangle at $this->x, $this->y<br>");
  }
}

$t = new Triangle("blue", 10, 20);
$t->draw();
?>
```

- Expliziter Zugriff mit `parent::Methodenname(...)` bzw. `parent::Variablennname` möglich

```
Ausgabe:
Drawing a blue polygon
A triangle at 10, 20
```

Abstrakte Klassen

Abstrakte Klassen

- PHP bietet die Möglichkeit, Schnittstellen zu definieren, die implementiert werden müssen
  - Schnittstellendefinitionen, die mit Hilfe leerer Methodenrümpfe realisiert werden, stellen eine potentielle Fehlerquelle dar, da deren Benutzung freiwillig ist

- Dies wird mit Hilfe **abstrakter Klassen** oder **Interfaces** durchgeführt

Schnittstellendefinition durch abstrakte Klassen

- Abstrakte Klassen werden durch das Schlüsselwort **`abstract`** deklariert

```
abstract class Shape {
  abstract function draw();
}
```

- Abstrakte Methoden besitzen nur einen Methodenkopf
  - Dieser definiert die zu implementierende Schnittstelle
  - Methodenname und Anzahl der Parameter legen Signatur fest
- **Abstrakte Klassen können nicht instanziert werden!**

Beispiel

```php
<?php
abstract class Shape {
  public function setColor($col) {
    $this->col = $col;
  }
  public function getColor($col) {
    return $this->col;
  }
  abstract function draw();
}
?>
```

- **`draw()`** muss in abgeleiteten Klassen implementiert oder diese Klassen als **`abstract`** deklariert werden

Beispiel

```
<?php

abstract class Shape {

  // wie zuvor...
  abstract function draw();
}

class Polygon extends Shape {

  public function __construct($col) {
    $this->col = $col;
  }

  public function draw() {
    echo("Drawing a $this->col polygon<br />");
  }
}

$p = new Polygon("red");
$p->draw();
?>
```

- **`draw()`** muss in abgeleiteten Klassen implementiert oder diese Klassen als **`abstract`** deklariert werden

Interfaces

Interfaces

- Mit Hilfe von Interfaces wird das Konzept der abstrakten Klasse hin zu einer reinen Schnittstelle weitergeführt
- In Interfaces werden **nur Methodenköpfe** deklariert
  - Interfaces können **keine Methodenrümpfe** enthalten
- Ein Interface entspricht einer abstrakten Klasse, die nur über abstrakte Methoden verfügt
- Alle in einem Interface deklarierten Methoden sind automatisch **`public`**

Interfaces

- Interfaces werden mit Hilfe des Attributs **`interface`** deklariert
  - Dies wird an Stelle des Attributs **`class`** verwendet
- Interfaces werden nicht durch Unterklassen erweitert, sondern von Klassen **implementiert**
- Hierfür wird das Schlüsselwort **`implements`** benutzt
- Klassen dürfen mehr als ein Interface implementieren
  - Die Interfaces werden voneinander mit Kommata abgetrennt

Beispiel

```
<?php
interface Shape {
  function draw();
}
class Polygon implements Shape {

  public function __construct($col) {
    $this->col = $col;
  }

  public function draw() {
    echo("Drawing a $this->col polygon<br />");
  }
}
$p = new Polygon("red");
$p->draw();
?>
```

- **`draw()`** muss in **`Polygon`** implementiert oder die Klasse als **`abstract`** deklariert werden

Datenbanken: Grundlagen

Einführung

- Beispiel:
  - Vermietung von Standplätzen durch eine Stadtverwaltung
  - Beispielsweise für:
    - Wochenmarkt
    - Saisonaler Markt (Weihnachtsmarkt, Kirmes etc.)
    - Flohmarkt etc.
  - Verwaltung der Standplätze und Mieter bisher z. B. mit Karteikartensystem
  - Soll nun mit Hilfe des Computers geschehen

Datenerfassung

- Zunächst sollen nur Standnummern mit Mietern assoziiert werden
  - Um z. B. das mehrfache Vermieten eines Standplatzes zu vermeiden
- Hierfür wird der *Name* und die *Adresse* des Standmieters nebst dessen *Telefonnummer* erfasst sowie die zugehörende *Standnummer*
- Für die Verwaltung der Daten wird ein *Programm* geschrieben, welches die Daten in einer *Datei* vorhält

Dateistruktur

- Die Datei wird in sogenannte *Datensätze* gegliedert
  - Ein Datensatz ist ein *Aggregat* unterschiedlicher Daten
    - Z. B. Nummer des Standplatzes plus Name und Adresse des Mieters
- Aufbau der Datei beispielsweise wie folgt:
  - **Datensatz 1**: Speicherung der Anzahl *n* der Standplätze
  - **Datensatz 2**: Speicherung der Anzahl *m* der noch verfügbaren Standplätze
  - **Datensatz 3 bis *n*+2**: Nummer des Standplatzes plus Name und Adresse des Mieters

Erweiterung

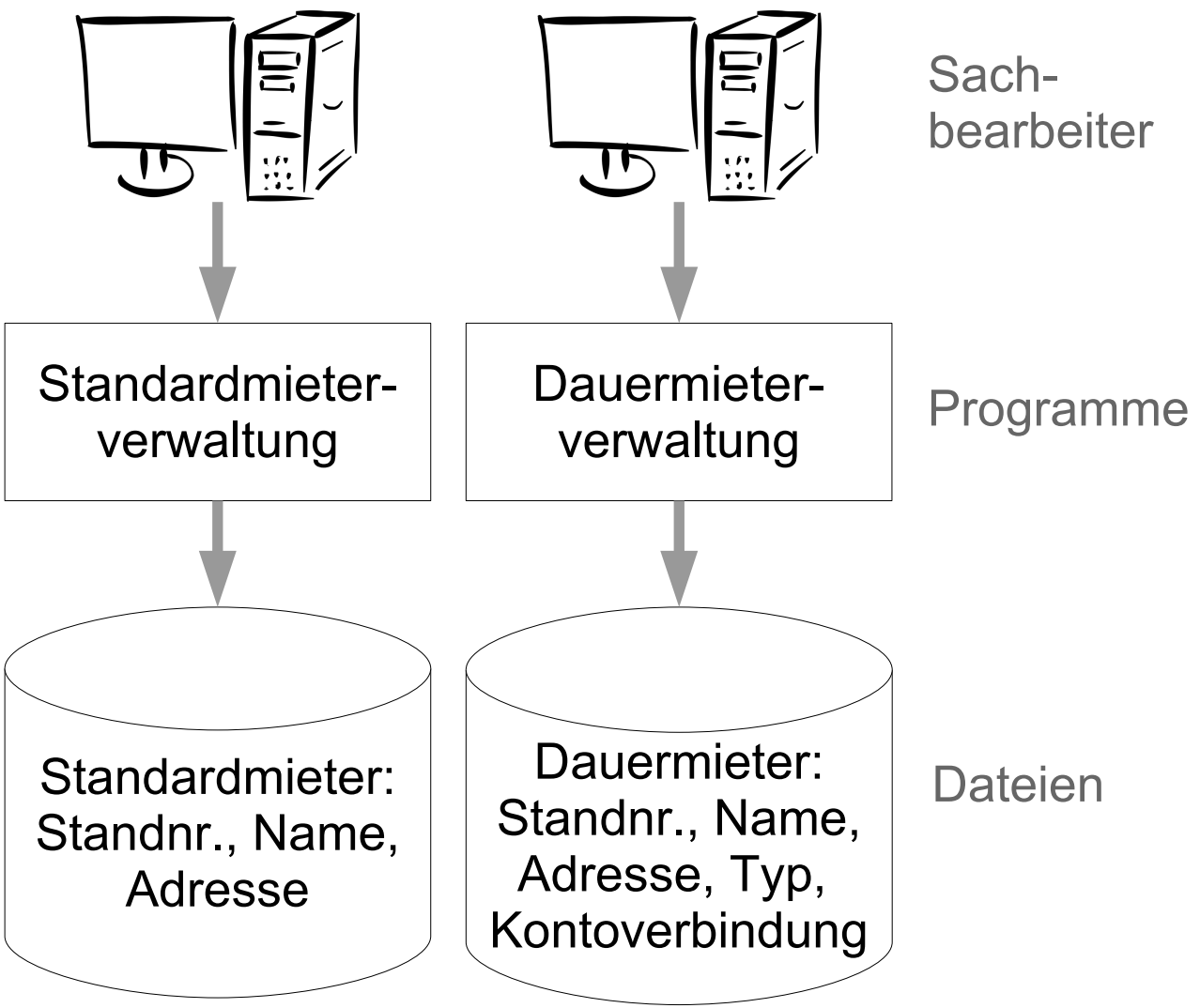
- Nach einiger Zeit hat sich das System erfolgreich etabliert und soll daher erweitert werden
- Eingeführt werden soll eine bequemere Abwicklung der Bezahlung für Dauermieter an Stelle des Abkassierens vor Ort:
  - Nun Zustellung von Rechnungen möglich oder
  - Abbuchung vom Konto, falls Einzugsermächtigung vorliegt
- Hierfür wird die Erfassung der Dauermieter mit *Name*, *Adresse*, *Art der Rechnung* und ggf. *Kontoverbindung* benötigt

Probleme

- Die meisten der Dauermieter sind bereits in der Datei des vorhandenen Programms als Standmieter mit ihrer Adresse erfasst
- Existierende Datei könnte also mitbenutzt werden
  - *Problem 1:* Die interne Struktur der Datei müsste abgeändert werden, um die Speicherung der neu hinzukommenden Daten zu ermöglichen
    - Art des Mieters und der Rechnung
    - Kontoverbindung
  - *Problem 2:* Das bestehende Programm muss angepasst werden, sodass es mit der neuen Dateistruktur arbeiten kann

Probleme

- Änderung des Programms und der bestehenden Dateistruktur zu aufwendig und teuer
- Als preiswerteste Lösung stellt sich heraus, ein weiteres Programm für die *Verwaltung der Dauermieter* zu erstellen



Probleme der separaten Datenverwaltung

- Dieselbe Information kommt in mehreren Dateien vor
- Hier: Name, Adresse und Telefonnummer der Mieter
  - Einmal in Datei für Standardmieter und ebenfalls in Datei für Dauermieter vorhanden
- Hieraus entsteht das Problem der *redundanten Datenhaltung*

Probleme der separaten Datenverwaltung

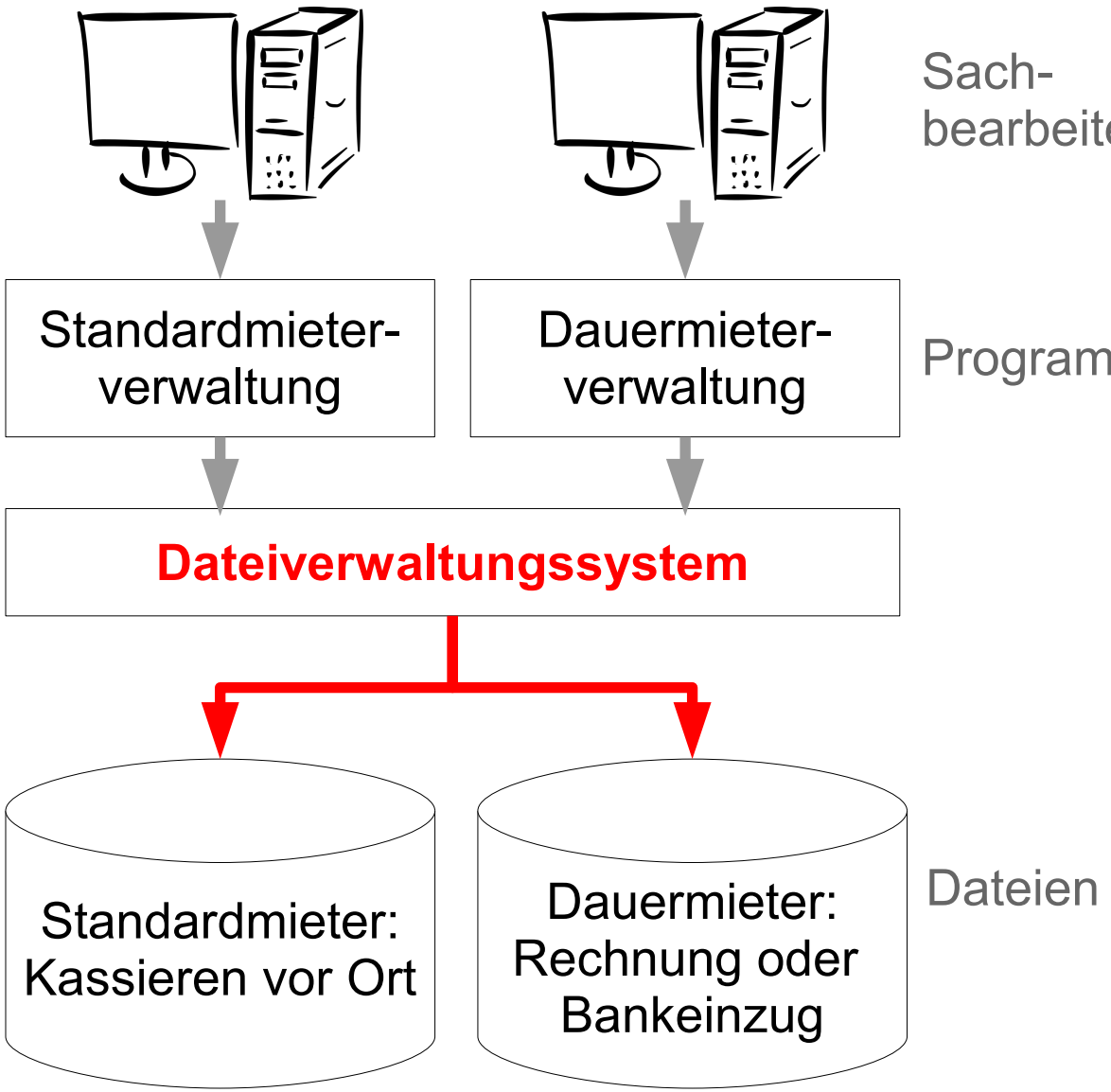
- Ändern sich diese Daten, so müssen mehrere Dateien angepasst werden
  - Erforderlich bei:
    - Neuer Adresse
    - Ende oder Änderung des Mietverhältnisses
    - Wechsel des Standplatzes
    - usw.
- Unnötige Arbeit
- Mögliche Fehlerquelle: Wird eine Datei vergessen, so drohen *inkonsistente Datensätze!*

Probleme der separaten Datenverwaltung

- Ändert sich die Dateistruktur, so impliziert dies eine Anpassung der Programme
- Folge von:
  - Änderung des Aufbaus der Datensätze
  - Änderung der Reihenfolge der Datensätze
  - Wechsel des Speichermediums
- Diese Abhängigkeit zwischen der Organisation einer Datei und den damit arbeitenden Programmen wird als *physische Datenabhängigkeit* bezeichnet

Lösung

- Physische Daten-unabhängigkeit durch *Dateiverwaltungs-system*
  - Eingebettet zwischen Dateien und Anwendungen
  - Vermeidet Inkonsistenzen durch kontrollierte Aktualisierung der Datensätze



Dateiverwaltungssystem

- Stellt Anwendungen *genormte Dateioperationen* zur Verfügung
  - Anwendungen greifen jetzt nicht mehr direkt, sondern mit Hilfe der Dateioperationen des Verwaltungssystems auf die Daten zu
- Erzielt physische Unabhängigkeit von:
  - Anordnung der Datensätze innerhalb der Dateien
  - Anordnung der Daten innerhalb der Datensätze
  - Physikalischer Anordnung der Daten auf dem Speichermedium

Bestehende Probleme

- Angenommen, bei Rechnungserstellung oder Abbuchung soll zusätzlich das Datum abgespeichert werden
  - Für Mahnungen und um zu vermeiden, dass mehr als eine Rechnung pro Berechnungszeitraum zugeschickt oder Abbuchung vorgenommen wird
- Derartige Änderungen an der Datenstruktur implizieren, dass
  - die Dateien und das Dateiverwaltungssystem angepasst werden müssen
  - sowie die von der Änderung betroffenen Programme
    - Anderenfalls wären weitere Programme erforderlich

Bestehende Probleme

- Es besteht also noch eine *logische Datenabhängigkeit*
- D. h. eine Abhängigkeit der Anwendungen vom Aufbau der Datensätze
  - Die Datensätze lassen sich bisher nicht auf einfache Weise erweitern oder verkleinern
- Deshalb ist eine Entkopplung der Anwendungen von Änderungen an den Datenstrukturen erforderlich
- Dies wird als *logische Datenunabhängigkeit* bezeichnet

Logische Datenunabhängigkeit

- Wird ermöglicht durch die Einführung sogenannter *logischer Datenstrukturen*
- Diese lassen sich beliebig ändern, da sie vom physischen Aufbau der Datensätze unabhängig sind
- Logische Datenstrukturen beruhen auf:
  - Der Einführung sog. *logischer Datenelemente*
  - Der Definition anwendungsspezifischer Beziehungen zwischen diesen Datenelementen
- Logische Datenstrukturen erlauben verschiedene *Sichten* auf die Gesamtmenge der Daten

Datenbank

- Alle Daten werden zur *Redundanzvermeidung* in einem gemeinsamen Bereich gespeichert
  - Dieser Bereich heißt *Datenbank*
- Der Zugriff auf die Daten in der Datenbank erfolgt durch einfache Sprachkonstrukte, die allen Anwendungen zur Verfügung stehen
- Mit Hilfe der Sprachkonstrukte erhält jede Anwendung genau die Sicht auf die Daten, die für die Lösung der jeweiligen Aufgabenstellung erforderlich ist

Datenintegrität

- Von kritischer Bedeutung ist die Gewährleistung der *Datenintegrität* innerhalb der Datenbank
- Die Integrität der Daten ist gewährleistet, wenn diese korrekt sind und korrekt verwendet werden
- Die Datenintegrität gliedert sich in drei Teilaspekte:
  - *Datenkonsistenz*
  - *Datensicherheit*
  - *Datenschutz*

Datenkonsistenz

- Zu den Aufgaben der Konsistenzüberprüfung gehört es zu gewährleisten, dass Kopien von Daten im System jederzeit alle
  - *auf dem gleichen Stand*,
  - *korrekt* und
  - *widerspruchsfrei* sind

- Dies wird als *semantische Integrität* bezeichnet
  - *Beispiel*: Die Adressen der einfachen Standmieter und Dauermieter müssen automatisch angeglichen werden bei Änderung

Datenkonsistenz

- Zu den Aufgaben der Konsistenzüberprüfung gehört es auch zu gewährleisten, dass bei gleichzeitigem Zugriff auf die Daten durch mehrere Benutzer keine inkonsistenten Daten sichtbar werden

- Dies wird als *operationale Integrität* bezeichnet

  - *Beispiel 1:* Sachbearbeiter liest Adresse eines Mieters aus, während anderer Sachbearbeiter die Daten dieses Mieters gerade bearbeitet oder gar löscht

  - *Beispiel 2:* Es muss verhindert werden, dass Kundendaten gelöscht werden, wenn noch Rechnungen offen sind

Datensicherheit

- Nach dem Ausfall von Betriebsmitteln (z. B. Absturz des Systems) muss es möglich sein, die Datenbank in einen konsistenten Zustand zurückzubringen

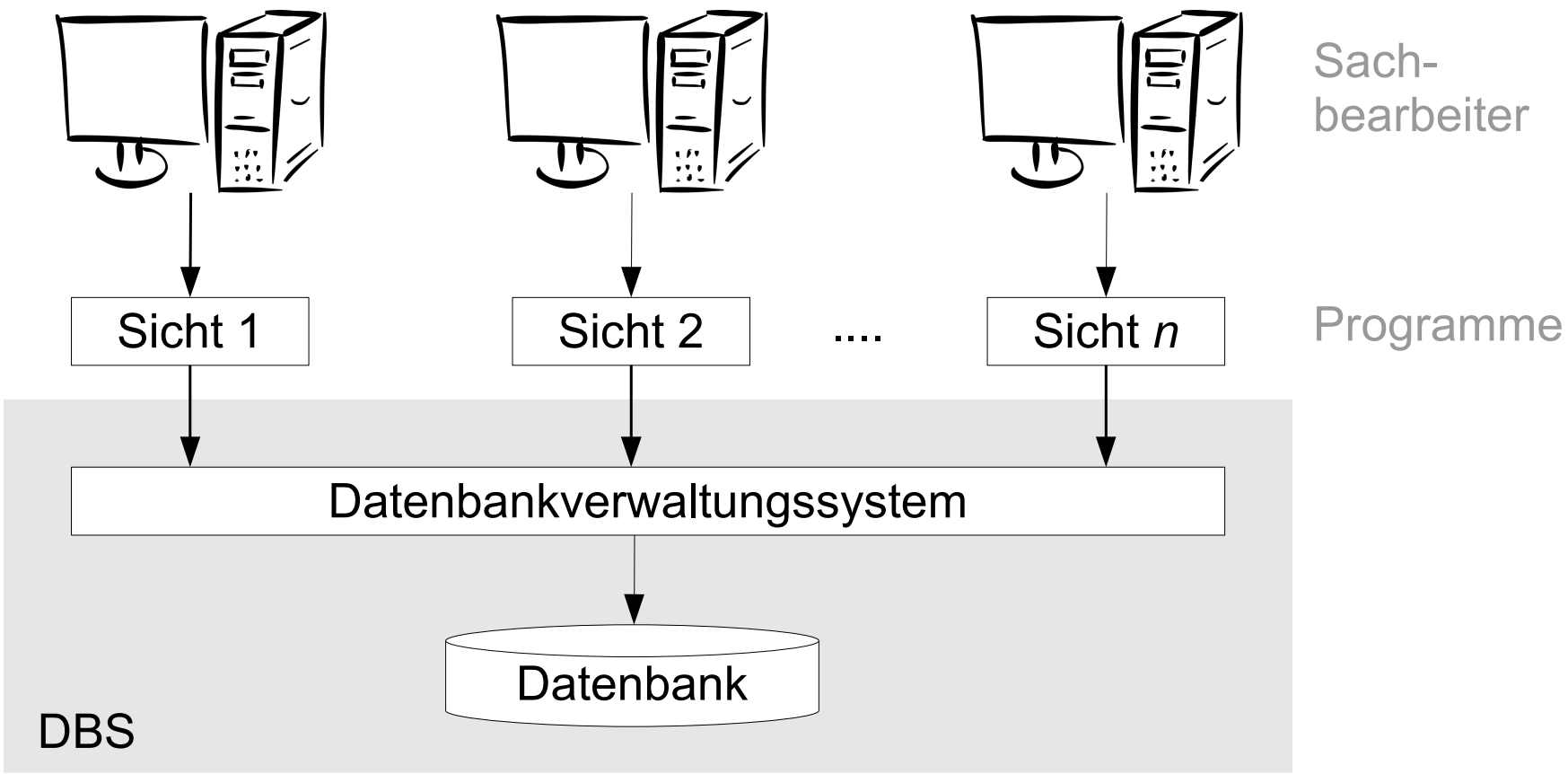
- Dabei müssen Datenverluste jeglicher Art möglichst vermieden werden

Datenschutz

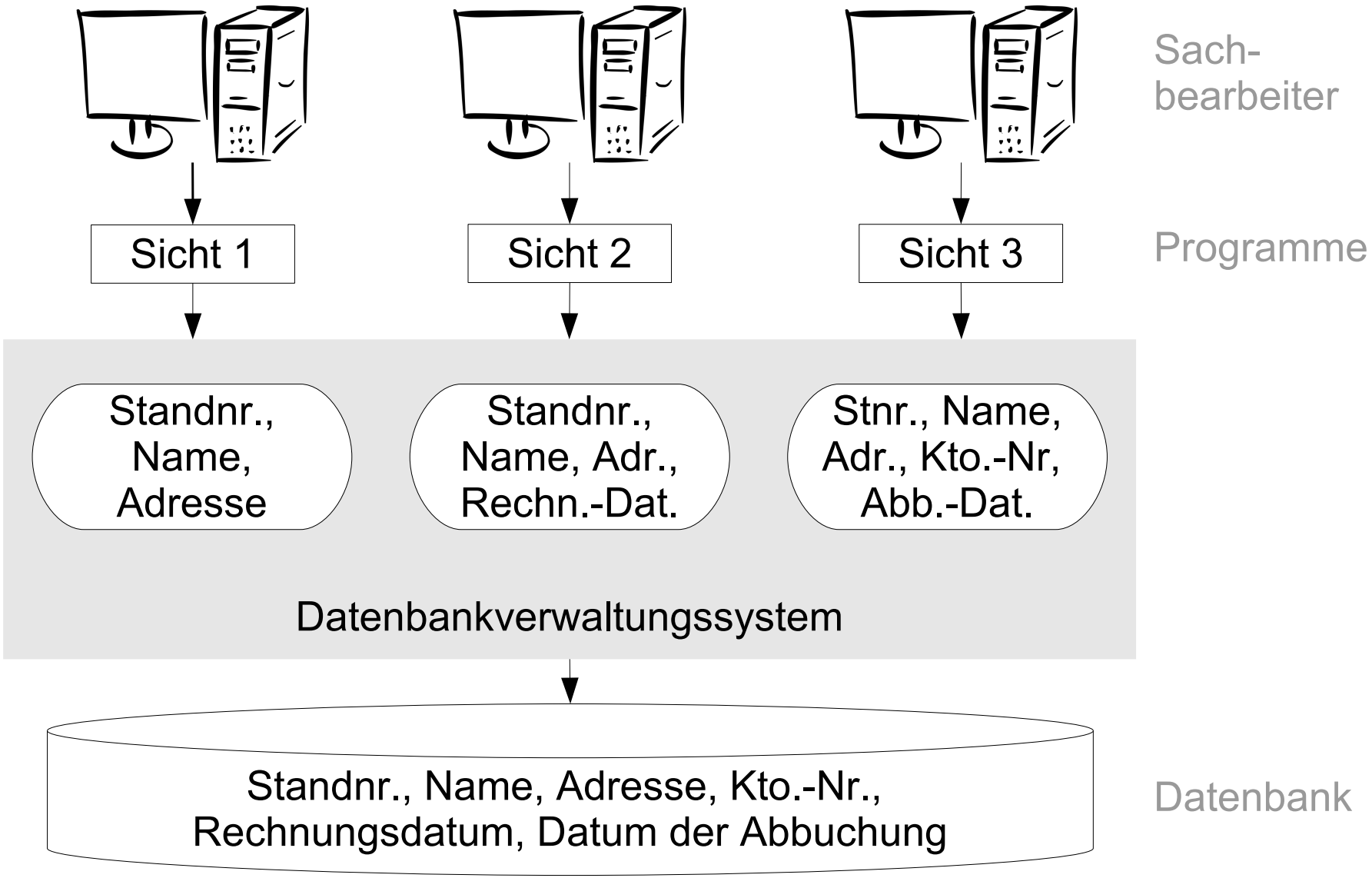
- Nicht jeder Anwender bzw. jede Anwendung benötigt jederzeit in jeder Weise Zugriff auf alle verfügbaren Daten
- Der sichtbare Bereich des Datenbestandes muss daher an die jeweilige Anwendung anpassbar sein
- Es muss ebenfalls sichergestellt werden, dass keine Unbefugten Zugriff auf die Daten erhalten

Datenbanksystem

- Ein System, welches diese Anforderungen erfüllt, wird als *Datenbanksystem* (DBS) bezeichnet



Beispiel: Standplatzvermietung



Umsetzung

- Um ein effizientes und entkoppeltes Design zu gestatten, werden DBS - genauso wie Betriebssysteme, Software-Pakete oder Übertragungsprotokolle – in voneinander unabhängige Systemebenen eingeteilt
  - Diese Systemebenen kommunizieren über definierte Schnittstellen miteinander
- Eine wichtige Architektur ist in diesem Zusammenhang die "*Drei-Schichten-Architektur*" nach ANSI-SPARC (1975)
  - ANSI: American National Standards Institute
  - SPARC: Standards Planning and Requirements Committee

Drei-Schichten-Architektur

- Unterteilt ein Datenbanksystem in drei unterschiedliche Ebenen, auf denen Datenstrukturen beschrieben werden:
  - *Konzeptionelle Ebene*
  - *Externe Ebene*
  - *Interne Ebene*

Konzeptionelle Ebene

- Auf der konzeptionellen Ebene wird die logische Gesamtsicht aller Daten und ihrer Beziehungen untereinander im *konzeptionellen Schema* repräsentiert
- Das konzeptionelle Schema enthält Beschreibungen aller relevanten Datenstrukturen
  - Hier geht es also um die Frage, welche Daten in der Datenbank abgelegt werden sollen
- Die Art der Beschreibung ist abhängig vom gewählten Datenmodell

Konzeptionelle Ebene

- Umfasst auch Beschreibung von Integritätsbedingungen
- Beispiele für Integritätsbedingungen:
  - Ein Standplatz kann nicht mehr als einen Mieter haben
  - Jeder Kunde hat genau eine Kundennummer
  - Kundennummern sind eindeutig
  - Jeder Kunde hat genau eine Kontoverbindung
  - etc.

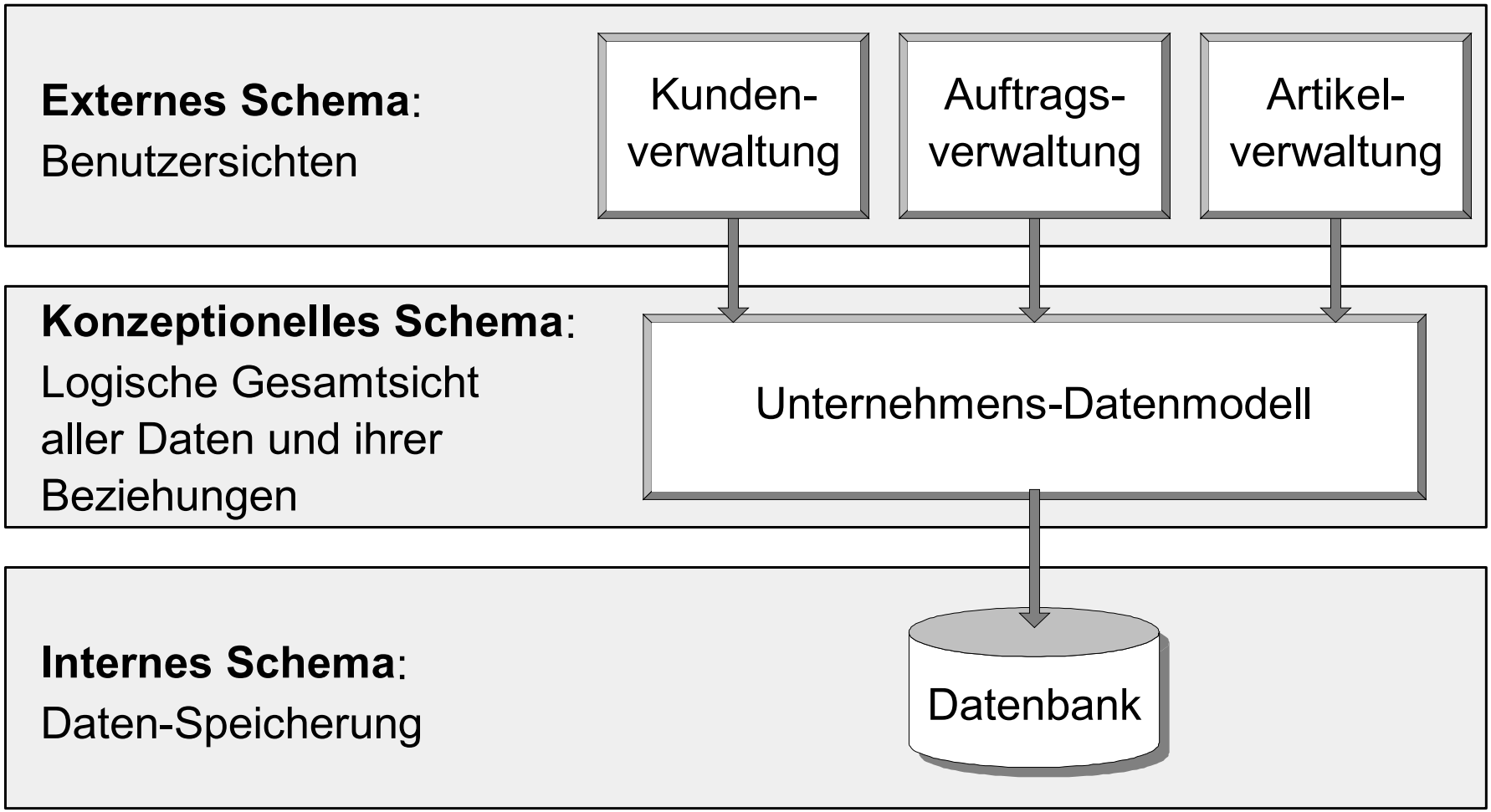
Externe Ebene

- Umfasst die individuellen Sichten der Benutzer bzw. Benutzergruppen auf die Datenbank
- Diese Sichten werden jeweils in einem eigenen *externen Schema* beschrieben
- In den externen Schemata werden Teilbereiche der Datenbank für verschiedene Nutzergruppen definiert
- Das externe Schema enthält den Teilbereich des konzeptionellen Schemas, welches der jeweilige Benutzer sehen können soll
- Hier werden also auch die Nutzungsrechte für die verschiedenen Benutzer festgelegt

Interne Ebene

- Ebene der physischen Datensätze
- Beschreibung der Speicherdarstellung der Datensätze nebst zugehörender Zugriffsmechanismen und -pfade im *internen Schema*
- Darstellung der Einzelheiten der Implementierung

ANSI-SPARC-Architektur

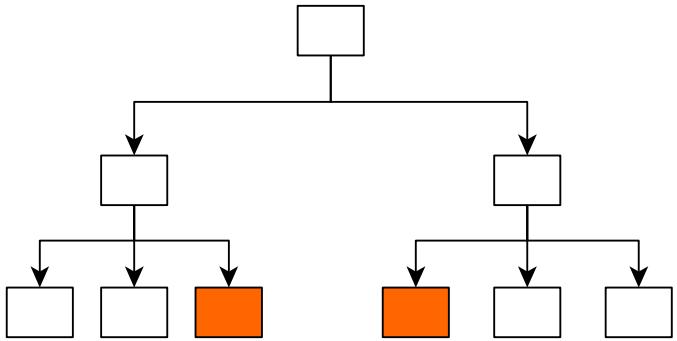


Datendefinitionssprache

- Zur Beschreibung der logischen Datenstrukturen des konzeptionellen Schemas und für die Definition der externen Schemata wird eine Sprache benötigt
- Diese Sprache wird als *Datendefinitionssprache* (engl.: *data definition language,* Abk.: *DDL*) bezeichnet
- Die DDL erlaubt die Beschreibung von Objekten und ihre Beziehungen untereinander beruhend auf einem bestimmten *Datenmodell* (engl.: *data model*)
- Im Datenmodell sind die elementaren Datenobjekte und ihre Beziehungen untereinander festgelegt
- Jedem Datenbanksystem liegt ein Datenmodell zugrunde

Datenmodelle: Hierarchisches Modell

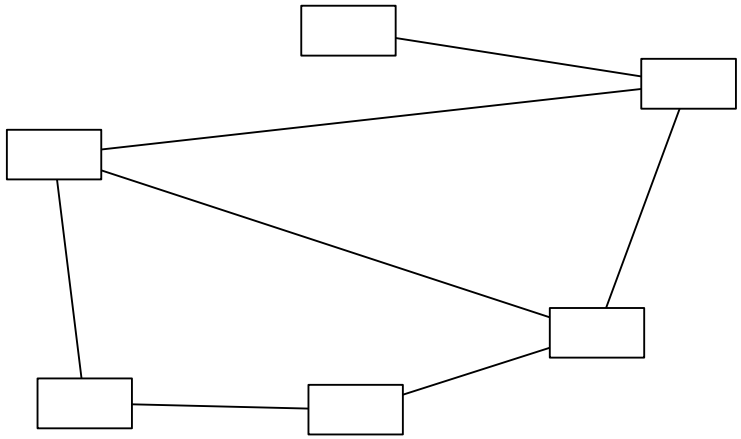
- Ältestes Modell
- Von IBM Ende der 1960er Jahre im Zusammenhang mit dem Datenbanksystem IMS (Information Management System) entwickelt
- Baumstruktur
- Datensätze sind Knoten ("Records") und bestehen aus einzelnen Feldern ("Data Items")



- Datenspeicherung vergleichbar mit XML
- Es lassen sich nur 1:1 und 1:n-Beziehungen darstellen
  - n:m-Beziehungen führen zu Datenredundanz
- Besitzt nur noch geringe Bedeutung

Datenmodelle: Netzwerkmodell

- In den 1970er Jahren von der DBTG ("data base task group") entwickelt
- Datenstruktur besteht wieder aus Records
- n:m-Beziehungen kein Problem
  - Datensatz kann eine beliebige Anzahl übergeordneter Datensätze besitzen
- Modellierung und Datenzugriff vergleichsweise kompliziert
  - Erfordert die Entwicklung von "Zugriffs-Algorithmen"
- Besitzt nur noch geringe Bedeutung



Datenmodelle: Relationales Modell

- Ab 1970 von E. F. Codd entwickelt und seit den 1980er Jahren kommerziell eingesetzt
- Daten werden mittels *Tabellen* repräsentiert
  - Intuitiv, einfach benutzbar
- Insbesondere im Web-Bereich das dominierende Modell
- Deswegen auch hier Grundlage der weiteren Betrachtungen

Datenmodelle: Objektorientiertes Modell

- Übernimmt Konzepte des objektorientierten Software Engineering
- Wird primär für das Zwischenspeichern von Daten objektorientierter Anwendungen eingesetzt
  - Datenstrukturen können 1:1 anwendungsinterne Objekte abbilden
  - Vermeidet konzeptionelle Schwächen des relationalen Modells für diesen Einsatzbereich
- Steht hinsichtlich allgemeiner Aufgaben im Schatten des relationalen Modells

Datenbeschreibung (1)

- Erfolgt durch Datendefinitionssprache. Diese erlaubt:
  - Eine logische Beschreibung der Datenstrukturen
  - Eine Beschreibung der Wertebereiche für jedes Element einer logischen Datenstruktur
  - Die Vergabe von Zugriffsrechten
  - Die Benennung von Datenstrukturelementen zur eindeutigen Identifikation bestimmter Datensätze
    - Derartige Datenstrukturelemente werden als *Schlüssel* bezeichnet

Datenbeschreibung (2)

- Beispiel: Datendefinitionssprache von SQL
- Erzeugung einer logischen Datenstruktur für Standmieter:

```
CREATE TABLE Standmieter (
  KundenNr INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
  Name VARCHAR(20),
  Vorname VARCHAR(20),
  Standnummer INT
);
```

Datenverarbeitung (1)

- Neben einer Sprache zur Beschreibung der Daten wird ebenfalls eine Sprache zur Manipulation der Daten benötigt
  - Suchen/Einfügen/Ändern/Löschen von logischen Datenstrukturen
- Eine solche Sprache wird *Datenmanipulationssprache* (engl.: *data manipulation language* - Abk.: *DML* - oder *query language*) genannt

Datenverarbeitung (2)

- Beispiel: Datenmanipulationssprache von SQL
  - In die Tabelle Standmieter wird ein neuer Datensatz eingefügt:

```
INSERT INTO Standmieter SET
   Name = "Meier",
   Vorname = "Bernd",
   Standnummer = 1;
```

- In SQL sind die DDL und DML in einer Sprache vereinigt

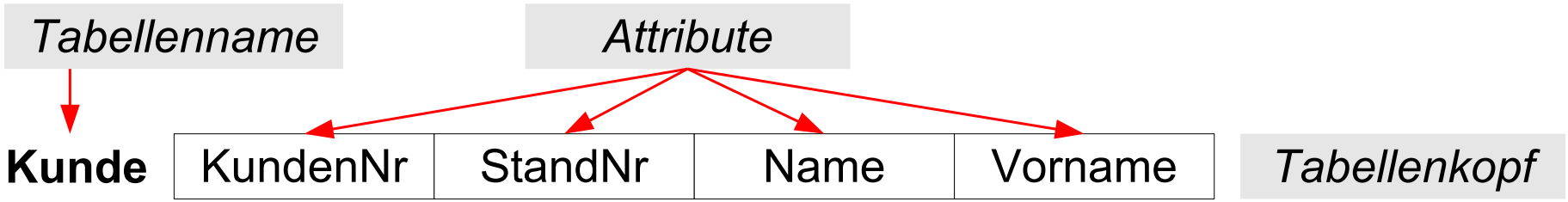
Relationale Datenmodellierung

Relationales Datenmodell

- Baut auf der mathematischen Theorie der Relationenalgebra auf
  - Zum Verständnis des Modells nicht erforderlich
- Grundstruktur des relationalen Datenmodells ist die *Tabelle*
- Tabellenstruktur ist Ursache für die einfache Handhabbarkeit
  - Jeder ist daran gewöhnt, Tabellen zu lesen, zu interpretieren und Daten in Tabellenform zu organisieren

Tabelle

- Eine Tabelle besteht aus *Tabellenkopf*, *Zeilen* und *Spalten*



- Der Tabellenkopf beschreibt eine Klasse von Objekten an Hand diverser Eigenschaften
  - Jede Eigenschaft wird durch eine eigene Spalte repräsentiert
  - Eigenschaften werden auch als *Merkmale* oder *Attribute* des Objekts bezeichnet

Tabelle

- Jede Tabellenzeile repräsentiert einen Datensatz
  - Datensatz wird auch als *Tupel* bezeichnet
- Datensätze stellen *Ausprägungen* bzw. *Instanzen* der durch die Tabelle definierten Klasse von Objekten dar

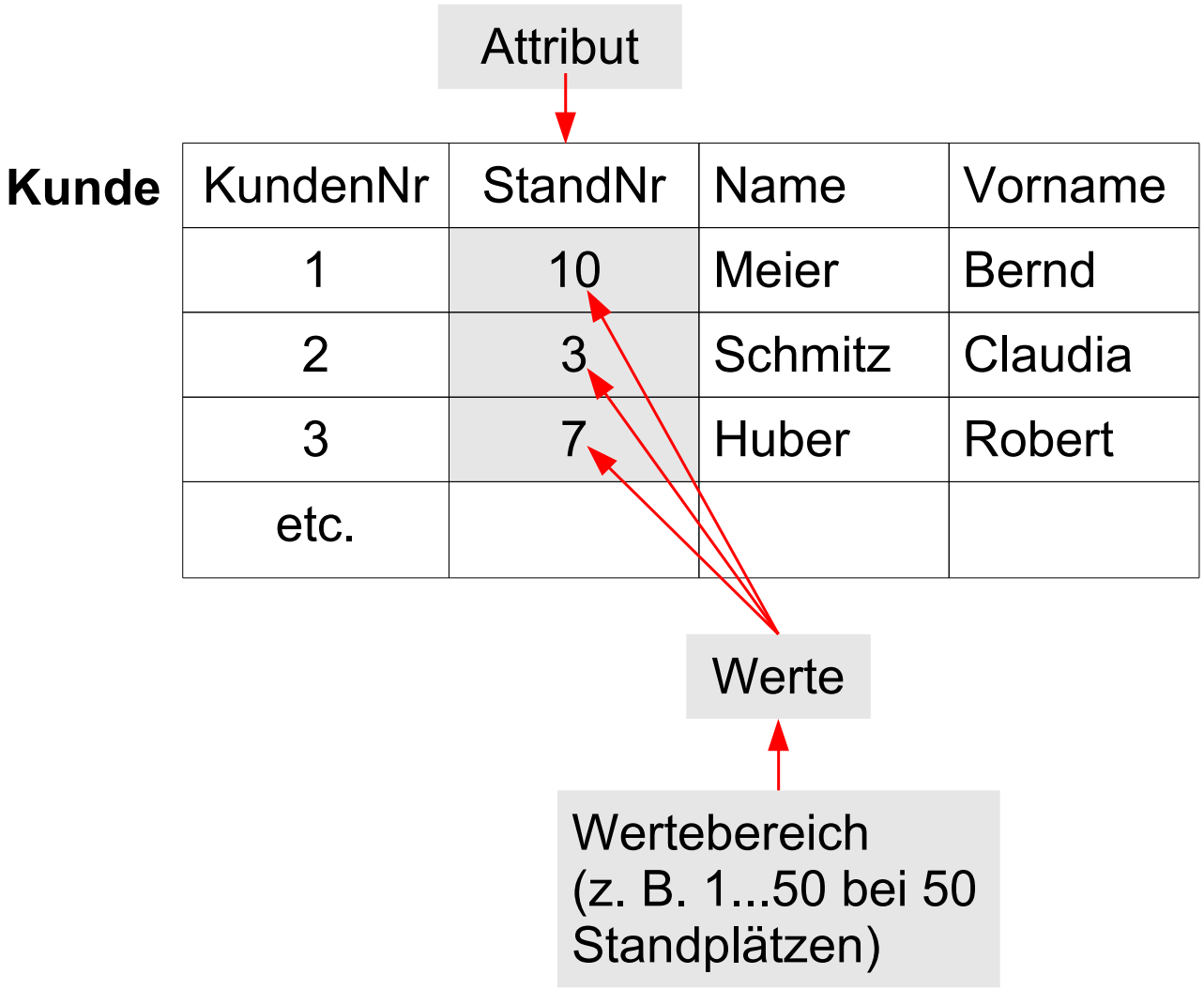
Kunde

KundenNr	StandNr	Name	Vorname
1	10	Meier	Bernd
2	3	Schmitz	Claudia
3	7	Huber	Robert
etc.			

← *Datensatz (Tupel)*

Tabelle

- Jedem Merkmal bzw. Attribut ist ein eigener *Wertebereich* zugeordnet
- Einträge in den Spalten sind Werte aus dem Wertebereich des Attributs



Tabelle

- Tabelleneinträge können mehrfach vorkommen!
- Im Beispiel tragen zwei *verschiedene* Personen den Namen "*Bernd Meier*"
- Daher ist ein Schlüsselattribut erforderlich, das jeden Datensatz eindeutig bestimmt
- Hier ist es das Attribut *KundenNr*

Kunde

KundenNr	StandNr	Name	Vorname
1	10	Meier	Bernd
2	3	Schmitz	Claudia
3	7	Meier	Bernd
etc.			

Schlüsselattribut

- Ein Schlüsselattribut oder kurz: *Schlüssel* ist ein Merkmal oder eine *minimale Merkmalskombination*, die es erlaubt, die Datensätze einer Tabelle eindeutig zu identifizieren
  - Minimal bedeutet hier, dass kein Schlüsselattribut weggelassen werden kann, ohne die Identifizierungseigenschaft zu zerstören
- Innerhalb einer Tabelle kann es mehrere Möglichkeiten geben Schlüssel zu bilden
- Diese verschiedenen Möglichkeiten heißen *Schlüsselkandidaten*
- Aus der Menge der Schlüsselkandidaten wird einer als sog. *Primärschlüssel* gewählt

Künstliche Schlüssel

Kunde

KundenNr	StandNr	Name	Vorname
1	10	Meier	Bernd
2	3	Schmitz	Claudia
3	7	Meier	Bernd
etc.			

- Ein Schlüssel kann auch auf einem künstlich eingeführten Attribut beruhen
  - Z. B. wenn es schwierig oder unmöglich ist, mit Hilfe der vorhandenen Attribute einen eindeutigen Schlüssel zu konstruieren
- Das Attribut *KundenNr* im Beispiel ist so ein künstlicher Schlüssel

Tabelle: Weitere Eigenschaften

- Jede Tabelle besitzt einen eindeutigen Tabellennamen
- Die Attributsnamen sind ebenfalls eindeutig und bezeichnen eine bestimmte Spalte mit einer definierten Eigenschaft
- Die Anzahl der Attribute bzw. Spalten ist beliebig
- Die Reihenfolge der Spalten ist beliebig aber fest
- Alle Zeilen bzw. Datensätze sind verschieden
- Die Reihenfolge der Datensätze ist beliebig

Tabelle: Manipulation

- Im Relationenmodell werden Datensätze in Form von Tabellenzeilen dargestellt
- Neue Datensätze werden folglich als Zeilen in die Tabelle eingefügt
- Überflüssige Datensätze als Zeilen aus der Tabelle gelöscht
- Änderungen an Datensätzen werden auf Tabelleneinträgen ausgeführt
- Abfragen liefern als Resultat eine Menge von Tabellenzeilen, die als Tabelle zurückgegeben werden

Definition und Manipulation

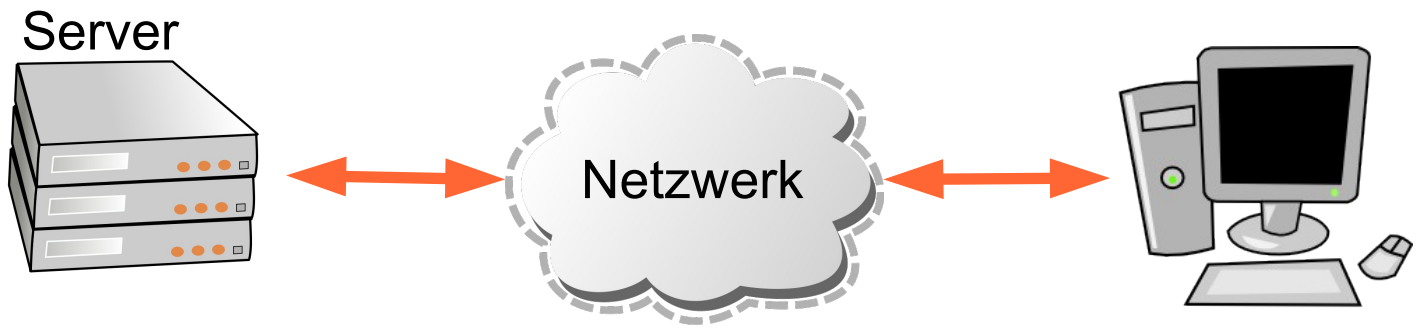
- Die wichtigste Definitions- und Manipulationssprache für das relationale Datenmodell heißt *Structured Query Language*
- Abgekürzt: SQL
- Zuerst entwickelt bei IBM in den 1970er Jahren für das Datenbanksystem System/R
- Inzwischen diverse ANSI/ISO Standards
  - SQL1 (1986) (ISO/IEC 9075), SQL89 (ISO/IEC 9075:1989)
  - SQL2 (1992) (ISO/IEC 9075:1992)
  - SQL3 (1999) (ISO/IEC 9075:1999)
  - usw. bis SQL:2019

Datenbanksysteme: Arten

- *Stand-Alone-Datenbank*
  - Desktop-Programm
  - Daten befinden sich auf einem Arbeitsplatzrechner
  - Auf die Daten kann immer nur ein Anwender über eine zugehörende Benutzeroberfläche zugreifen
  - Benutzeroberfläche heute zumeist grafisch
  - Beispiele: Microsoft Access, Open/Libre Office Base

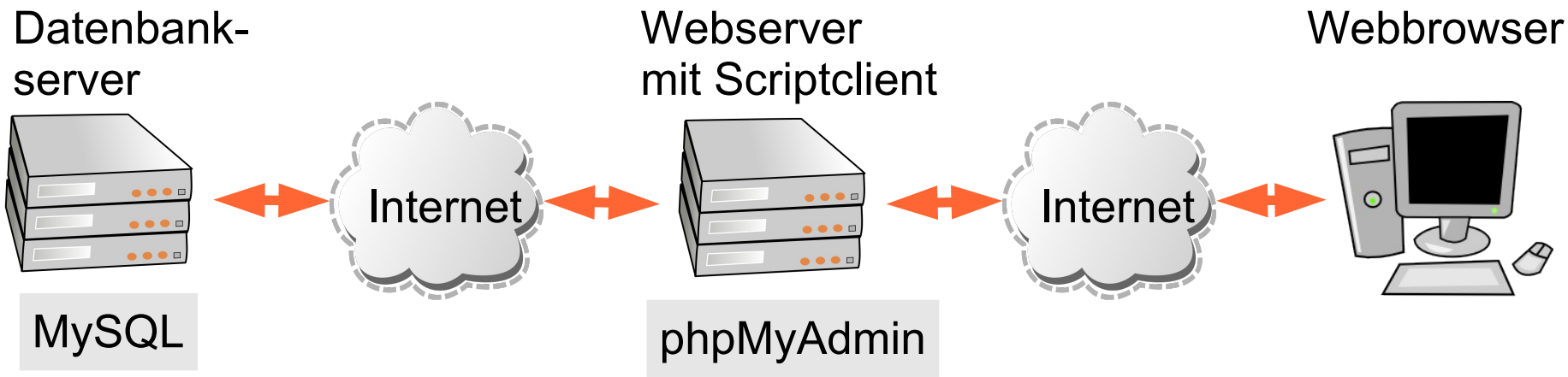
Datenbanksysteme: Arten

- *Client-Server-Datenbank*
  - Zugriff auf die Daten der Datenbank hat nur der Server
  - Anfragen werden vom Client an den Server geschickt, von diesem bearbeitet und das Resultat zurück an den Client gesendet



Client-Server-Datenbank: Webanwendung

- Zugriff auf die Datenbank mit Hilfe eines serverseitigen Scriptclients, der im Webbrowser aufgerufen wird
- Scriptclient und Datenbankserver sind häufig auf dem gleichen Rechner installiert
- Sehr häufig im Web eingesetzte Datenbank: MySQL (MariaDB)
- Im XAMPP-Paket enthalten



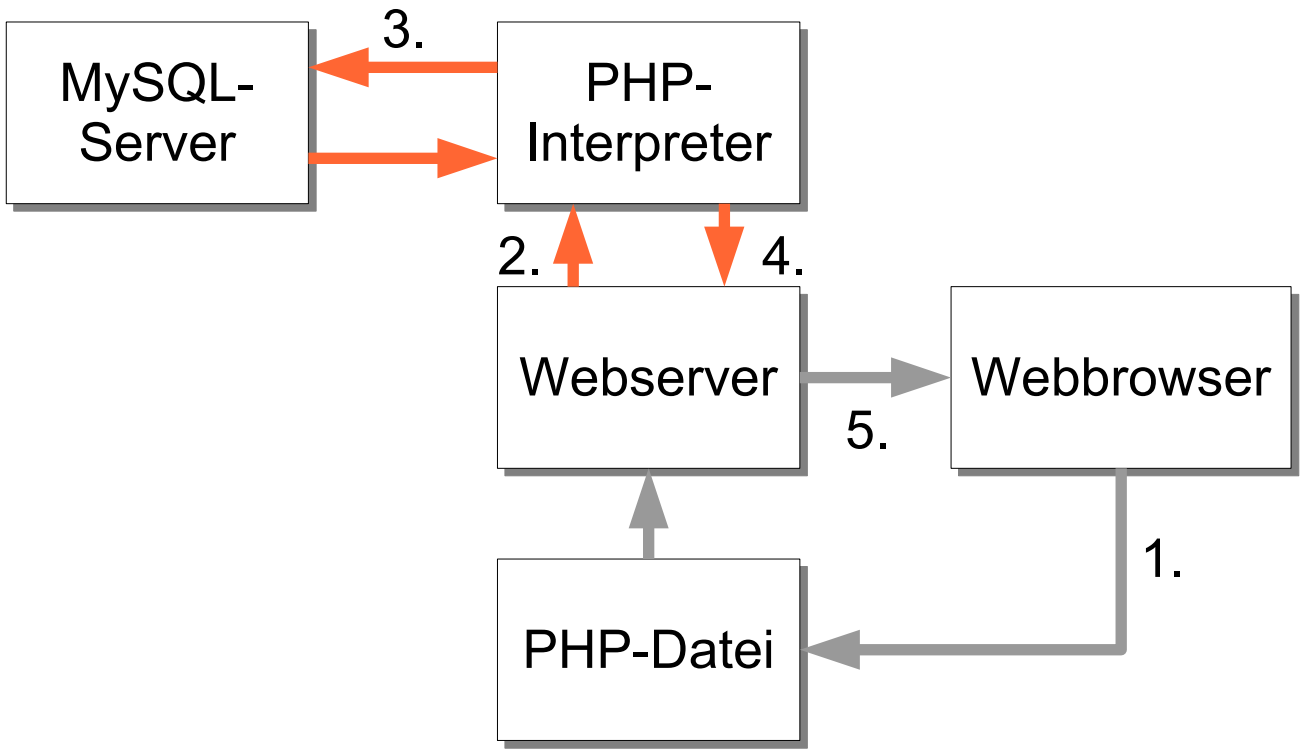
PHP und MySQL

Einführung

- PHP stellt die Klasse **`mysqli`** für den Zugriff auf MySQL-Datenbank-Server zur Verfügung
- Mit Hilfe der Methoden dieser Klasse können Anfragen an MySQL-Datenbanken formuliert und Datenbankinhalte modifiziert werden
- Auf diese Weise ist die Erstellung von dynamischen Webseiten, die ihre Inhalte aus Datenbanken beziehen sowie die Implementierung von Content Management Systemen (CMS) möglich
  - Beispiele: WordPress, TYPO3, Joomla, Drupal, etc.

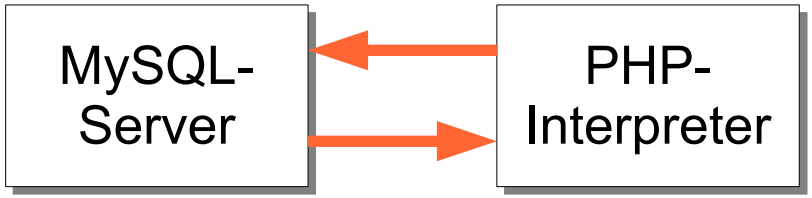
PHP und MySQL

- PHP-Script fragt mittels SQL-Befehlen Informationen aus MySQL-Datenbanken ab und integriert diese Informationen in das dynamisch erzeugte HTML-Dokument (3.)



PHP und MySQL: Phasen der Kommunikation

1. Verbindungsaufbau mit Auswahl der gewünschten Datenbank
2. MySQL-Befehl abschicken und Ergebnis entgegennehmen
3. Ergebnis auswerten
4. Weitere Anfrage? Falls Ja: Zurück zu 2
5. Verbindung beenden



- PHP stellt für jede Phase spezielle MySQL-Funktionen zur Verfügung (s. PHP Handbuch)

Verbindungsaufbau

- Verbindung zum MySQL-Server wird durch Erzeugen eines **`mysqli`**-Objekts hergestellt
- Erfordert:
  - *Name des Servers* zu dem Verbindung aufgebaut werden soll
    - **`localhost`**, falls Server auf dem gleichen Rechner wie Webserver und PHP
    - Ansonsten Name des Rechners, auf dem MySQL-Server läuft, z. B. **`xy.hs-ulm.de`**
  - *Benutzerkennung*, z. B. **`dmuserx`**
  - *Passwort* des Benutzers (entsprechend des MySQL-Accounts)
  - *Datenbankname*

Verbindungsaufbau: Beispiele

- ```
  $mysqli = @new mysqli("localhost",
                        "dmuserx",
                        "pwuserx",
                        "dmuserx");
  ```
- Instanzierung liefert Referenz (hier: `$mysqli`) für die externe Verbindung
- Referenz wird nachfolgend benutzt, um Befehle an MySQL-Server abzuschicken
- Parameter:

```
mysqli(Servername, Benutzerkennung,
       Passwort, Datenbankname);
```
```

Verbindungsaufbau

- ```
  $mysqli = @new mysqli("localhost",
                        "dmuserx",
                        "pwuserx",
                        "dmuserx");
  ```
- `@` unterdrückt automatisch generierte Fehlermeldung
- Falls keine Verbindung möglich, liefert `$mysqli->connect_error` Informationen zum Fehler
  - Z. B. bei Angabe falscher Benutzerdaten oder bei Netzwerkproblemen
- und `$mysqli->connect_errno` Nummer des Fehlers
  ```

Verbindungsaufbau: Fehlerbehandlung

```
$mysqli = @new mysqli("localhost",
                      "dmuserx",
                      "pwuserx",
                      "dmuserx");

if ($mysqli->connect_errno) {

   die ("Failed to connect to MySQL: " .
         $mysqli->connect_error);

}
```

- Fehlerinformationen in **`$mysqli`** können für die Ausgabe einer eigenen, instruktiven Fehlermeldung ausgewertet werden
  - **`die`** kombiniert Fehlermeldung mit Termination

MySQL-Befehl abschicken

- MySQL-Befehl wird als String formuliert und der Methode **`query`** übergeben

```
$queryString = "SELECT * FROM buch";
$result = @$mysqli->query($queryString);
```

- Befehls-String wird nicht mit Semikolon terminiert
- Im Falle eines Fehlers liefert **`query FALSE`**
  - Fehlerzustand entsteht z. B. dann, wenn im Beispiel die Tabelle **`buch`** nicht existiert

MySQL-Befehl abschicken

- Erläuterungen zum Fehler stehen in der Variable **error**
- Die Fehlernummer in der Variable **errno**

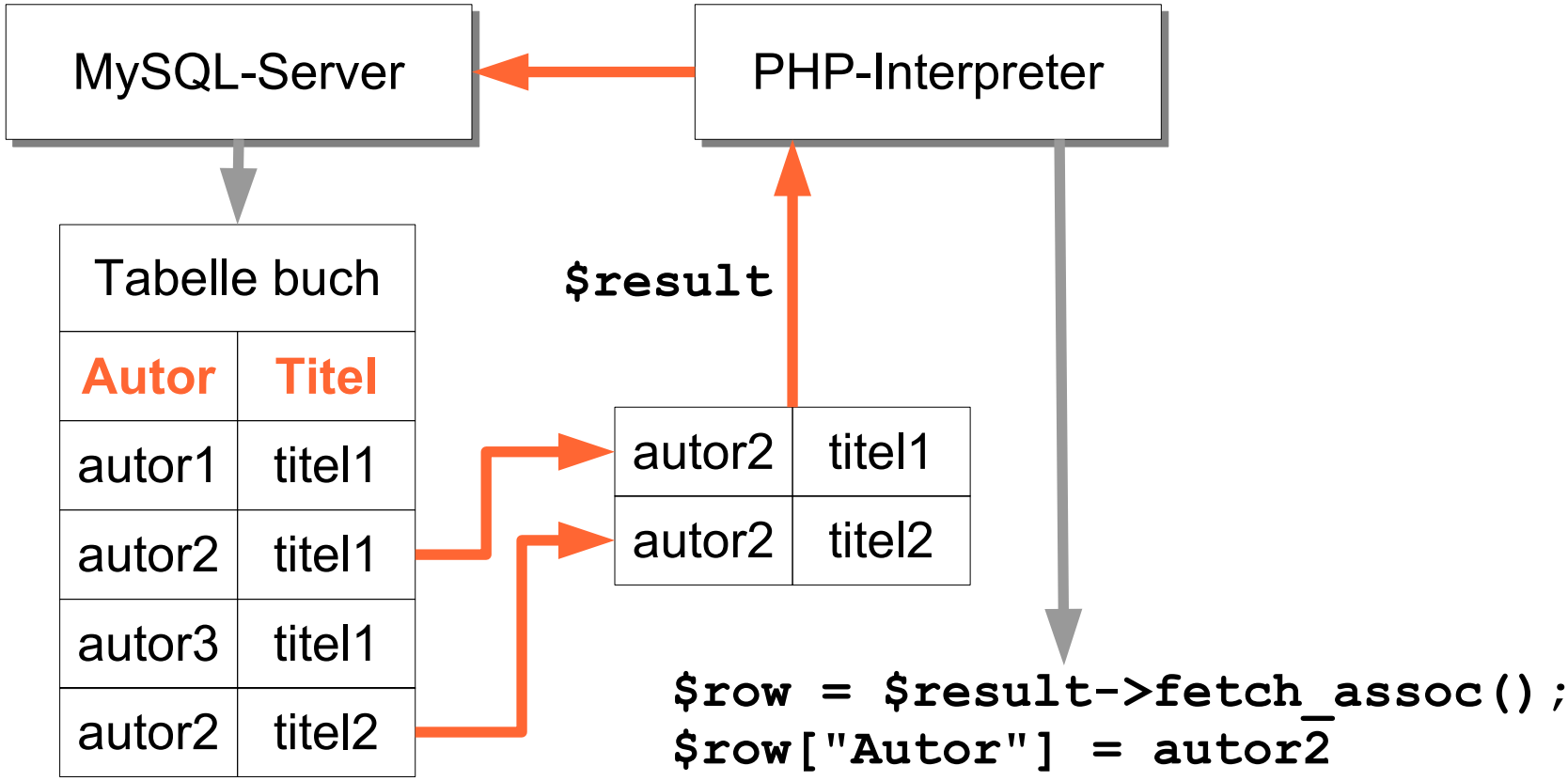
```
$queryString = 'SELECT * FROM buch';
$result = @$mysqli->query($queryString);
if (!$result) {
  die("Fehler bei der Ausführung der Abfrage:" .
      $mysqli->error);
}
```

Ergebnis auswerten

- Für **`SELECT`**, **`EXPLAIN`**, **`SHOW`** oder **`DESCRIBE`** liefert **`query`** bei Erfolg Ausgabedaten, sonst lediglich **`TRUE`**
- Ausgabedaten werden in Form einer Liste von Datensätzen geliefert
- Informationen müssen aus dieser Liste mit Hilfe einer **`fetch`**-Methode extrahiert werden
  - Im Falle eines **`SELECT`**-Befehls liefert **`query`** eine Liste von Tabellenzeilen, die die im **`SELECT`**-Befehl formulierte(n) Bedingung(en) erfüllen
  - Diese Tabellenzeilen entsprechen Datensätzen, die aus einer oder mehrerer MySQL-Tabellen entstammen

Ergebnis auswerten

```
$result =
$mysqli->query("SELECT * FROM buch WHERE Autor='autor2'");
```



Ergebnis auswerten

- **`fetch_assoc`** nimmt als Parameter die Ergebnisliste entgegen und liefert bei jedem Aufruf die jeweils nächste Tabellenzeile aus der Ergebnisliste als assoziatives Feld zurück:

```
$row_1 = $result->fetch_assoc();
$row_2 = $result->fetch_assoc();
   ⋮
$row_n = $result->fetch_assoc();
```

- Ist keine Tabellenzeile mehr in der Liste vorhanden, so liefert **`fetch_assoc`** **`NULL`** und damit **`FALSE`**

Ergebnis auswerten

- Wird **`fetch_assoc`** als Teil einer **`while`**-Schleife eingesetzt, lassen sich sukzessive alle Ergebniszeilen auslesen

```
$i = 1;
while ($row = $result->fetch_assoc()) {
  echo("Ergebnis $i: ");
  echo($row["Autor"] . ", ");
  echo($row["Titel"] . ", ");
  echo($row["Verlag"] . ", ");
  echo($row["Erscheinungsjahr"] . ", ");
  echo($row["Preis"]);

  echo("<br />");
  $i++;
}
```

Ergebnis auswerten

- Auf die Werte einer Ergebniszeile wird über den jeweiligen Spaltennamen der MySQL-Tabelle zugegriffen
- Die Spaltennamen fungieren also als Schlüssel des assoziativen Felds

```
$i = 1;
while ($row = $result->fetch_assoc()) {
  echo("Ergebnis $i: ");
  echo($row["Autor"] . ", ");
  echo($row["Titel"] . ", ");
  echo($row["Verlag"] . ", ");
  echo($row["Erscheinungsjahr"] . ", ");
  echo($row["Preis"]);

  echo("<br />");
  $i++;
}
```

Fetch-Varianten

- `fetch_assoc`
  - Liefert Datensatz als assoziatives Feld
- `fetch_row`
  - Liefert Datensatz als indiziertes Feld
- `fetch_array`
  - Liefert Datensatz als assoziatives und als indiziertes Feld

Ergebnis auswerten mit `foreach`

- Assoziatives Feld mittels **`foreach`** auslesen

```
$i = 1;
while ($row = $result->fetch_assoc()) {
  echo("Ergebnis $i: ");

  foreach ($row as $key => $val) {
    if ($key != "ID") echo($val);
    if ($key != "Preis" && $key != "ID") echo(", ");
  }

  echo("<br />");
  $i++;
}
```

Ergebnis auswerten mit `implode`

- Feldelemente mittels **`implode`** verbinden

```
$i = 1;

while ($row = $result->fetch_assoc()) {

  echo("Ergebnis $i: ");

  // Datenbank-ID mit substr von Ausgabe ausklammern
  // Einträge mit ", " voneinander separieren

  echo(substr(implode(", ", $row),3) . "<br />");

  $i++;
}
```

Speicher freigeben

- Nach Abschluss der Auswertung sollte der von der Ergebnisliste belegte Speicher mit Hilfe von **`close`** freigegeben werden

  Beispiel: **`$result->close();`**

- Insbesondere wichtig, wenn Ergebnisliste viel Hauptspeicher belegt
- Anderenfalls erfolgt die Freigabe automatisch nach Beendigung des PHP-Scripts
- Freigabe des **`mysqli`**-Objekts:

```
$threadID = $mysqli->thread_id;

$mysqli->kill($threadID); // Closes TCP connection

$mysqli->close(); // Close MySQL object
```

Cookies und Sessions

Cookies

Überblick

- Webauftritte stellen häufig eigenständige Anwendungen dar
- Wie jede andere Anwendung auch, benötigen diese eine Möglichkeit, anfallende Daten zu speichern
  - Z. B. Benutzerdaten und Anwendungseinstellungen
- Diese Daten müssen beim nächsten Aufruf wieder eingelesen werden können
- Das Speichern von Daten auf dem Server erfordert eine Benutzerverwaltung
- Bei geringem Datenvolumen ist es einfacher, diese Daten dort zu speichern, wo sie anfallen:
  Auf dem Clientrechner

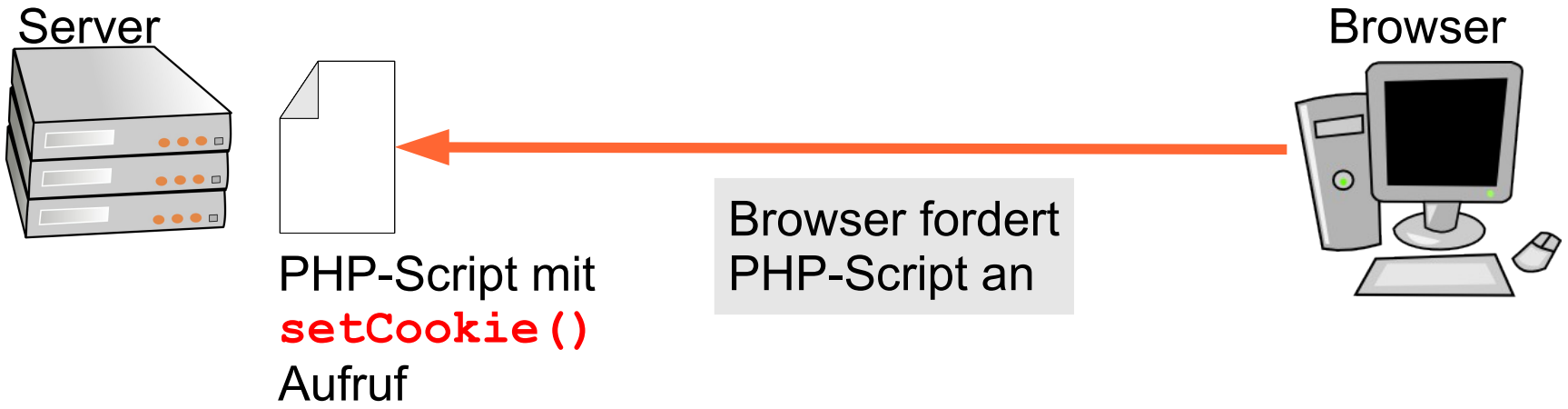
Cookies

- Cookies erlauben es, kleine Datenmengen auf dem Clientrechner zu speichern
- Ein Cookie ist ein benannter Container in dem ein Datenwert gespeichert werden kann
- Vergleichbar mit einer Variable
  - Der Variablenwert wird aber nicht im Hauptspeicher abgelegt, sondern in einer Datei
- Cookies können aus PHP-Scripts heraus auf dem Clientrechner abgelegt werden
  - Die Speicherung erfolgt in einer speziellen Datei des Webbrowsers

Cookies

- Der Cookie ist mit der Website assoziiert, die den Cookie gesetzt hat
- Alle zukünftigen Anforderungen von dieser Site enthalten den Cookie, solange dieser gültig ist
  - Der Inhalt des Cookies wird automatisch vom Webbrowser dem HTTP-Header hinzugefügt
- Der PHP-Interpreter übergibt die Daten des übertragenen Cookies an das zur Webseite gehörende Script
- Andere Websites können nicht auf die Daten des Cookies zugreifen

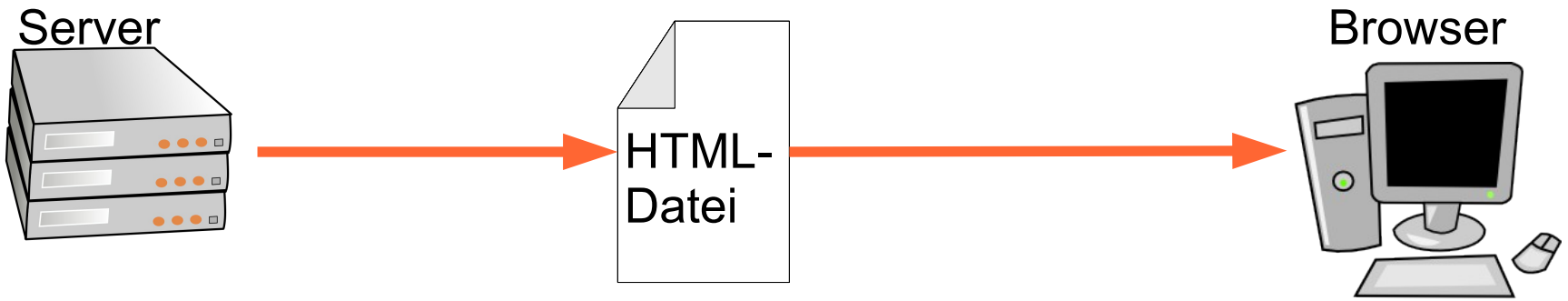
Lebenszyklus eines Cookies



- `setCookie()` definiert einen mit den HTTP-Header-Informationen zu übertragenden Cookie, welcher die zu speichernden Daten enthält, z. B. eine Benutzer-ID oder einen Zugriffszähler

- Cookies müssen vor irgendwelchen anderen Ausgaben des Skriptes, inklusive **<html>**- oder **<head>**-Tags, gesendet werden: Einschränkung des HTTP-Protokolls

Lebenszyklus eines Cookies

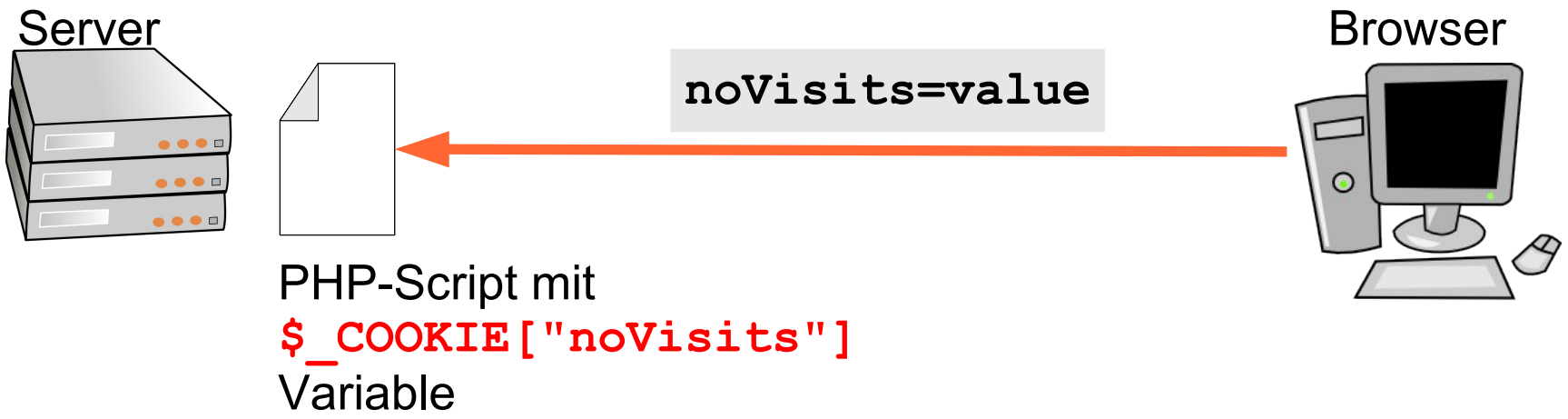


- Server antwortet mit HTML-Datei
- Diese enthält den Cookie-Eintrag im HTTP-Header
- Eintrag besteht aus Name und Wert des Cookies
- Beispiel: **`noVisits = value`**

Lebenszyklus eines PHP-Cookies

- Browser wertet HTTP-Header aus und extrahiert Cookie-Daten
- Daten werden zusammen mit einem Hinweis auf die Ursprungs-Website in einer browserinternen Cookie-Datenbank abgelegt

Lebenszyklus eines PHP-Cookies



- Browser fordert erneut Seite von Webserver an
  - Anforderung enthält die Cookie-Daten im HTTP-Header
- PHP-Interpreter erzeugt Eintrag im globalen Feld `$_COOKIE`
  - Im PHP-Script wird das entsprechende Feldelement ausgewertet

setCookie()

- **boolean setCookie (string name [, string value [, int expire [, string path [, string domain [, int secure]]]]])**
  - Liefert **false**, falls im PHP-Script schon eine Ausgabe vor dem Aufruf erfolgte
- **name**: Name des Cookies
- **value**: Wert des Cookies
- **path, domain**: Erweiterung des Cookie-Versandes auf angegebene Domain bzw. angegebenes Server-Verzeichnis
- Außer **name** sind alle Argumente in den eckigen Klammern optional

`setCookie()`

- **`expire`**: Zeitpunkt, wann der Cookie verfallen soll
  - Ab diesem Zeitpunkt hören die Cookie-Daten auf zu existieren
  - Gemessen in Sekunden ab dem 1. Januar 1970
  - Ist der Parameter nicht gesetzt, verfällt der Cookie am Ende der Browsersession
- Beispiel:
  - **`time()+60`**
    - Dieser Cookie verfällt nach 60sec.
    - **`time()`**: Gibt die seit dem 1. Januar 1970, 00:00:00h GMT bis jetzt vergangene Zeit in Sekunden zurück

Beispiele

- Cookie setzen:

```
setCookie("noVisits",$noVisits,time()+3600);
/* Cookie verfällt in 1 Stunde */
```

- Cookie-Wert auslesen:

```
echo($_COOKIE["noVisits"]);
```

- Cookie mit Verfallsdatum löschen

```
setCookie("noVisits","",time()-3600);
```

  - Verfallsdatum in der Vergangenheit festlegen

Abfragen, ob Cookie gesetzt ist

```
if (isset($_COOKIE["noVisits"])) {
  $visits = $_COOKIE["noVisits"] + 1;
}
else {
  $visits = 1;
}
setCookie("noVisits",$visits,time()+3600);
```

Ausgabepufferung

- Die Ausgabe eines PHP-Scripts kann gepuffert und somit verzögert werden
- Auf diese Weise ist das Setzen von Cookies auch nach Ausgaben aus dem Script möglich
- Beispiel:

```
<?php
ob_start(); // Aktiviere Ausgabepufferung
echo ("Hello");
setcookie ("cookiename", "cookiedata");
ob_end_flush(); // Leere Ausgabepuffer
?>
```

Nachteile

- Cookie-Spezifikation RFC 6265 (2011)
- Browser *sollen* die folgenden Mindestwerte einhalten:
  - Cookie-Größe: Mindestens 4096 Bytes
  - Mindestens 50 Cookies pro Domain
  - Mindestens 3000 Cookies insgesamt
- Cookies also in ihrer Größe beschränkt
- Alte Cookies können willkürlich gelöscht werden, sobald eine der angegebenen Maximalzahlen überschritten wird
  - Websites mit hohem Cookie-Aufkommen können Cookies anderer Sites aus dem Speicher verdrängen
- Benutzer können Cookies abschalten!

Sessions

Überblick

- Erlauben ebenfalls das Erhalten von Variablenwerten zwischen den Aufrufen einer PHP-Datei
- Anstatt auf dem Clientrechner werden die Daten indes temporär auf dem *Server* gespeichert
- Sessions unterliegen daher hinsichtlich der Speicherung von Daten nicht den Restriktionen von Cookies
- Dafür sind die Daten nur für die Zeitdauer der Session gültig

Überblick

- Session wird mit Hilfe einer Session-Identifikationsnummer (Session-ID) identifiziert
- Diese wird zu Beginn der Session vergeben
- Alle zur Session gehörenden Daten werden dieser ID zugeordnet und später mittels dieser ID im lokalen Speicherbereich des Servers identifiziert

Session-ID

- Session-ID wird als Cookie auf dem Clientrechner gespeichert
  - Browser überträgt ID fortan automatisch an Server wenn zugehörende Seite angefordert wird
- Falls Cookies deaktiviert sind, wird Session-ID automatisch beim Aufruf der PHP-Datei an den URL angehängt und dadurch übergeben
- Sessions funktionieren daher immer!

Session: Lebensdauer

- Session wird automatisch beendet:
  - wenn Besucher Website verlässt,
  - Browserapplikation beendet wird,
  - durch expliziten Befehl im PHP-Script
  - oder durch Timeout auf dem Server
- Nach Beendigung der Session werden temporäre Daten auf dem Server automatisch gelöscht

Eine Session anlegen

- Eine neue Session wird durch Aufruf der PHP-Funktion

  `session_start()`

  eingerichtet

- Sofern beim Aufruf der PHP-Datei eine Session-ID übermittelt wurde, wird die unter dieser ID gesicherte Session durch den Aufruf wieder restauriert

Variablen in der Session ablegen

- Sessionvariablen werden im vordefinierten Feld **`$_SESSION[]`** abgelegt

- Beispiel 1: Anlegen einer neuen Sessionvariable vom Typ Integer mit dem Namen "**`zaehler`**":

```
$_SESSION["zaehler"] = 0;
```

- Beispiel 2: Anlegen einer neuen Sessionvariable vom Typ Array mit dem Namen "**`strArray`**":

```
$_SESSION["strArray"] = array();
```

Den Wert einer Sessionvariablen ändern

- Sessionvariablen können wie normale Variablen benutzt werden
- Beispiel: Ändern des Wertes der Sessionvariable "**zaehler**":

```
$_SESSION["zaehler"]++;
```

- Beispiel: Ändern der Sessionvariable "**strArray**":

```
$_SESSION["strArray"][] = "Neuer String";
```

Abfragen, ob eine Variable schon existiert

- Eine neue Variable wird nur dann angelegt, wenn sie noch nicht existiert
- Hierfür muss die Existenz der Variablen geprüft werden
- Dies erfolgt mit Hilfe der PHP-Funktion **`isset()`**
- Beispiel:

```
if (!isset($_SESSION["zaehler"]))
  $_SESSION["zaehler"] = 0;
else
  $_SESSION["zaehler"]++;
echo($_SESSION["zaehler"]);
```

Eine Sessionvariable oder Session löschen

- Sessionvariablen und allgemein Variablen können mit Hilfe der PHP-Funktion **`unset()`** gelöscht werden

- Beispiel:

```
unset($_SESSION["zaehler"]);
```

- Eine ganze Session kann durch Aufruf von **`session_destroy()`** gelöscht werden

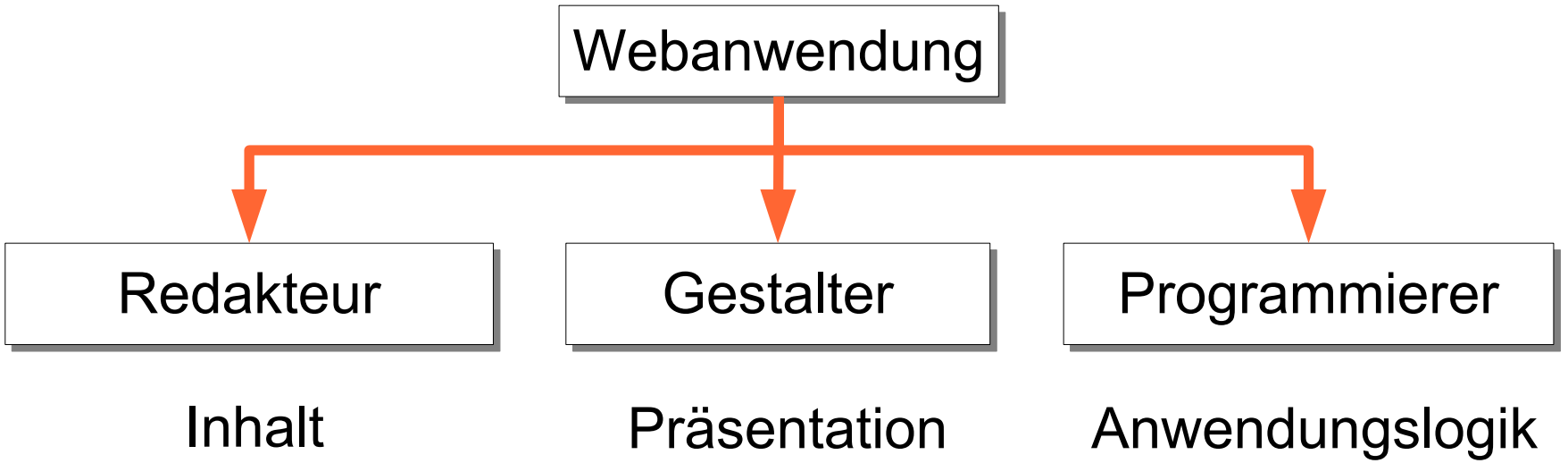
Sicherheitsprobleme

- Session-ID kann durch Angreifer entwendet und weiterverwendet werden
  - Session ist nicht an einen bestimmten Rechner gebunden
- IP-Adresse des Rechners, von dem aus die Session gestartet wurde, in der Session speichern
  - `$_SESSION["remAdr"] = $_SERVER['REMOTE_ADDR'];`
- Im PHP-Script bei Aufruf IP-Adresse überprüfen
  - Vergleich der gespeicherten Adresse mit der Adresse des Aufrufers
- Erhöht Sicherheit

Templates

Webanwendung

- Webanwendungen erfordern die Zusammenarbeit von Spezialisten unterschiedlicher Disziplinen



- Die Möglichkeit eines asynchronen Arbeitens an den HTML-, CSS- und JavaScript-Inhalten ist daher wünschenswert

Dynamische Webseite

- Auf der Grundlage von PHP erzeugte dynamische Webseiten können ohne passende Maßnahmen eine Mischung aus HTML-, CSS- und Programmcode darstellen
- Inhalt, Logik und Gestaltung sind nicht richtig voneinander getrennt
- Diese Trennung ist allerdings Voraussetzung für unabhängiges Arbeiten

MVC

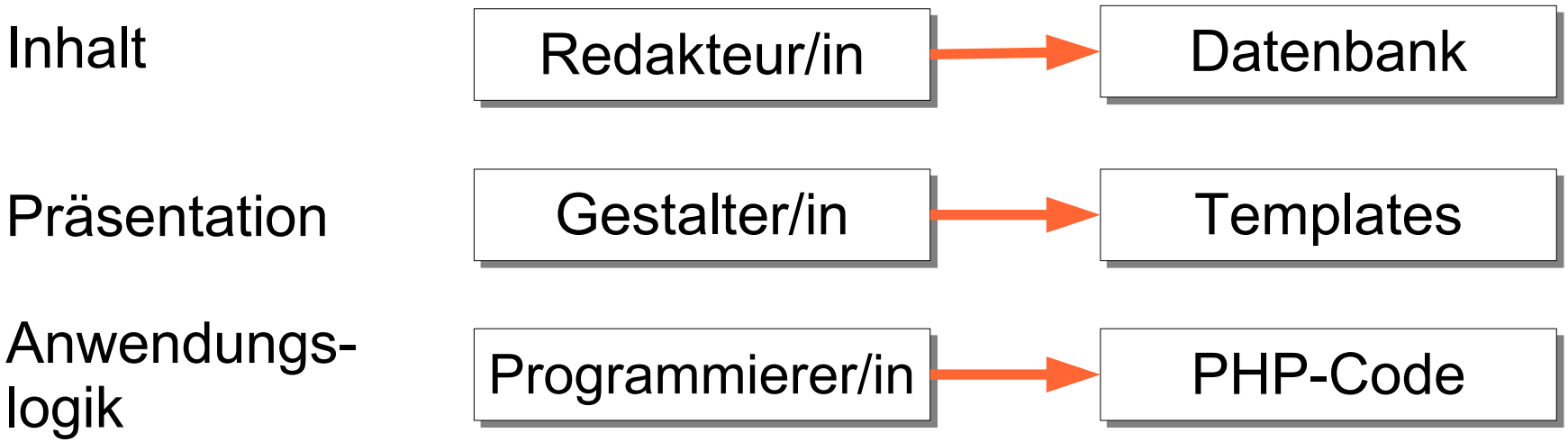
- MVC-Designmuster
- Model-View-Controller („Modell-Präsentation-Steuerung“)
  - Eine Anwendung besteht aus dem Datenmodell (Modell), dem Präsentationsteil (View) und der Programmsteuerung (Controller), die Model und View miteinander verbindet
- Impliziert die Möglichkeit, diese Programmteile unabhängig voneinander zu entwickeln und zu pflegen
- Die verschiedenen Codebestandteile werden hierfür soweit wie möglich voneinander getrennt

Templates (1)

- Eine Trennung zwischen Layout und Inhalten wird möglich mit Hilfe von Vorlagen bzw. Templates
- Templates erlauben die Erstellung von Schablonen, die das Layout der zukünftigen Webseiten besitzen
  - Templates unterstützen also ein einheitliches Layout
- Die Schablonen enthalten Platzhalter, die bei der dynamischen Erzeugung der Webseite durch Inhalte ersetzt werden
- Die Inhalte stammen zumeist aus Datenbanken

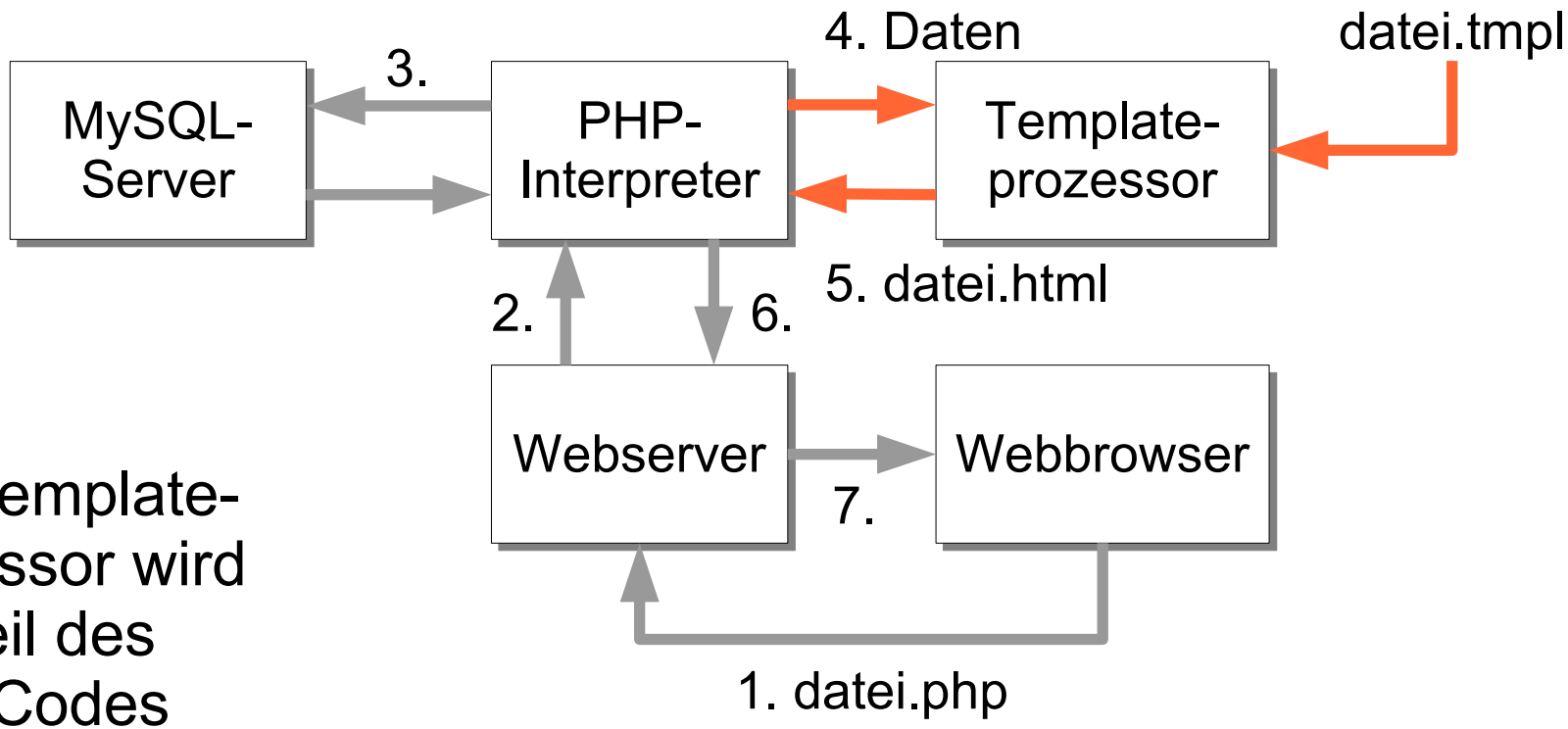
Templates (2)

- Die Trennung von Layout und Inhalten ermöglicht asynchrones Arbeiten



Templateprozessor

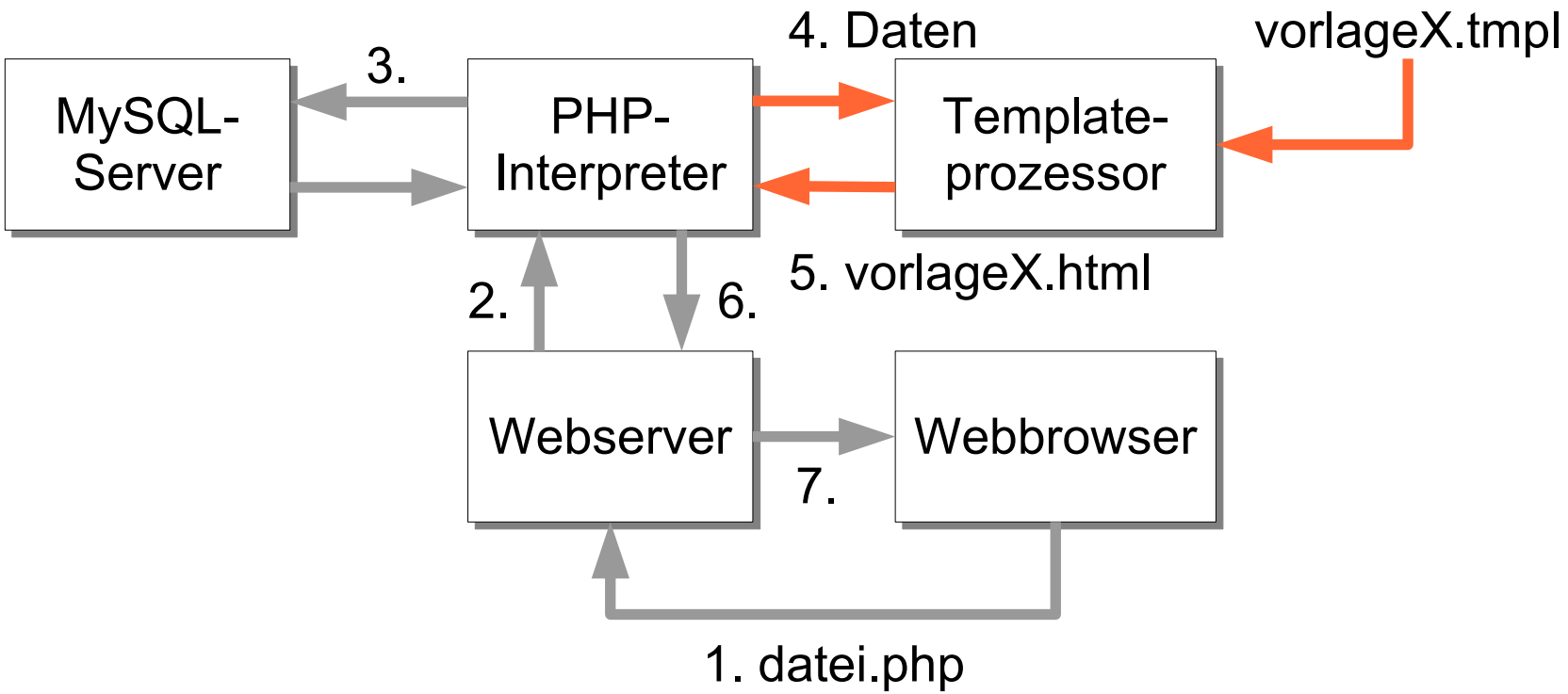
- Das Zusammenführen von Vorlagendatei und Inhalten aus Datenbanken geschieht durch den Templateprozessor



- Der Template-prozessor wird als Teil des PHP-Codes aufgerufen

Mehrere Vorlagen

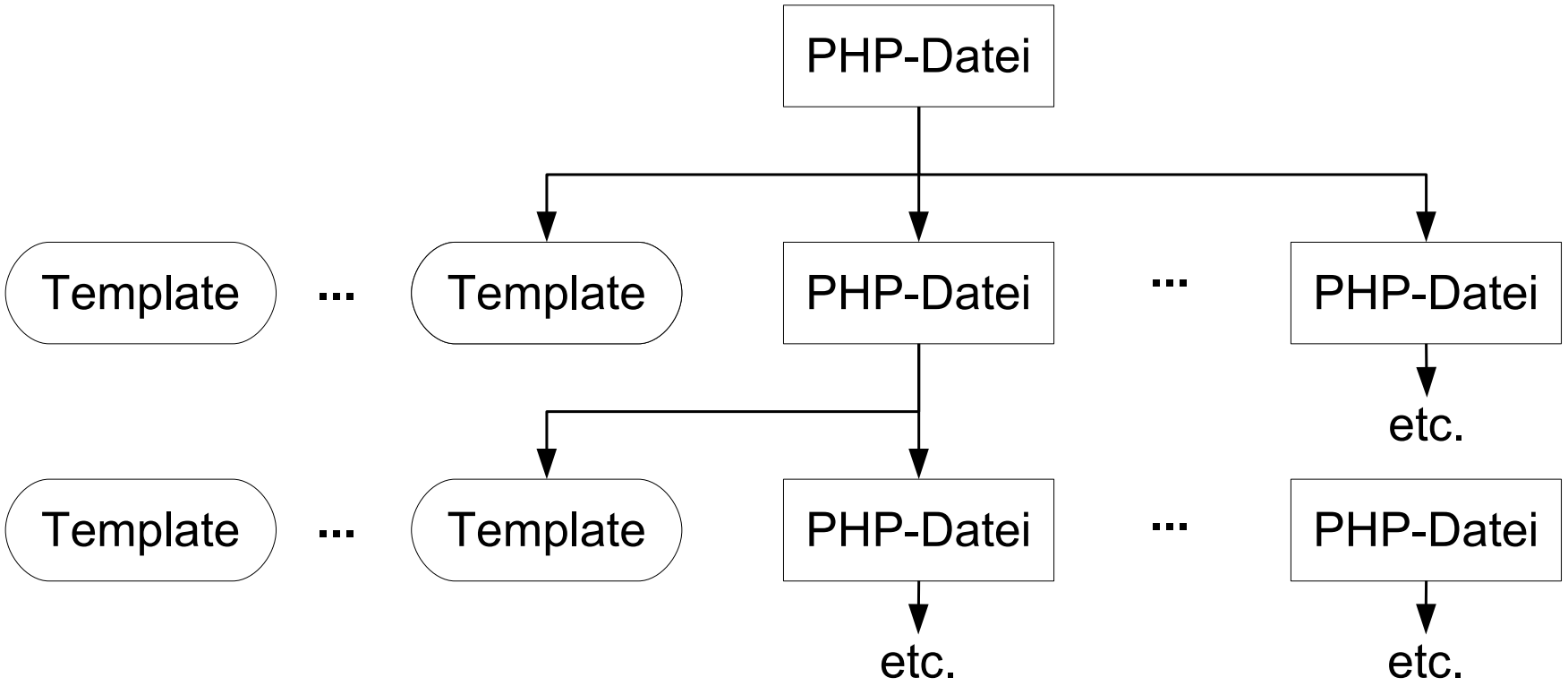
- Pro Webseite können durchaus mehrere Vorlagen zum Einsatz kommen



- Es ist auch möglich, Vorlagen hierarchisch zu staffeln

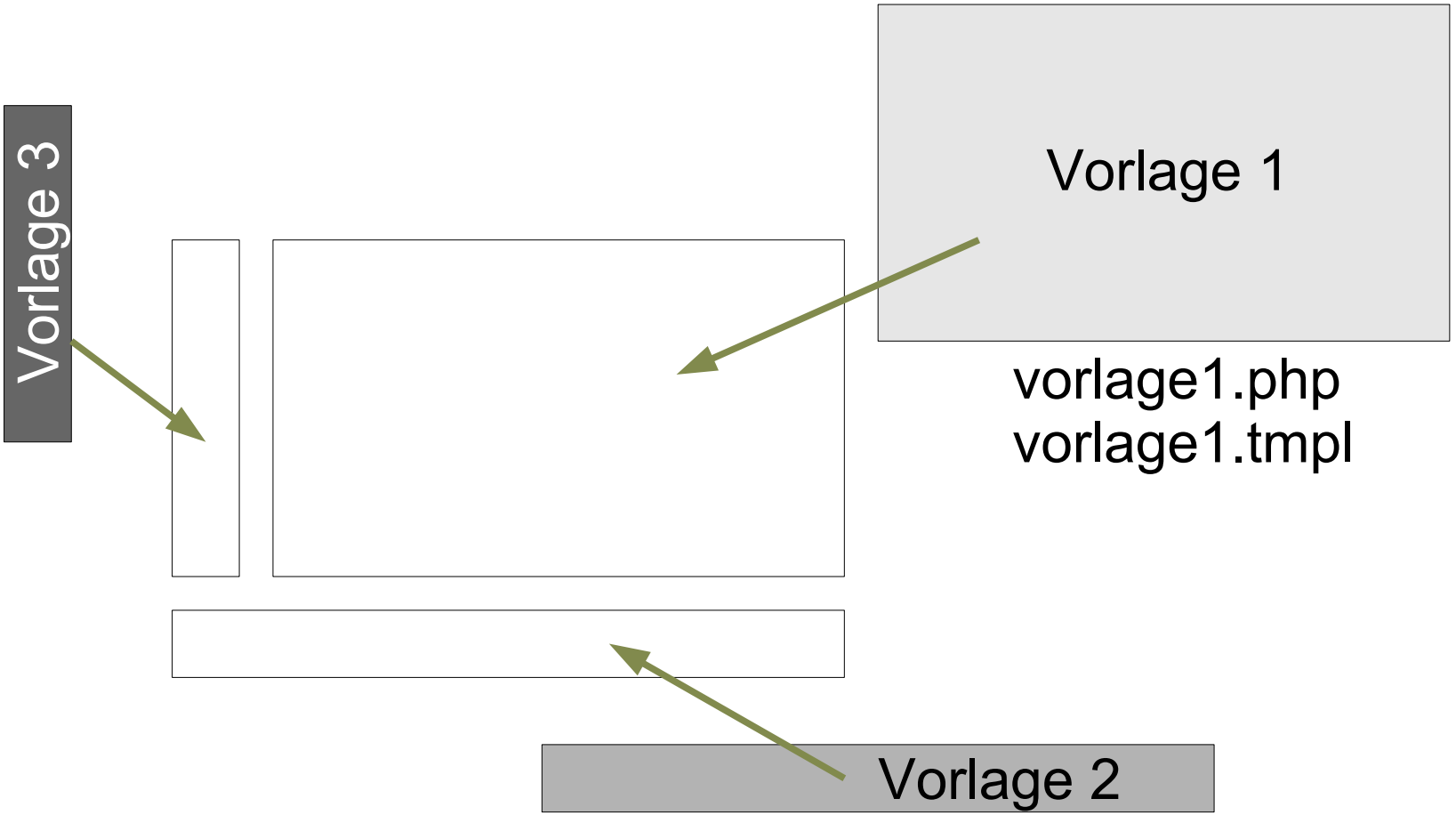
Hierarchische Vorlagen

- PHP-Code greift auf Vorlage und weitere PHP-Dateien zu, die wiederum auf Vorlagen zugreifen



Seitenaufteilung

- Das Layout der einzelnen Boxen bzw. Frames kann unterschiedlichen Vorlagen entstammen



Templateprozessor: Arten

- TP häufig nicht vorinstalliert auf dem Webserver
- TP in Maschinensprache programmiert (native Code)
  - Schnell
  - Erfordert Installation als PHP-Extension-DLL
  - Beispiel: PHP-Templates
- TP in PHP programmiert
  - Langsamer als native Code
  - Dafür einfache Installation
  - Beispiele: XTemplate, Smarty, Twig

Beispiel

- **`webseite.html`**: Inhalte und Layout nicht getrennt

```
<html>
<head>
<title>Meine Webseite</title>
</head>
<body>
<hr>
<p>Aktuelles: Habe heute Geburtstag.
              Geschenke willkommen.</p>
<hr>
<p>Mein Text.</p>
</body>
</html>
```

Lösung mit XTemplate

- **webseite.tmpl**:

```
<!-- BEGIN: main -->
<html>
<head>
<title>{titel}</title>
</head>
<body>
<hr>
<p>Aktuelles: {aktuelles}</p>
<hr>
<p>{text}</p>
</body>
</html>
<!-- END: main -->
```

- Zu bearbeitender Bereich wird in benannten Kommentar eingeschlossen
- Platzhalter werden in geschweiften Klammern notiert
- Syntax für Platzhalter bei Prozessoren häufig an eigene Bedürfnisse anpassbar

PHP-Script (1)

- Templateprozessor einbinden:

```
require_once('xtemplate.class.php');
```

- Template öffnen:

```
$xtpl = new Xtemplate('webseite.tmpl');
```

  - Templatecode steht in **`webseite.tmpl`**
  - Instanzierung liefert ein Objekt (hier: **`$xtpl`**), welches Grundlage der weiteren Verarbeitungsoperationen ist

PHP-Script (2)

- Variablen Werte zuweisen mit **assign**:

```
$xtpl->assign('titel', "Meine Webseite");
$xtpl->assign('aktuelles', "Habe heute
    Geburtstag. Geschenke willkommen.");
$xtpl->assign('text', "Mein Text.");
```

- Die Daten bzw. Einträge entstammen üblicherweise einer Datenbank

PHP-Script (3)

- Benannten Bereich (hier: **`main`**) in Template verarbeiten mit **`parse`**:

```
$xtpl->parse('main');
```

- Resultierendes HTML ausgeben mit **`out`**:

```
$xtpl->out('main');
```

Templateprozessor

```
<html>
<head>
<title>{titel}</title>
</head>
<body>
<hr>
<p>Aktuelles:</p>
<p>{aktuelles}</p>
<hr>
<p>{text}</p>
</body>
</html>
```

- Templateprozessor ersetzt Platzhalter durch zugewiesene Werte

```
"Meine Webseite";
```

```
"Habe heute
Geburtstag...";
```

```
"Mein Text.";
```

Tabellen

- Relationale Datenbanken liefern Resultate von Anfragen als Tabellen
  - Resultate einer Suchmaschine
  - Gefundene Einträge eines bestimmten Typs (z. B. Bücher eines bestimmten Autors)
  - etc.
- Diese Daten sind häufig ebenfalls als Tabellen oder Listen in eine anzuzeigende HTML-Seite zu integrieren
- Dabei ist typischer Weise im Vorhinein die Anzahl der anzuzeigenden Zeilen nicht bekannt, da abhängig vom Anfrageergebnis

Variable Zeilenzahl

- Für die Anzeige einer variablen Anzahl von Zeilen wird ein sogenannter **iterierter Templatekontext** benötigt
- Dieser wird im Template durch einen eigenen benannten **`BEGIN-END`**-Bereich markiert
- Bei der Verarbeitung des Templates wird dieser Bereich entsprechend der Anzahl vorliegender Daten mehrfach abgearbeitet

Beispiel: Template und PHP-Script

```
<!-- BEGIN: main -->
<html><body>
<table>
<!-- BEGIN: row -->
<tr>
  <td> {col1} </td>
  <td> {col2} </td>
</tr>
<!-- END: row -->

</table>
</body></html>
<!-- END: main -->
```

```
require_once('xtemplate.class.php');
$xtpl = new XTemplate('bsptempvl.tmpl');

// Daten in das Template schreiben
for ($i=1; $i<=10; $i++) {
  $xtpl->assign("col1","val1_$i");
  $xtpl->assign("col2","val2_$i");
  $xtpl->parse('main.row');
}
$xtpl->parse('main');
$xtpl->out('main');
```

Verschachtelte iterierte Templatekontexte

- Nicht nur die Anzahl der Zeilen, sondern auch die Anzahl der Tabellenspalten kann variabel sein
- In diesem Fall ist ein verschachtelter iterierter Templatekontext erforderlich
- Beispiele:

```
<!-- BEGIN: row -->
<tr>
  <!-- BEGIN: col -->
  <td> {col} </td>
  <!-- END: col -->
</tr>
<!-- END: row -->
```

```
<!-- BEGIN: row -->
<tr>
  <td> {col1} </td>
  <!-- BEGIN: col -->
  <td> {col2} </td>
  <!-- END: col -->
</tr>
<!-- END: row -->
```

Beispiel: Template und PHP-Script

```
<!-- BEGIN: main -->
<html><body>
<table>

<!-- BEGIN: row -->
<tr>
  <td> {col1} </td>
  <!-- BEGIN: col -->
  <td> {col2} </td>
  <!-- END: col -->
</tr>
<!-- END: row -->

</table>
</body></html>
<!-- END: main -->
```

```
// Templateprozessor laden
// Template öffnen

for ($i=1; $i<=10; $i++) {

 $xtpl->assign("col1","R$i");

 for ($j=1; $j<=5; $j++) {

   $xtpl->assign("col2","$i:$j");
   $xtpl->parse('main.row.col');

 }

 $xtpl->parse('main.row');

}

$xtpl->parse('main');
$xtpl->out('main');
```

Datenbankentwurf

Beispiel: Buchdatenbank

ID	Autor	Titel	Verlag	Erscheinungsjahr	Preis
16	Joanne K. Rowling	Harry Potter und der Feuerkelch	Carlsen	2001	22.50

- Alle Daten werden in einer gemeinsamen Tabelle gehalten
- Erhebliche Redundanzen vorhanden
  - Name der Autorin bzw. des Autors tritt potentiell mehrfach auf, ebenfalls der Name des Verlags
  - Bei Änderung des Autorennamens oder Verlags sind mehrere Zeilen zu ändern
- Erhaltung der semantischen Integrität unter diesen Bedingungen erschwert
- Weiteres Problem: Bei Löschen aller Bucheinträge eines Autors geht dessen Name und damit die Information zum Autor verloren

Normalformen

- Zur Vermeidung derartiger Probleme sollten die in der Datenbank gespeicherten Informationen durch Transformation in eine Reihe von *Normalformen* über mehrere Tabellen verteilt werden
- Jede Normalform reduziert die vorhandenen Redundanzen und hilft so die Integrität der Datenbank zu erhalten
- Es gibt sechs Normalformen, von denen die ersten drei die wichtigsten sind
- Diese werden im weiteren erläutert

Datenbanken: Normalformen

Notation

- Aus Platzgründen werden im weiteren Verlauf Tabellen in Form von Relationen notiert
- Eine Tabelle für Kundennamen

KID	Name	Vorname
1	Schmitt	Bernd
2	Maier	Maria
...	...	...

- steht dann z. B. für die Relation

  Kunde (KID, Name, Vorname)

  - Schlüsselattribute werden unterstrichen

Beispielszenario: Eine Internet-Tauschbörse

- Anbieter können unterschiedliche Artikel zum Tausch einstellen
- Bieter können zu den eingestellten Artikeln Angebote abgeben
- Anbieter kann Angebot annehmen oder ablehnen
- Erforderliche Relationen:
  - Ware
    - Speichert Informationen zu den angebotenen Waren
  - Tauschen
    - Speichert Informationen zu den Tauschgeschäften
- Diese Relationen bilden zusammen das konzeptuelle Schema
  - D. h. die Gesamtheit aller erfassten Daten

Relation "Ware"

- Ware (Datum, KID, Name, Adresse, Bezeichnung, Kategorie, Beschreibung, Anzahl, Alter, Art)
  - Kann für Anbieter und Bieter eingesetzt werden
  - Beispieldatensatz:
  - ("31.01.25, 10:57", "1", "Max Muster", "Ulmstr. 2, 89075 Ulm, 0176...", "TV", "Elektronik", "LG+ LCD 50'", "1", "5 Jahre", "Biete")

Tausch

- Um einen Tausch real stattfinden zu lassen, wird eine Relation benötigt, die eine Beziehung zwischen dem Anbieter und dem Bieter herstellt
  - In dieser Relation können ebenfalls zusätzliche Parameter, wie z. B. der Status des Tausches, erfasst werden
- Relation "Tauschen":
  - Tauschen (DatumA, AnbieterID, DatumB, BieterID, Status)
- Um Eindeutigkeit des Tausches zu erreichen, sind vier Schlüsselattribute erforderlich
  - Sowohl Anbieter, als auch Bieter können mehrere Tauschgeschäfte gleichzeitig laufen haben
- Status: Codiert, ob Tausch akzeptiert wurde oder nicht

Nicht atomare Wertebereiche

- Beispieldatensatz:
- ("31.01.25, 10:57", "1", "Max Muster", "Ulmstr. 2, 89075 Ulm, 0176...", "TV", "Elektronik", "LG+ LCD 50'", "1", "5 Jahre", "Biete")
- Diverse Attributwerte haben zusammengesetzten Charakter
- Derartige Zusammensetzungen erschweren Datenbankabfragen

Nicht atomare Wertebereiche: Probleme (1)

- Beispieldatensatz:
- ("31.01.25, 10:57", "1", "Max Muster", "Ulmstr. 2, 89075 Ulm, 0176...", "TV", "Elektronik", "LG+ LCD 50'", "1", "5 Jahre", "Biete")
- Beispiele:
  - Suche nach dem Namen „Klaus“ liefert alle Einträge mit diesem Vor- und Nachnamen
  - Sortieren nach Nachnamen oder Orten
  - Sortieren nach Ort und Kategorie
  - Extraktion von Ortsnamen aus Suchergebnissen
  - Vor- und Nachname zur Zeit ohne feste Reihenfolge
  - Automatische Verarbeitung der Daten (z. B. Auswertung für Serienbrief) daher erschwert

Nicht atomare Wertebereiche: Probleme (2)

- Beispieldatensatz:
- ("31.01.25, 10:57", "1", "Max Muster", "Ulmstr. 2, 89075 Ulm, 0176...", "TV", "Elektronik", "LG+ LCD 50'", "1", "5 Jahre", "Biete")
- Suche, Sortierung oder Extraktion von bzw. nach Teilinformationen wird erschwert oder ist sogar unmöglich
- Es resultiert ein sogenanntes *Pflegeproblem*
- Es ist daher besser, nur *atomare Attributwerte* zuzulassen
  - Attribute, die zusammengesetzte Wertebereiche repräsentieren, müssen in verschiedene Attribute zerlegt werden
- Dies führt uns auf die *1. Normalform*

1. Normalform

1. Normalform: Definition

- Die erste Normalform erlaubt nur *atomare Attributwerte*
  - D. h. in Tabelleneinträgen darf jeweils nur ein Wert stehen
- Attribute, die mehrere Werte umfassen, werden in entsprechend viele getrennte Attribute aufgegliedert
- Hierdurch entsteht eine neue Relation mit mehr Attributen

Zerlegung zusammengesetzter Attribute

- Beispieleintrag:
- Aus

  ("31.01.25, 10:57", "1", "Max Muster", "Ulmstr. 2, 89075 Ulm 0176...", "TV", "Elektronik", "LG+ LCD 50'", "1", "5 Jahre", "Biete")
- wird
- ("31.01.25", "10:57", "1", "Max", "Muster", "Ulmstr.", "2", "89075", "Ulm", " 0176...", "TV", "Elektronik", "LG+ LCD 50'", "1", "5", "Jahre", "Biete")
- bzw. die folgende Relation mit neuen Attributen:
  - Ware (Datum, Uhrzeit, KID, Vorname, Nachname, Str, Nr, PLZ, Ort, Tel, Bezeichnung, Kategorie, Beschreibung, Anzahl, Alter, Zeitraum, Art)

Relation Ware: Nähere Beschreibung (1)

- Ware (Datum, Uhrzeit, KID, Vorname, Nachname, Str, Nr, PLZ, Ort, Tel, Bezeichnung, Kategorie, Beschreibung, Anzahl, Alter, Zeitraum, Art)
  - Primärschlüssel:
    - Datum, Uhrzeit: Genauer Zeitpunkt des Angebots
    - KID: Identifikationsnummer oder -kennung des (An-)Bieters
  - Str, Nr, PLZ, Ort, Tel: Adressangaben des (An-)Bieters
  - Bezeichnung: Warenname, z. B. TV, Radio
  - Kategorie: Spezifiziert die Warengruppe, der ein Artikel zugeordnet werden kann, z. B. Computer, Kleidung, Elektronik, etc.
  - Beschreibung: Nähere Beschreibung des Angebots, z. B. PAL-TV 22', DVD, Sack Kartoffeln, etc.

Relation Ware: Nähere Beschreibung (2)

- Ware (Datum, Uhrzeit, KID, Vorname, Nachname, Str, Nr, PLZ, Ort, Tel, Bezeichnung, Kategorie, Beschreibung, Anzahl, Alter, Zeitraum, Art)
  - Anzahl: Wie viel Ware angeboten wird, z. B. *ein* TV-Gerät
  - Alter: Wie alt z. B. gebrauchte Geräte sind
  - Zeitraum: Jahre, Monate, Tage, etc.
  - Art: Die Rolle (Anbieter oder Bieter)

Wiederholungsgruppen (1)

- Drei Beispielangebote von Max Muster:
  - ("31.01.25", "10:57", "*1*", "*Max*", "*Muster*", "*Ulmstr.*", "*2*", "*89075*", "*Ulm*", "*0176...*", "TV", "Elektronik", "LG+ LCD 50'", "1", "5", "Jahre", "Biete")
  - ("1.02.11", "12:01", "*1*", "*Max*", "*Muster*", "*Ulmstr.*", "*2*", "*89075*", "*Ulm*", "*0176...*", "CD", "Musik", "Bach Matthäuspassion", "1", "10", "Jahre", "Biete")
  - ("1.02.11", "16:33", "*1*", "*Max*", "*Muster*", "*Ulmstr.*", "*2*", "*89075*", "*Ulm*", "*0176...*", "Kartoffeln", "Nahrung", "1 Sack", "25kg", "1", "Woche", "Biete")
- Zu einem Satz Kundendaten existieren mehrere Tauschangebote bzw. jedes Tauschangebot enthält wieder alle Kundendaten
  - Kundendaten repräsentieren somit eine *Wiederholungsgruppe* und stellen redundante Information dar

Wiederholungsgruppen (2)

- Datenredundanz verursacht einen erhöhten Aktualisierungsaufwand
  - Inkonsistente Datensätze und eine Verletzung der semantischen Integrität können die Folge sein
- Insgesamt haben wir erneut ein *Pflegeproblem*, welches vermieden werden sollte
- *Wiederholungsgruppen* werden bei der 1. Normalform in eigene Relationen *ausgegliedert*
- Der Schlüssel der neuen Relation beruht auf dem Schlüssel der alten Relation
  - Auf diese Weise bleibt die Beziehung zum ursprünglichen Datensatz erhalten

Behandlung der Wiederholungsgruppe (1)

- Wir betrachten die Relation
  - Ware (Datum, Uhrzeit, KID, *Vorname*, *Nachname*, *Str*, *Nr*, *PLZ*, *Ort*, *Tel*, Bezeichnung, Kategorie, Beschreibung, Anzahl, Alter, Zeitraum, Art)

- Wir können eine Relation "Kunde" ausgliedern:
  - Kunde (Datum, Uhrzeit, KID,
    Vorname, Nachname, Str, Nr, PLZ, Ort, Tel)

- Es bleibt dann für die Relation "Ware":
  - Ware (Datum, Uhrzeit, KID,
    Bezeichnung, Kategorie, Beschreibung,
    Anzahl, Alter, Zeitraum, Art)

Behandlung der Wiederholungsgruppe (2)

- Dies ist allerdings hier nicht sinnvoll, da die Attribute der Wiederholungsgruppe nicht vom gesamten Primärschlüssel abhängen
  - Ware (Datum, Uhrzeit, *KID*, *Vorname*, *Nachname*, *Str*, *Nr*, *PLZ*, *Ort*, *Tel*, Bezeichnung, Kategorie, Beschreibung, Anzahl, Alter, Zeitraum, Art)
  - Abhängigkeit besteht nur zum Attribut "KID"
    - Datum und Uhrzeit überflüssig in "Kunde"
  - Probleme:
    - Bei weiteren Wareneinträgen geht Bezug zwischen Ware und Kunde verloren
    - Tabelle wird unnötig umfangreich => Redundanz
- Dies führt uns auf die *2. Normalform*

2. Normalform

2. Normalform: Definition

- Eine Relation ist in der zweiten Normalform
  - wenn die erste Normalform vorliegt und
  - alle Nichtschlüsselattribute von allen Schlüsselattributen abhängen
    - D. h. nicht nur von einer Teilmenge der Schlüsselattribute
  - Eine Relation mit nur einem Schlüsselattribut ist automatisch in 2. NF wenn sie in 1. NF ist
- Eine Relation wird in die 2. NF überführt,
  - indem alle von einem Teilschlüssel abhängigen Attribute zusammen mit dem Teilschlüssel in eine eigene Relation ausgegliedert werden
- Missachtung der 1. und 2. NF kann zu diversen *Anomalien* führen

Änderungs-Anomalie

- Verbleiben auslagerungsfähige Attribute in der Originalrelation, so hat eine Änderung am Wertebereich des Attributs Änderungen aller Einträge zur Folge
- Beispiele:
  - Adressdaten ändern sich

    => Alle Adresseinträge müssen angepasst werden
  - Beispiel: Kategorie "Elektronik" wird in "Unterhaltungselektronik" geändert

    => Alle Einträge mit der Bezeichnung Elektronik müssen angepasst werden
    - Dies lässt sich vermeiden, indem Kategorien in einer eigenen Relation verwaltet und aus anderen Relationen nur auf diese verwiesen wird

Einfüge-Anomalie

- Eine neue Gattung kann erst eingeführt werden, wenn das erste Objekt dieser Gattung eingefügt wird
- Beispiele:
  - Daten eines Ortes können erst angegeben werden, wenn ein Bewohner dieses Ortes ein Tauschgebot abgegeben hat
  - Bevor nicht ein Eintrag der Kategorie "Elektronik" in der Tabelle steht, existiert diese nicht
  - Definition einer Kategorie erfolgt erst im Anwendungsfall, es sei denn, es existiert eine Liste vordefinierter Kategorien

Lösch-Anomalie

- Wird der letzte Eintrag einer Gattung gelöscht, so gehen alle Informationen über diese Gattung verloren
- Beispiele:
  - Wenn das letzte Gebot aus einem Ort abgearbeitet ist, verschwinden wieder alle Informationen über diesen Ort
  - Wird der letzte Eintrag der Kategorie "Elektronik" aus der Tabelle entfernt, existiert diese Kategorie nicht länger
- Um diese Anomalien zu vermeiden werden Attribute mit eingeschränkter Abhängigkeit in eigene Relationen ausgegliedert

Behandlung der Wiederholungsgruppe

- Wir betrachten wieder die Relation
  - Ware (Datum, Uhrzeit, KID, Vorname, Nachname, Str, Nr, PLZ, Ort, Tel, Bezeichnung, Kategorie, Beschreibung, Anzahl, Alter, Zeitraum, Art)
  - Abhängigkeit der Kundendaten besteht nur zum Attribut "KID"
- Wir gliedern die betreffenden Attribute zusammen mit dem *Teilschlüssel* in die Relation "Kunde" aus:
  - Kunde (KID, Vorname, Nachname, Str, Nr, PLZ, Ort, Tel)
- Es bleibt dann für die Relation "Ware":
  - Ware (Datum, Uhrzeit, KID, Bezeichnung, Kategorie, Beschreibung, Anzahl, Alter, Zeitraum, Art)

Weitere Analyse

- Kunde (KID, Vorname, Nachname, Str, Nr, PLZ, Ort, Tel)
  - Ein Kunde könnte mehrere Adressen haben, unter denen er/sie erreichbar ist (z. B. Privatadresse und Geschäftsadresse)

    => eine weitere Wiederholungsgruppe, die sich ausgliedern lässt
  - Unterschiedliche Adressen lassen sich z. B. über die Festnetztelefonnummer und/oder die PLZ unterscheiden
  - Diese Attribute gehören aber nicht zum Schlüssel der Relation
  - Die Adressen sind also nicht vom Primärschlüssel abhängig
  - Dies führt uns auf die *3. Normalform*

3. Normalform

3. Normalform: Definition

- Eine Relation ist in der dritten Normalform
  - wenn die zweite Normalform vorliegt und
  - alle Nichtschlüsselattribute nicht *transitiv*, d. h. nur mittelbar, von den Schlüsselattributen abhängen
    - D. h. Nichtschlüsselattribute dürfen nicht von anderen Nichtschlüsselattributen abhängen
- Eine Relation wird in die 3. NF überführt,
  - indem alle transitiv abhängigen Attribute in neue Relationen ausgegliedert werden
- Missachtung der 3. Normalform kann wieder zu den genannten *Anomalien* führen (siehe 2. NF)

Weitere Analyse (1)

- Kunde (KID, Vorname, Nachname, Str, Nr, PLZ, Ort, Tel)
- Wir gliedern die von der Telefonnummer abhängigen Attribute in eine eigene Relation "Adresse" aus:
  - Adresse (Tel, Str, Nr, PLZ, Ort)
- Die Relation "Kunde" reduziert sich hierdurch auf:
  - Kunde (KID, Vorname, Nachname, Tel)
- Der Teilschlüssel Tel gestattet es, unterschiedliche Adressen eines Kunden zu unterscheiden

Weitere Analyse (2)

- Die nähere Betrachtung der Relation "Adresse" offenbart eine weitere Wiederholungsgruppe mit Abhängigkeit von einem Nichtschlüsselattribut
- Adresse (Tel, Str, Nr, PLZ, Ort)
  - Ort ist von PLZ abhängig
- Wir gliedern deshalb PLZ und Ort ebenfalls in eine eigene Relation "Orte" aus:
  - Orte (PLZ, Ort)
- Die Relation Adresse reduziert sich hierdurch zu:
  - Adresse (Tel, Str, Nr, PLZ)
  - PLZ braucht kein Teilschlüssel zu sein, da Adressen schon über Tel unterscheidbar

Weitere Analyse (3)

- Wir betrachten die Relation "Ware":
  - Ware (Datum, Uhrzeit, KID, Bezeichnung, Kategorie, Beschreibung, Anzahl, Alter, Zeitraum, Art)
- Das Attribut "Bezeichnung" folge vorgegebenen Warenbezeichnungen (z. B. TV, CD)
- Sofern sich diese bestimmten Kategorien fest zuordnen lassen, existiert auch hier eine Wiederholungsgruppe mit Abhängigkeit von einem Nichtschlüsselattribut
- Beispiele:
  - CD => Musik
  - Kartoffeln => Nahrung
- Diese sollte in eine eigene Relation ausgelagert werden

Weitere Analyse (4)

- Wir erhalten dann die Relation "Warenart", für die Beziehung zwischen Bezeichnung und Kategorie:
  - Warenart (Bezeichnung, Kategorie)
- Ware reduziert sich hierdurch zu:
  - Ware (Datum, Uhrzeit, KID, Bezeichnung, Beschreibung, Anzahl, Alter, Zeitraum, Art)

Gesamtmenge aller Relationen in 3. NF

- Orte (PLZ, Ort)
- Adresse (Tel, Str, Nr, PLZ)
- Kunde (KID, Vorname, Nachname, Tel)
- Warenart (Bezeichnung, Kategorie)
- Ware (Datum, Uhrzeit, KID, Bezeichnung, Beschreibung, Anzahl, Alter, Zeitraum, Art)
- Tauschen (DatumA, AnbieterID, DatumB, BieterID, Status)

Dualitätsprinzip

- Normalisierung führt zur Entstehung einer neuen, wenn auch vergleichsweise geringen Redundanz, die die erzielbare Redundanzreduktion verringert
  - Redundante Einträge dienen zur inhaltlichen Verknüpfung von Tabellen
- Beispiel: Die Relationen Orte, Adresse und Kunde
- Alle beschriebenen Normalisierungsschritte sind umkehrbar
  - D. h. ein Zusammenfassen der aufgegliederten Relationen in die ursprüngliche Relation ist jederzeit möglich
  - Kann für eine bessere Zugriffseffizienz sinnvoll bzw. erforderlich sein

Datenbankanfragen bei mehreren Tabellen

- Stehen alle Daten in einer gemeinsamen Tabelle, so lassen sich die gewünschten Informationen mit Hilfe eines einfachen **`SELECT`**-Befehls abfragen
- Beispiel:
  - Ursprüngliche Relation für Waren
    - Ware (Datum, Name, Adresse, Bezeichnung, Kategorie, Beschreibung, Anzahl, Alter, Zeitraum, Art)
  - Beispiel: Selektion aller Angebote von "Max Muster" mit Auswahl einer Untermenge der vorhandenen Attribute:
    - **`SELECT Datum, Bezeichnung, Kategorie, Beschreibung, Ort FROM Ware WHERE Name="Max Muster"`**
  - Derartige Anfragen sind nun nicht mehr möglich, da die Attribute durch die Normalisierung über diverse Relationen verteilt wurden

Tabellen mittels Joins verknüpfen

- *Verbünde* bzw. *Joins* erlauben es, Daten, die über mehrere Tabellen verteilt sind, so zu behandeln, als ob sie einer Tabelle entstammen
- Syntax des Joins:
  - **`SELECT spalten FROM tabellen WHERE bedingung`**
  - Bei der Anfrage werden also mehrere Tabellen angegeben
- Der Join besteht aus zwei Schritten:
  - Bildung des *kartesischen Produkts* der beteiligten Tabellen
  - *Selektion* jener Daten aus der kombinierten Tabelle, die der angegebenen Bedingung genügen

Beispiel: Schlösser und Schlüssel

- Zwei Tabellen bzw. Relationen, in denen die Beziehung zwischen einigen Schlüsseln und Schlössern codiert ist:

SchlossNr	Typ	Form	SchlueNr
1	A	A0815	2
2	B	B0816	1
3	C	C0817	2

SchlueNr	Typ	Form
1	D	DRFK1
2	E	EZTX3
3	F	FKNA9

- Schlüssel Nr. 2 kann z. B. die Schlösser Nr. 1 und 3 öffnen

Schritt 1: Kartesisches Produkt

- **`SELECT * FROM Schloss, Schluessel`** liefert:

Schloss:

SchlossNr	Typ	Form	SchlueNr
1	A	A0815	2
2	B	B0816	1
3	C	C0817	2

Schluessel:

SchlueNr	Typ	Form
1	D	DRFK1
2	E	EZTX3
3	F	FKNA9

Schloss x Schluessel:

SchlossNr	Typ	Form	SchlueNr	SchlueNr	Typ	Form
1	A	A0815	2	1	D	DRFK1
2	B	B0816	1	1	D	DRFK1
3	C	C0817	2	1	D	DRFK1
1	A	A0815	2	2	E	EZTX3
2	B	B0816	1	2	E	EZTX3
3	C	C0817	2	2	E	EZTX3
1	A	A0815	2	3	F	FKNA9
2	B	B0816	1	3	F	FKNA9
3	C	C0817	2	3	F	FKNA9

Schritt 2: Selektion (1)

- Selektiere alle Schlösser, die mit Schlüssel Nr. 2 geöffnet werden können
- Versuch 1: **`SELECT * FROM Schloss, Schluessel WHERE Schluessel.SchlueNr = 2`**

SchlossNr	Typ	Form	SchlueNr	SchlueNr	Typ	Form
1	A	A0815	2	1	D	DRFK1
2	B	B0816	1	1	D	DRFK1
3	C	C0817	2	1	D	DRFK1
1	A	A0815	2	2	E	EZTX3
2	B	B0816	1	2	E	EZTX3
3	C	C0817	2	2	E	EZTX3
1	A	A0815	2	3	F	FKNA9
2	B	B0816	1	3	F	FKNA9
3	C	C0817	2	3	F	FKNA9

Bedingung unzureichend:
Es werden alle Schlösser gelistet

SchlossNr	Typ	Form	SchlueNr	SchlueNr	Typ	Form
1	A	A0815	2	2	E	EZTX3
2	B	B0816	1	2	E	EZTX3
3	C	C0817	2	2	E	EZTX3

Schritt 2: Selektion (2)

- Selektiere alle Schlösser, die mit Schlüssel Nr. 2 geöffnet werden können
- Lösung: `SELECT * FROM Schloss, Schluessel WHERE`
  `Schluessel.SchlueNr = 2 AND`
  `Schloss.SchlueNr = 2`

  Bei gleichem Attributnamen Tabellennamen voranstellen

SchlossNr	Typ	Form	SchlueNr	SchlueNr	Typ	Form
1	A	A0815	2	1	D	DRFK1
2	B	B0816	1	1	D	DRFK1
3	C	C0817	2	1	D	DRFK1
1	A	A0815	2	2	E	EZTX3
2	B	B0816	1	2	E	EZTX3
3	C	C0817	2	2	E	EZTX3
1	A	A0815	2	3	F	FKNA9
2	B	B0816	1	3	F	FKNA9
3	C	C0817	2	3	F	FKNA9

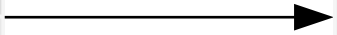
SchlossNr	Typ	Form	SchlueNr	SchlueNr	Typ	Form
1	A	A0815	2	2	E	EZTX3
3	C	C0817	2	2	E	EZTX3

Schritt 2: Selektion (3)

- Filtern überflüssiger Daten:

```
SELECT SchlossNr FROM Schloss, Schluessel WHERE
          Schluessel.SchlueNr = 2 AND
          Schloss.SchlueNr = 2
```

SchlossNr	Typ	Form	SchlueNr	SchlueNr	Typ	Form
1	A	A0815	2	1	D	DRFK1
2	B	B0816	1	1	D	DRFK1
3	C	C0817	2	1	D	DRFK1
1	A	A0815	2	2	E	EZTX3
2	B	B0816	1	2	E	EZTX3
3	C	C0817	2	2	E	EZTX3
1	A	A0815	2	3	F	FKNA9
2	B	B0816	1	3	F	FKNA9
3	C	C0817	2	3	F	FKNA9



SchlossNr
1
3

Weitere Joins

- Der beschriebene Join ist nur einer unter vielen, mit denen sich die Daten von Tabellen kombinieren lassen
- Er wird auch als (expliziter) "Cross Join" bezeichnet
  - `SELECT * FROM Schloss CROSS JOIN Schluessel`
- Entsprechend ist die bisher verwendete Syntax ein *impliziter* Cross Join, da die Schlüsselworte fehlten
- Weitere Joins:
  - Inner Join, Outer Join, Equi Join, Left Join, Right Join,...

Entity-Relationship-Modell

Datenbankentwurf

- Normalformen dienen häufig der Überarbeitung bzw. Verbesserung vorhandener Datenbankentwürfe
- Durch einen strukturierten Entwurf kann indes schon in der Datenmodellierungsphase ein Konzept erarbeitet werden, welches die Anforderungen der Normalformen erfüllt
- Hierfür bieten sich die *Entity-Relationship-Modelle* (ERM) bzw. Gegenstands-Beziehungs-Modelle an
  - Diese wurden 1976 von Peter Chen in seiner Veröffentlichung "The Entity-Relationship Model" vorgestellt
    - Seither gab es eine Reihe unterschiedlicher Weiterentwicklungen und Varianten

Entity-Relationship-Modell: Entität

- Der zentrale Begriff im ERM ist das *Objekt* bzw. die *Entität*
- Entitäten bezeichnen eindeutig identifizierbare Objekte der realen Welt
  - Dies können *konkrete Objekte* wie Gegenstände oder Personen sein, aber auch *abstrakte Dinge*, wie Gesetze oder Verträge
- Entitäten besitzen Eigenschaften
  - Die Eigenschaften werden als *Attribute* bezeichnet
  - Jedem Attribut ist eine *Domäne*, d. h. ein Wertebereich, zugeordnet
    - Enthält alle Werte, die ein Attribut annehmen kann

Entity-Relationship-Modell: Beziehungen

- Zwischen Entitäten bestehen Beziehungen
- Diese werden nach ihrer Komplexität unterschieden
- Es gibt drei verschiedene Beziehungsfälle, je nachdem, wie viele Entitäten an einer Beziehung beteiligt sind

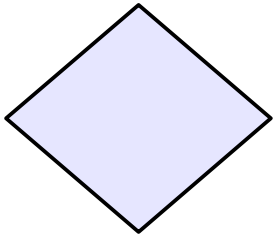
Einfache Beziehung

- 1:1 Beziehung
  - Einer Entität aus einer Menge A von Entitäten ist *genau eine* Entität einer anderen Menge B von Entitäten zugeordnet
- Beispiele:
  - Ein bestimmter Schlüssel passt zu genau einem bestimmten Schloss
  - Eine Fernsteuerung steuert ein bestimmtes Gerät
- Zur Darstellung von Beziehungen führte Chen eine spezielle Notation ein
  - Im Laufe der Zeit haben sich unterschiedliche Varianten dieser Notation herausgebildet

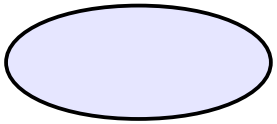
Notation nach Chen

- Entitäten bzw. Objekte werden durch *Rechtecke* repräsentiert

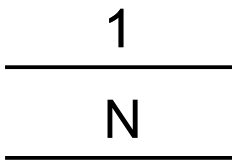
- Beziehungen durch *Rauten*



- Attribute durch *Ellipsen*

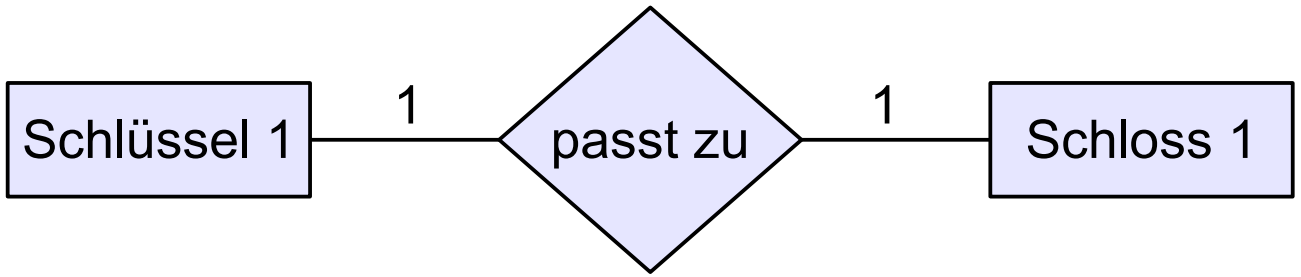


- Verbunden werden Entitäten und Beziehungen durch *Kanten*
  - Die *Beschriftung* der Kante gibt den Typ der Beziehung an



Einfache Beziehung: Darstellung

- Schlüssel 1 passt nur zu Schloss 1
  - Schlüssel 1 schließt nur Schloss 1 auf
- Schloss 1 passt nur zu Schlüssel 1
  - Schloss 1 kann nur von Schlüssel 1 geöffnet werden

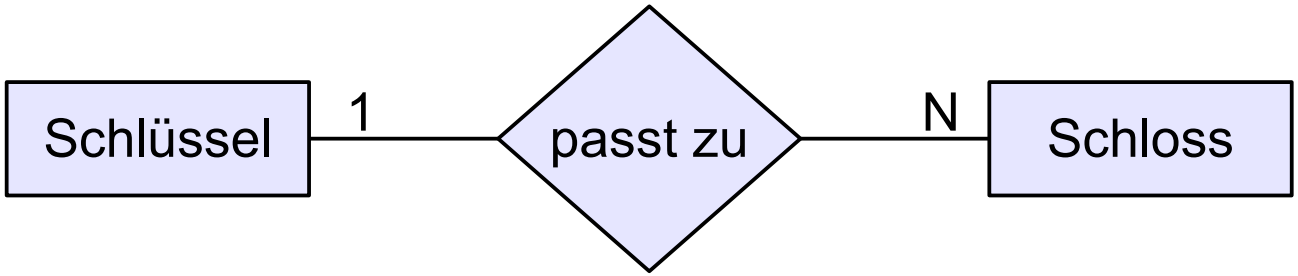


Multiple Beziehung

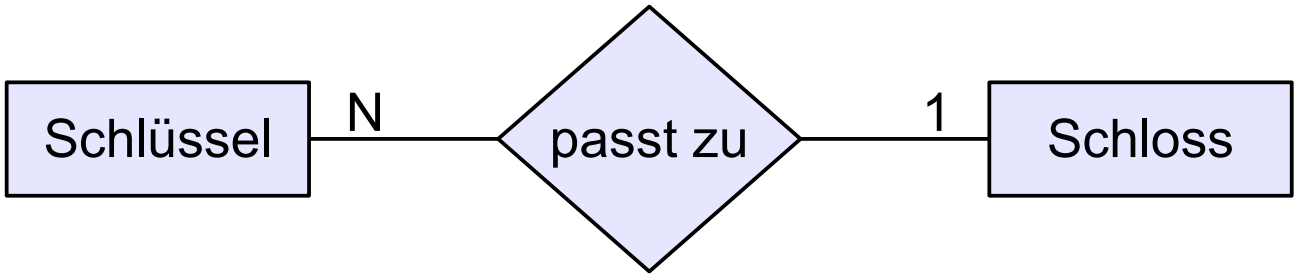
- 1:n Beziehung
  - Einer Entität aus einer Menge A von Entitäten ist *mindestens eine* Entität einer anderen Menge B zugeordnet
- Beispiele:
  - Ein Generalschlüssel kann mindestens eines, aber in der Regel mehrere Schlösser öffnen
  - Zu einem Schloss existiert mindestens ein Schlüssel, aber ggf. auch mehrere Nachschlüssel
  - Eine Universalfernsteuerung steuert evtl. mehrere Geräte, aber mindestens ein Spezielles

Multiple Beziehung: Darstellung

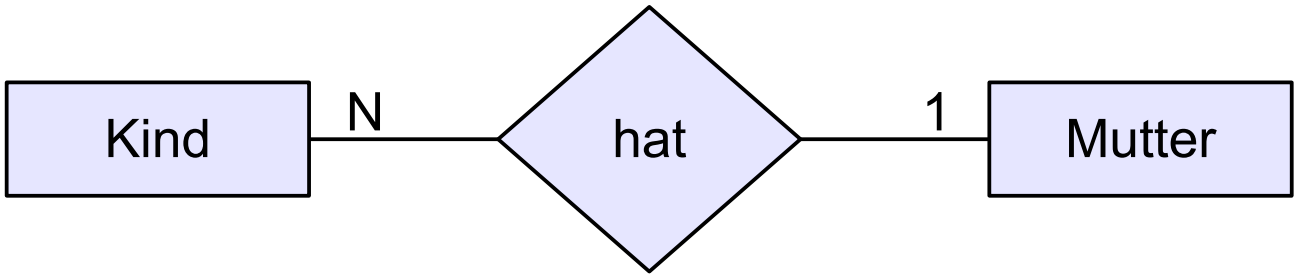
- Generalschlüssel öffnet mehrere Schlösser



- Zu einem Schloss existieren mehrere Schlüssel



- Eine Mutter kann mehrere Kinder haben

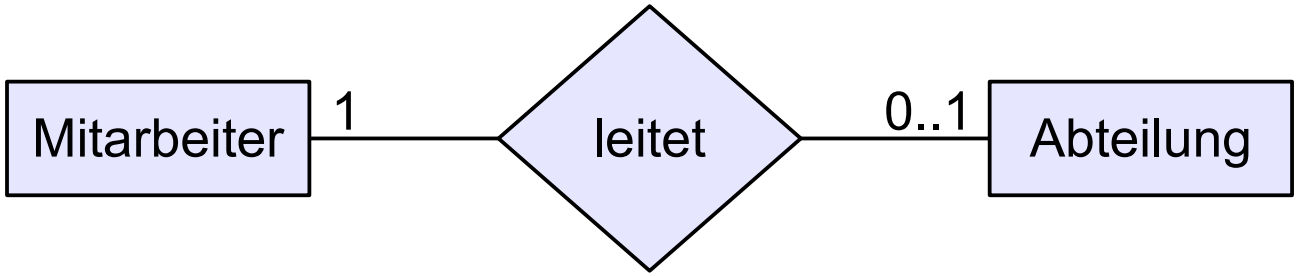


Konditionelle Beziehung

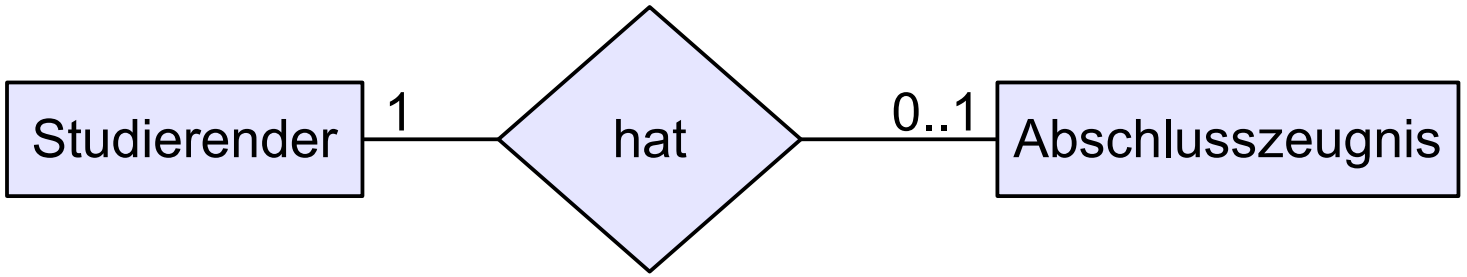
- c-Beziehung mit n=0 oder n=1
  - Einer Entität aus einer Menge A von Entitäten ist *keine oder genau eine* Entität einer anderen Menge B zugeordnet
- Beispiele:
  - Nach einer Klausur kann ein Studierender keine oder genau eine Note haben
    - Je nachdem, ob er/sie an der Klausur teilgenommen hat oder nicht
  - Ein Kind kann genau eine oder keine Mutter/Vater (mehr) haben
  - Ein Studierender kann keinen oder genau einen Bachelorabschluss in einem bestimmten Fach haben

Konditionelle Beziehung: Darstellung

- Ein Mitarbeiter leitet entweder eine oder keine Abteilung
  - Er/Sie ist entweder Abteilungsleiter/in oder nicht

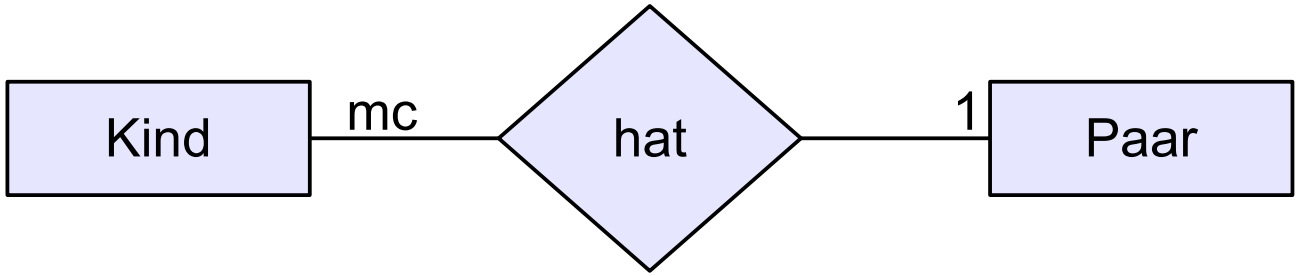


- Ein Studierender hat entweder kein oder ein Abschlusszeugnis, welches seine persönlichen Daten enthält



Multiple konditionelle Beziehung

- mc-Beziehung mit n = 0, 1, ...
  - Einer Entität aus einer Menge A von Entitäten ist *keine, eine oder mehrere* Entitäten einer anderen Menge B zugeordnet
- Beispiele:
  - Ein Elternpaar kann 0, 1 oder mehrere Kinder haben



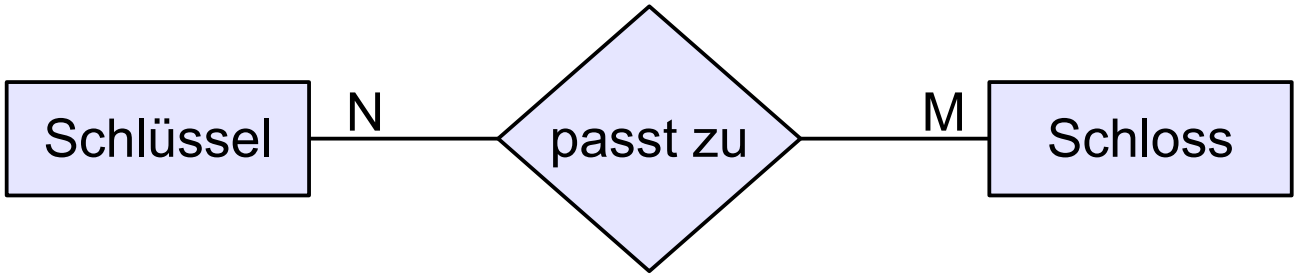
  - Nach der Prüfungszeit kann ein Studierender 0, 1 oder mehrere Prüfungsergebnisse haben

Komplexe Beziehung

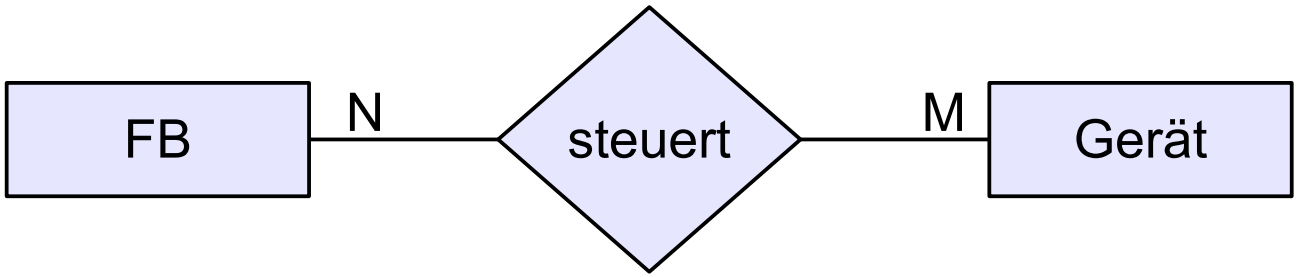
- n:m Beziehung
  - Einer Entität aus einer Menge A von Entitäten sind *beliebig viele* Entitäten einer anderen Menge B zugeordnet (1:m) und umgekehrt (n:1)
- Beispiele:
  - Ein Mieterschlüssel öffnet Haustür und Wohnungstür, die Haustür kann durch mehrere Mieterschlüssel geöffnet werden
  - Eine Universalfernsteuerung steuert mehrere Geräte und ein Gerät kann durch mehrere Fernsteuerungen bedient werden

Komplexe Beziehung: Darstellung

- Generalschlüssel und Mieterschlüssel
  - Ein Schlüssel passt zu mehreren Schlössern
  - Zu einem Schloss passen mehrere Schlüssel

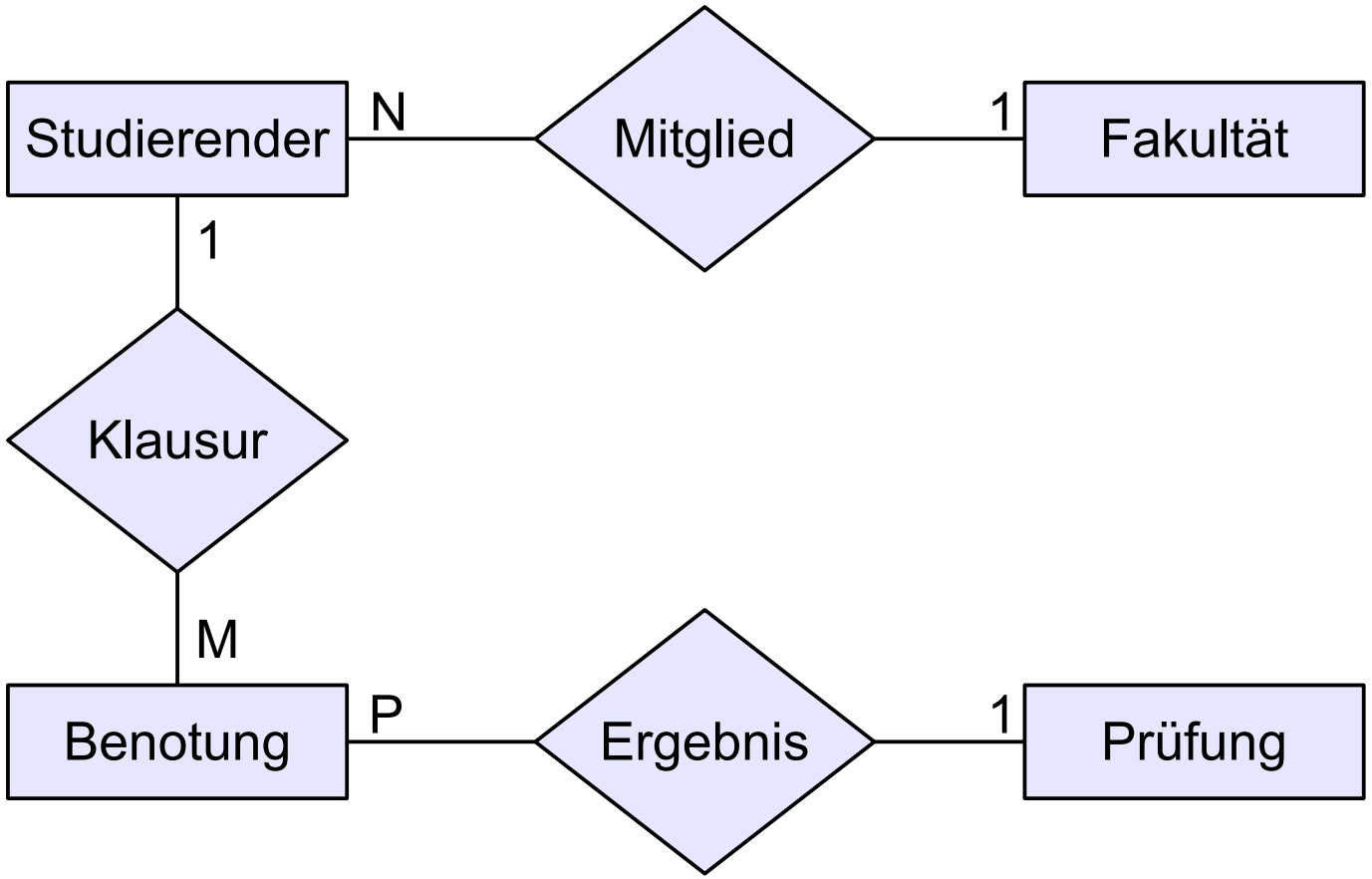


- Fernbedienung und Geräte

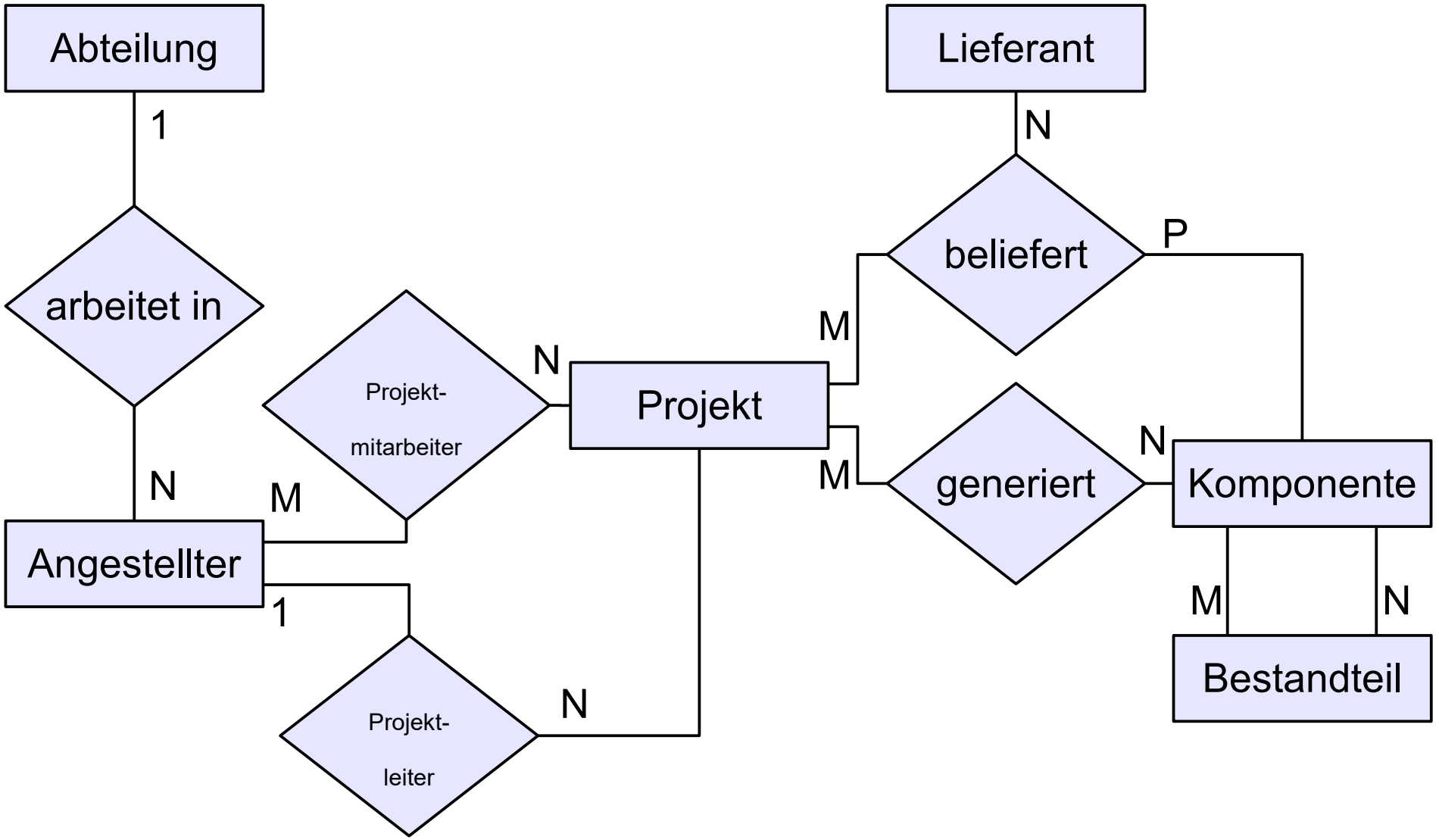


Weitere Beispiele: Verwaltung Studierender

- Neben N können auch weitere Buchstaben verwendet werden, um unterschiedliche Anzahlen zu verdeutlichen



Weitere Beispiele: Firmenverwaltung

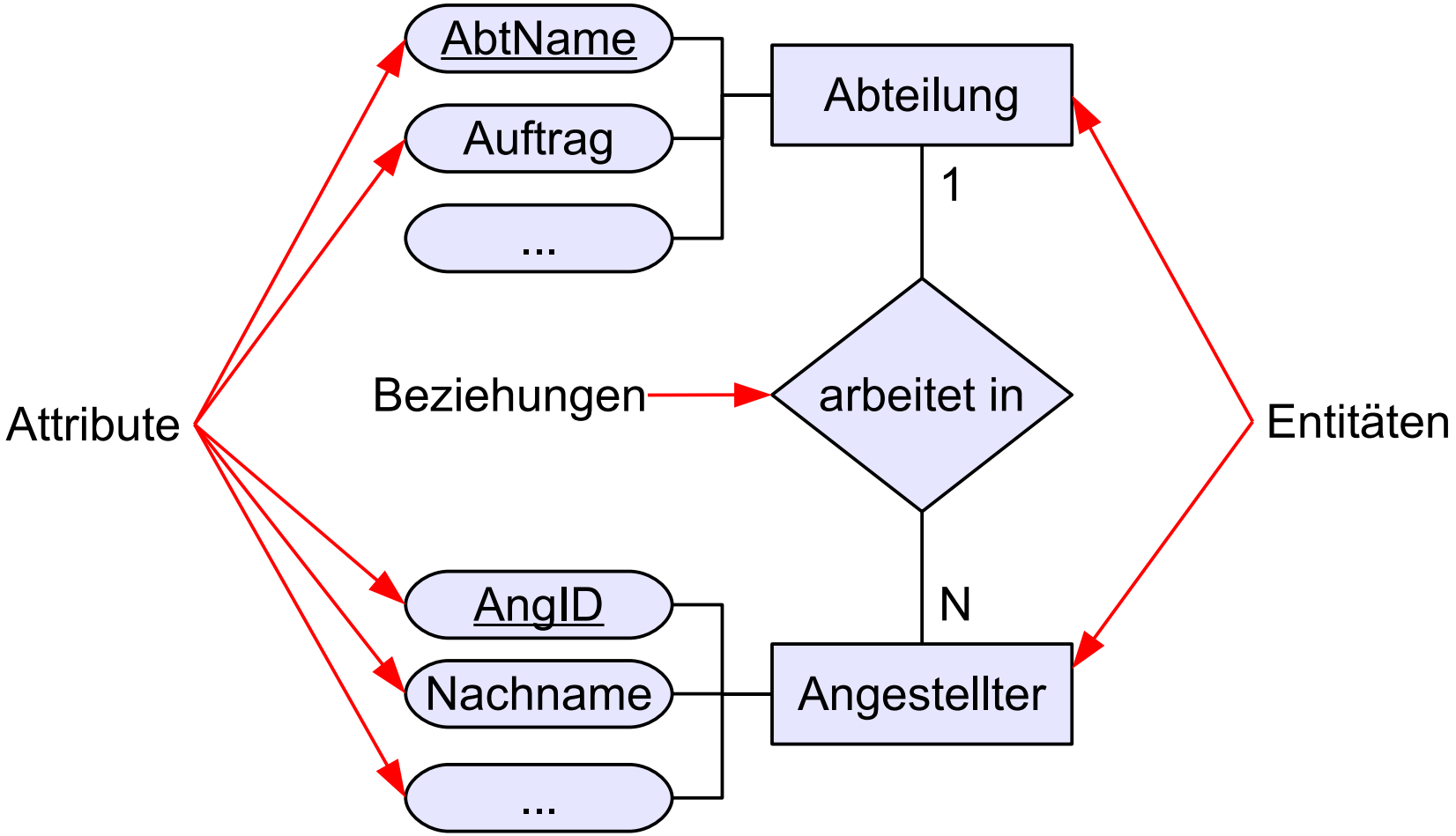


Datenbankdesign mit dem ERM

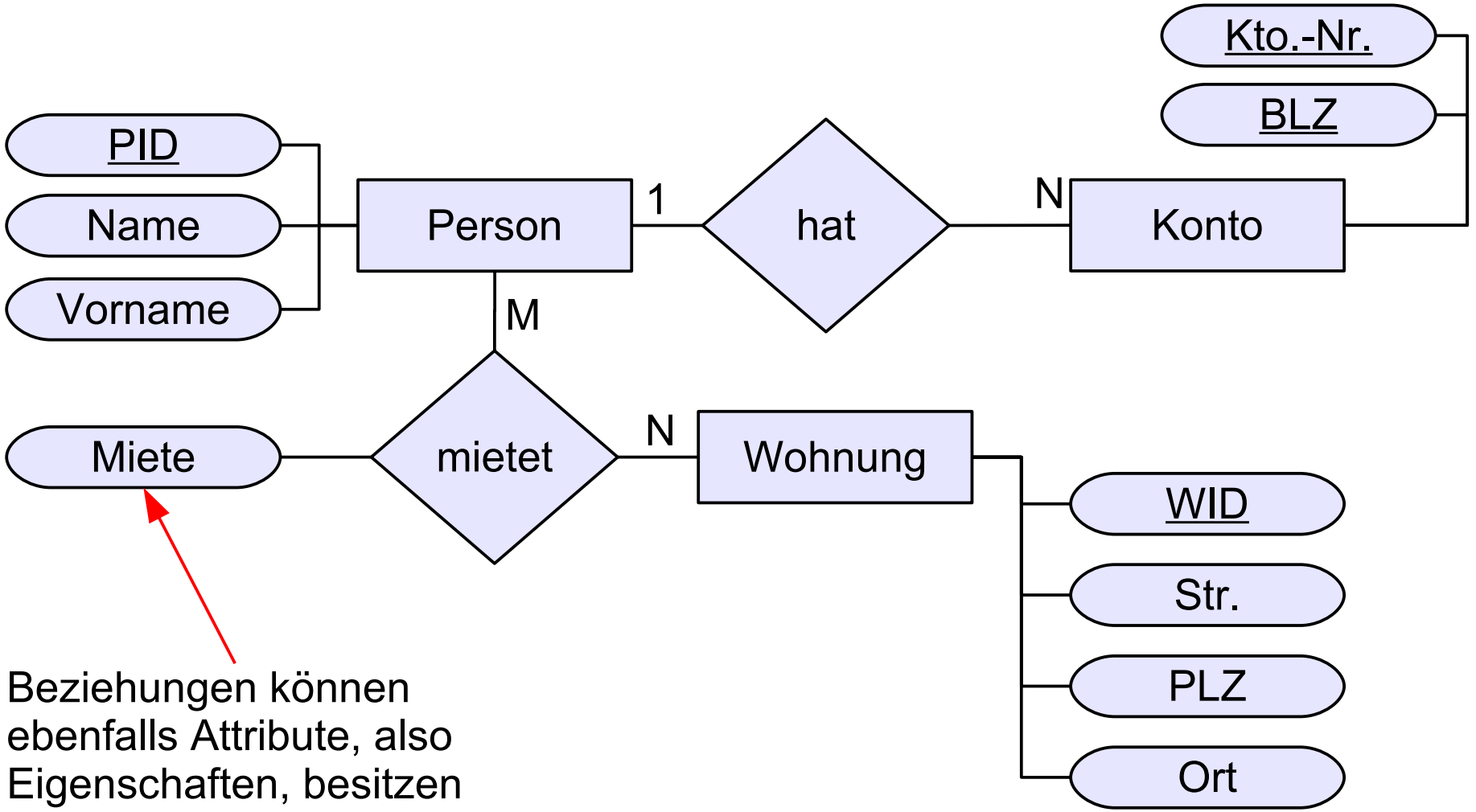
- Vollzieht sich in vier Schritten:
  - Identifikation der relevanten Entitäten und Beziehungen
  - Identifikation der Art der Beziehungen (1:1, 1:N, N:M)
  - Definition der Attribute und deren Wertebereiche
  - Darstellung der Entitäten und Beziehungen und Festlegung der Primärschlüssel
- Prozess der zunehmenden Verfeinerung

Angestellte und Abteilungen mit Attributen

- Primärschlüssel werden unterstrichen



Mieter und Wohnung



Relationale Datenmodellierung

- Das Ergebnis der ER-Modellierung wird schließlich in eine korrespondierende Tabellenstruktur überführt
- Da Tabellen die einzige verfügbare Datenstruktur der relationalen Datenmodellierung sind, müssen hierzu sowohl die *Entitäten* als auch die *Beziehungen* durch Tabellen dargestellt werden
- Beispieltabellen für Entitäten:
  - Person:

PID	Name	Vorname

  - Konto:

Kto.-Nr.	BLZ

  - Wohnung:

WID	Str.	PLZ	Ort

- Tabelle stellt einen bestimmten Objekttyp mit seinen Ausprägungen dar

Darstellung der Beziehungen

- Tabelle muss Beziehungstyp (1:1, etc.) widerspiegeln
- Am einfachsten sind *n:m Beziehungen* durch Tabellen darstellbar
  - Tabelle trägt den Namen der Beziehung
  - Hierzu werden die Primärschlüssel der beteiligten Entitäten in eine neue Tabelle ausgegliedert
  - Primärschlüssel der neuen Tabelle ist eine Kombination der Primärschlüssel der beteiligten Tabellen
- Tabelle muss ebenfalls alle für die Beziehung wichtigen Attribute besitzen

Beispiel: Mieter und Wohnung

- Grundlage sei eine n:m Beziehung
- Entitäten:
  - Person:

PID	Name	Vorname
1	Schmitt	Bernd
2	Maier	Maria
...	...	...

  - Wohnung:

WID	Str.	PLZ	Ort
1	Adolfstr.	89075	Ulm
2	Bergstr.	10587	Berlin
...	...	...	...

- Beziehungen:
  - Mietet

PID	WID	Miete
1	2	750
2	1	1000
...	...	...

Beispiel: Schloss und Schlüssel

- Grundlage sei eine n:m Beziehung
- Entitäten:
  - Schloss:

SchlossNr	Typ	Form
1	A	A0815
2	B	B0816
...	...	...

  - Schlüssel:

SchlueNr	Typ	Form
1	C	C0815
2	D	D0816
...	...	...

- Beziehungen:
  - Passt_zu

SchlossNr	SchlueNr
1	2
2	2
...	...

1:n Beziehung im Relationenmodell

- Wie bei der n:m Beziehung kann diese durch eine Tabelle dargestellt werden, deren Attribute aus den Primärschlüsseln der ursprünglichen Tabellen besteht
- Das Schlüsselattribut des eindeutigen Objekttyps, der für den ":n-Teil" steht, wird als Primärschlüssel gewählt
  - D. h. es ist hier keine Kombination der Schlüssel, wie bei der n:m Beziehung, erforderlich
  - Primärschlüssel muss eindeutig sein, hieraus folgt eine Belegung der zugehörenden Spalte mit unterschiedlichen Werten

1:n Beziehung: Beispiel 1

- Entitäten:
  - Schloss:

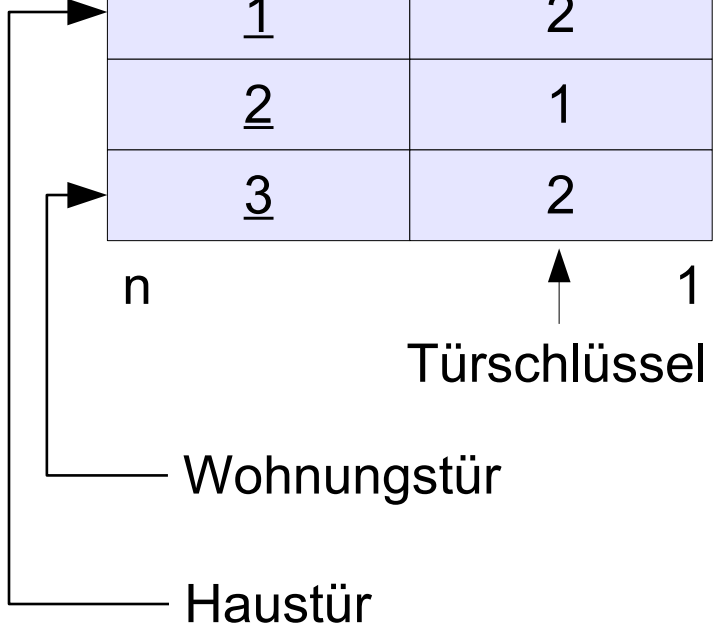
SchlossNr	Typ	Form
1	A	A0815
2	B	B0816
3	C	C0817

  - Schlüssel:

SchlueNr	Typ	Form
1	D	DRFK1
2	E	EZTX3
3	F	FKNA9

- Beziehungen:
  - SchließtAuf

SchlossNr	SchlueNr
1	2
2	1
3	2



1:n Beziehung: Beispiel 2

- Entitäten:
  - Schloss:

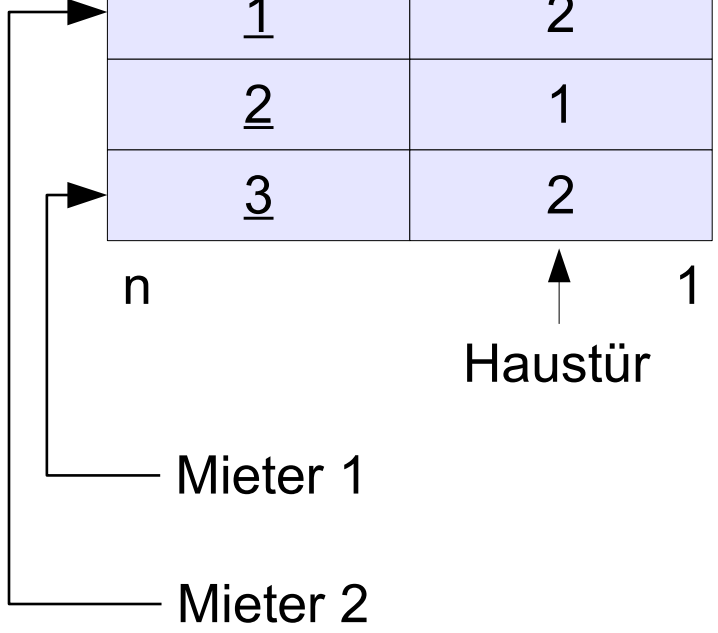
SchlossNr	Typ	Form
1	A	A0815
2	B	B0816
3	C	C0817

  - Schlüssel:

SchlueNr	Typ	Form
1	D	DRFK1
2	E	EZTX3
3	F	FKNA9

- Beziehungen:
  - WirdGeöffnetVon

SchlueNr	SchlossNr
1	2
2	1
3	2



1:n Beziehung: Redundanzreduktion

- Beziehung kann auch ohne eigenständige Tabelle ausgedrückt werden
  - Dazu wird in der Tabelle des eindeutigen Objekttyps ein *Fremdschlüssel* auf die referenzierte Tabelle eingeführt

SchlossNr tritt nun nur noch einmal auf

SchlossNr	Typ	Form	SchlueNr #
1	A	A0815	2
2	B	B0816	1
3	C	C0817	2

SchlueNr	Typ	Form
1	D	DRFK1
2	E	EZTX3
3	F	FKNA9

1:1 Beziehung im Relationenmodell

- Nicht direkt darstellbar im Relationenmodell
- 1. Fall: Darstellung durch eine Tabelle deren Attribute aus den Primärschlüsseln der ursprünglichen Tabellen besteht
  - Zuweisung zweier Schlüssel führt auf n:m Beziehung
  - Nur die Kombination der beiden Attributwerte muss eindeutig sein
- 2. Fall: Einführung eines Fremdschlüssels in der Tabelle des eindeutigen Objekttyps
  - Nur der Primärschlüssel muss eindeutig sein
  - Fremdschlüsselwerte können mehrfach auftreten
    - Fremdschlüssel ist nur in der zugehörenden Objektrelation eindeutig

1:1 Beziehung im Relationenmodell: Lösung

- Beziehung muss durch Anwendungsprogramm durchgesetzt werden
  - Programm muss gewährleisten, dass Attributwerte nach Einfügeoperationen stets unterschiedlich sind
- Zu diesem Zweck gibt es in SQL bzw. MySQL das **`UNIQUE`**-Constraint
- Eine beim Tabellenentwurf als **`UNIQUE`** definierte Spalte kann nur unterschiedliche Werte aufnehmen: Eindeutiger Schlüssel
- Sonst Fehlermeldung
- Primärschlüssel ist Schlüssel der zusätzlich nicht **`NULL`** sein darf
  - D. h. das Schlüsselattribut muss in jeder Tabellenzeile einen Wert besitzen

1:1 Beziehung im Relationenmodell (1)

- Entsprechend zu 1:n Beziehung mit Fremdschlüssel
  - Beziehung wird also ohne eigenständige Tabelle ausgedrückt durch Einführen eines Fremdschlüssels in der Tabelle des eindeutigen Objekttyps

SchlossNr tritt als Primärschlüssel nur einmal auf

SchlossNr	Typ	Form	SchlueNr #
1	A	A0815	2
2	B	B0816	1
3	C	C0817	3

SchlueNr	Typ	Form
1	D	DRFK1
2	E	EZTX3
3	F	FKNA9

Durch **UNIQUE** wird Eindeutigkeit des Fremdschlüssels erzwungen

1:1 Beziehung im Relationenmodell (2)

- Tritt die Entität SchlueNr nicht in weiteren Beziehungen auf, so muss diese nicht als eigenständige Tabelle modelliert werden
- In diesem Fall lässt sich die 1:1 Beziehung effizienter durch eine einzelne Tabelle darstellen

SchlossNr	Typ	Form	SchlueNr	STyp	SForm
1	A	A0815	2	E	EZTX3
2	B	B0816	1	D	DRFK1
3	C	C0817	3	F	FKNA9

SchlossNr
tritt als Primär-
schlüssel
nur einmal auf

Durch **UNIQUE**
Eindeutigkeit von
SchlueNr
erzwungen

Konditionelle Beziehung im Relationenmodell

- Entsprechend zu 1:1, aber als 1:0,1 Beziehung
- Beispiel: Zu jedem Schloss existiert kein oder genau ein Schlüssel
- Grundlage ist die Tabelle, die den Primärschlüssel liefern kann

SchlossNr tritt als Primär-schlüssel auf jeden Fall auf

SchlossNr	Typ	Form	SchlueNr #
1	A	A0815	2
2	B	B0816	NULL
3	C	C0817	3

Schlüssel kann z. B. verloren gegangen sein

SchlueNr	Typ	Form
1	D	DRFK1
2	E	EZTX3
3	F	FKNA9

Durch **UNIQUE** Eindeutigkeit des Fremdschlüssels erzwungen

Literatur

[1] S.Bakken et al.: *PHP Handbuch,* PHP-Dokumentationsgruppe

[2] P. Chen: *The Entity-Relationship Model*, 1976

[3] M. Ebner: *SQL lernen*, Addison-Wesley, 1999

[4] G. Matthiessen, M. Unterstein: *Relationale Datenbanken und SQL*, Addison-Wesley, 1997

[5] A. Meier: *Relationale Datenbanken – Eine Einführung für die Praxis*, Springer, 1998

[6] L. Piepmeyer: Grundkurs Datenbanksysteme, Hanser, 2011

[7] J. D. Ullman: *Principles of database and knowledge base systems*, Computer Science Press, 1988

[8] G. Vossen: *Datenbankmodelle, Datenbanksprachen und Datenbankmanagementsysteme*, Oldenbourg, 1999