Datenbank-basierte Webserver

Dynamische Erstellung von Webseiten mit PHP

Karl Riedling
Institut für Sensor- und Aktuatorsysteme

Dynamische Erstellung von Webseiten mit PHP

- Allgemeines zu PHP
- Die Geschichte von PHP
- Kompatibilität zwischen PHP-Versionen
- Einführung in PHP
- Erstellung und Testen von PHP-Programmen

Dynamische Erstellung von Webseiten mit PHP

- **Allgemeines zu PHP**
- Die Geschichte von PHP
- Kompatibilität zwischen PHP-Versionen
- Einführung in PHP
- Erstellung und Testen von PHP-Programmen

Was bedeutet „PHP“?

- Ursprüngliche Bedeutung: „*Personal Home Page*“
- Interpretation ab PHP 3: Rekursives Akronym für „*PHP: Hypertext Preprocessor*“ (analog zu „GNU“ = „*GNU’s Not UNIX*“)

Was ist PHP?

- Script-Sprache für allgemeine Anwendungen
- *Open Source*-Software
- C-artige Syntax mit Elementen von Perl und Java
- Besonders geeignet für Web-Anwendungen
- Eingebettet in HTML

Eingebettet in HTML

- Beispiel:

```
<html>
    <head>
        <title>Beispiel</title>
    </head>
    <body>
        <?php echo "Ich bin ein PHP-Script!"; ?>
    </body>
</html>
```

Eingebettet in HTML

- Seiten, bei denen nur Teile dynamisch verändert werden sollen, können größtenteils als „normaler“ HTML-Code mit eingebetteten PHP-Befehlen realisiert werden
- PHP-Code steht zwischen den PHP-Tags „<?php“ und „?>“

Eingebettet in HTML

- Gegensatz zu anderen Server-Programmiersprachen:
  - Nur Teile des HTML-Codes müssen vom Programm erstellt werden
- Gegensatz zu Client-seitiger Programmierung:
  - Benutzer bekommt nur die Ergebnisse des Programmcodes, nicht aber den Programmcode selbst zu sehen

Anwendungsmöglichkeiten von PHP

- Serverseitige Scripts in Verbindung mit einem Webserver (Hauptanwendung)
- Allgemeine Scripts (z.B. für cron-Anwendungen unter UNIX oder Linux; grundsätzlich auch als Script-Sprache in Windows verwendbar)
- Clientseitige GUI- (Graphical User Interface) Anwendungen – PHP-GTK (Erweiterung von PHP um den objektorientierten GTK+-Toolkit)

Betriebssystems-Unterstützung von PHP

- Linux
- UNIX (HP-UX, Solaris, OpenBSD)
- Microsoft Windows (9x aufwärts)
- Mac OS X
- RISC OS
- ...

Webserver-Unterstützung von PHP

- Apache
- Microsoft Internet Information Server
- Personal Web Server
- Netscape und iPlanet Server
- Oreilly Website Pro server
- Caudium
- Xitami
- OmniHTTPd,
- ...

Implementierung von PHP

- In den meisten Umgebungen als Modul für den Webserver
- Jedenfalls auch als CGI-Prozessor verwendbar

Programmiertechniken unter PHP

- *Prozedurale* Programmierung
- *Objektorientierte* Programmierung (mit einer zunehmenden Anzahl von Features objektorientierter Sprachen)
- Beliebige Kombinationen beider Techniken

Ausgabemöglichkeiten mit PHP

- HTML-Code
- Bilder (JPEG – *Joint Photographic Experts Group*, PNG – *Portable Network Graphics,* und GIF – *Graphics Interchange Format*)
- *Flash*-Animationen
- PDF- (*Portable Data Format*) Code
- Beliebige (Text-) Formate, z.B. XML; damit auch Unterstützung für *AJAX* möglich
- Ausgabe auf Webseiten oder in Dateien am Server

Datenbank-Support von PHP

Adabas D	Ingres	Oracle (OCI7, OCI8)
dBase	InterBase	Ovrimos
Empress	FrontBase	PostgreSQL
FilePro (read-only)	mSQL	Solid
Hyperwave	Direct MS-SQL	Sybase
IBM DB2	MySQL	Velocis
Informix	ODBC	Unix dbm

Kommunikation unter PHP

- Unterstützung der folgenden Protokolle:
  - LDAP
  - IMAP
  - SNMP
  - NNTP
  - POP3
  - HTTP
  - COM (unter Windows)
  - WDDX

Weitere Features von PHP

- Instanzierung von Java-Objekten
- Reguläre Ausdrücke (Perl)
- SAX- und DOM-Standards für Bearbeitung von XML-Dokumenten
- Schnittstellen für Finanztransaktionen
- Suchmaschinen
- Datenkompressions-Tools

Dynamische Erstellung von Webseiten mit PHP

- Allgemeines zu PHP
- **Die Geschichte von PHP**
- Kompatibilität zwischen PHP-Versionen
- Einführung in PHP
- Erstellung und Testen von PHP-Programmen

Entwicklung von PHP

- PHP-FI („Personal Home Page / Forms Interpreter“)
  - Entwickelt 1995 von Rasmus Lerdorf
  - Ursprünglich Perl-Scripts, später C-Code
  - Perl-artige Variable, automatische Interpretation von Formular-Parametern, Einbettung in HTML

Entwicklung von PHP

- PHP-FI 2.0
  - Offiziell freigegeben Ende 1997
  - Im Wesentlichen Projekt von Rasmus Lerdorf
  - Verbesserter C-Code
  - In etwa 50.000 Domains (1% des Internet) in Verwendung

Entwicklung von PHP

- PHP 3
  - 1997 neu geschrieben von Andi Gutmans und Zeev Suraski
  - Entspricht im Wesentlichen heutigem PHP
  - Leistungsfähigere, verbesserte und objektorientierte Syntax
  - Erweiterbar – große Anzahl neuer Funktionen von zahlreichen Entwicklern
  - Im Juni 1998 offiziell freigegeben
  - Auf ca. 10% aller Webserver installiert

Entwicklung von PHP

- PHP 4
  - Ab 1998 Neuentwicklung des PHP-Codes durch Andi Gutmans und Zeev Suraski
  - Verbesserung der Performance für komplexe Anwendungen
  - „*Zend-Engine*“ („Zend“ = Zeev + Andi) offiziell freigegeben im Mai 2000
  - Installiert auf mehreren Millionen Sites, etwa 20% aller Webserver

Entwicklung von PHP

- PHP 4
  - Unterstützung weiterer Webserver
  - HTTP-Sessions
  - Output-Pufferung
  - Sicherere Behandlung von Benutzereingaben
  - Neue Sprach-Konstrukte

Entwicklung von PHP

- PHP 5
  - Version 2.0 der *Zend-Engine* – verbesserte Performance
  - Verbesserte Handhabung von Klassen (bessere Performance, zusätzliche Features – vergleichbarer mit anderen OOP-Sprachen)
  - Weitgehend kompatibel zu PHP 4 (mit einigen Ausnahmen ➔ PHP-Dokumentation)

Entwicklung von PHP

- PHP 6
  - Beginn der Entwicklung 2005; erste offizielle Release angekündigt für den 3. PHP World Kongress 2009 in München (24. und 25. November 2009); im Frühjahr 2010 Entwicklung *de facto* stillgelegt (Problem: Unicode-Unterstützung)
  - Zahlreiche Neuerungen geplant; viele davon bereits in PHP 5.3 bzw. 5.4+ implementiert
  - Etliche Features von PHP 5 zur Kompatibilität mit früheren Versionen (PHP 3, PHP 4) entfernt – (noch) nicht in PHP 5.3, teilweise in PHP 5.4

Entwicklung von PHP

- PHP 6
  - PHP 6 wird in der ursprünglich vorgesehenen Form nie erscheinen.

Entwicklung von PHP

- PHP 7
  - Release von PHP 7 im Dezember 2015
  - Wesentliche neue Features:
    - Optionale Deklaration von (skalaren) Funktions-Parametern und -Ergebnissen
    - Neue Operatoren („??“, „<=>“)
    - Array-Konstaanten
    - Anonyme Klassen

Entwicklung von PHP

- PHP 7
  - Wesentliche neue Features:
    - Volle Unicode-Unterstützung (einschließlich Unicode-Stringkonstanten)
    - Verbesserungen bei *Closures* (anonymen Funktionen)
    - Sicherheitsrelevante Verbesserungen (z.B. bei `unserialize()`; kryptographische Funktionen)
    - *Expectations* (verbessertes `assert()`)

Entwicklung von PHP

- PHP 7
  - Wesentliche neue Features:
    - Verbesserte *Generator*-Funktionalität
    - Division mit ganzzahligem Ergebnis
    - Verbesserte Session-Optionen
    - Verbesserungen bei Callbacks bei regulären Ausdrücken

Entwicklung von PHP

- Gegenwärtig (Stand 03.11.2018) aktuellste PHP-Releases:
  - 5.4.45 (Support beendet am 14.09.2014)
  - 5.5.38 (Support beendet am 10.07.2016)
  - 5.6.38 (Support vorgesehen bis 31.12.2018)
  - 7.0.32 (Support vorgesehen bis 03.12.2018)
  - 7.1.23 (Support vorgesehen bis 01.12.2019)
  - 7.2.11 (Support vorgesehen bis 30.11.2020)
- PHP 7.3 ist derzeit in Pre-Releases verfügbar
- Derzeit Lebensdauer von 3 Jahren für eine Release

Dynamische Erstellung von Webseiten mit PHP

- Allgemeines zu PHP
- Die Geschichte von PHP
- **Kompatibilität zwischen PHP-Versionen**
- Einführung in PHP
- Erstellung und Testen von PHP-Programmen

Kompatibilität zwischen PHP-Versionen

- Syntaktische volle Kompatibilität für PHP 3-Code unter PHP 4 oder für PHP 4-Code unter PHP 5 bis 5.2, *aber*
- Subtile Unterschiede in der Funktionalität und Performance von Funktionen und Sprachkonstrukten (speziell in Grenzbereichen der Definition, selbst zwischen Sub-Releases innerhalb einer Version)
- Erhebliche Änderungen zwischen PHP-Versionen bis einschließlich 5.3 und PHP 5.4+ durch schrittweise Implementierung der für PHP 6 vorgesehenen Features:

Kompatibilität zwischen PHP-Versionen

- Neue Features ursprünglich geplant für PHP 6, implementiert ab PHP 5.3:
  - Unicode-Unterstützung (UTF-8, UTF-16, UTF-32)
    - Performance-Einbußen bis zu 300%!
    - Ohne Unicode-Erweiterung wurde PHP 6 als „nicht wesentlich langsamer als PHP 5“ beschrieben.
  - 64-Bit-Integers
  - Integrierter Cache für PHP-Scripts

Kompatibilität zwischen PHP-Versionen

- Neue Features ursprünglich geplant für PHP 6, implementiert ab PHP 5.3 (Fortsetzung):
  - Eigener XML-Parser (`XMLReader` und `XMLWriter` für Lesen und Schreiben von XML-Dateien)
  - Neue Extension `fileinfo` (Informationen zum MIME-Typ von Dateien)
  - Webservice-Extension für SOAP („**S**imple **O**bject **A**ccess **P**rotocol“)

Kompatibilität zwischen PHP-Versionen

- Neue Features ursprünglich geplant für PHP 6, implementiert ab PHP 5.3 (Fortsetzung):
  - Erweiterte Datums-Funktionalität
  - Aktivierbarer Filter für Datenübergaben aus Aufrufen
  - Separate *Name Spaces* für Klassen
  - Neustrukturierung der Datenbank-Unterstützung
  - Erweiterungen bei Schleifen

Kompatibilität zwischen PHP-Versionen

- Massive Änderungen ursprünglich vorgesehen für PHP 6, implementiert ab PHP 5.3 (Fortsetzung):
  - Keine Unterstützung für einige PHP 3-Features:
    - `register_globals` entfällt ersatzlos
    - Globale Arrays `$HTTP_GET_VARS` und `$HTTP_POST_VARS` entfallen ersatzlos
    - `magic_quotes`-Filterung entfällt ersatzlos
  - Änderungen im Bereich des *Safe Modes*
  - Verbesserung der Sicherheit beim Öffnen von (fremden) URLs

Kompatibilität zwischen PHP-Versionen

- Änderungen ursprünglich vorgesehen für PHP 6, implementiert ab 5.3 (Fortsetzung):
  - Verbesserte Fehlermeldungen
  - Reguläre Ausdrücke nach POSIX-Standard entfallen
  - Kompatibilitäts-Mode zu PHP 4 (ZEND 1.0) entfällt
  - Einige alte Extensions entfallen

Kompatibilität zwischen PHP-Versionen

- Viele der genannten alten Features stehen jedoch bis einschließlich PHP 5.3 zur Verfügung
  - Sie gelten aber als „deprecated“ (veraltet)
  - Ihre Verwendung kann Kompatibilitäts-Warnungen bewirken

Kompatibilität zwischen PHP-Versionen

- Änderungen in PHP 5.3 (unter Anderem):
  - `goto`-Befehl neu
  - Unterstützung für *Namespaces* neu
  - Unterstützung für *Late Static Bindings* neu
  - Kurzform für den ternären Operator `?:`
  - Anonyme Funktionen („*Closures*“) neu

Kompatibilität zwischen PHP-Versionen

- Änderungen in PHP 5.4 (unter Anderem):
  - *Safe mode* entfernt
  - `register_globals` and `register_long_arrays` ini-Optionen entfernt
  - `import_request_variables()` entfernt
  - `allow_call_time_pass_reference` entfernt
  - `magic_quotes` entfernt
  - Default-Zeichensatz ist UTF-8 (statt wie bisher ISO-8859-1) – betrifft z.B. `htmlentities()`
  - *Regular expression*-Funktionen nach POSIX-Standard entfernt (`ereg()`, `ereg_replace()`…)

Kompatibilität zwischen PHP-Versionen

- Änderungen in PHP 5.4 (unter Anderem):
  - Binär-Konstanten neu
  - *Traits* (feiner granulare Methode zur Vererbung von Funktionen innerhalb einer Klassen-Hierarchie) neu
  - *Short array syntax* neu
  - Unterstützung für Multibyte-Strings ist Default
  - zahlreiche Verbesserungen der ZEND-Engine

Kompatibilität zwischen PHP-Versionen

- Änderungen in PHP 5.5 (unter Anderem):
  - Support für Windows XP und 2003 entfernt; lauffähig unter Windows ab Vista
  - Ursprüngliche MySQL-Extension (Funktionen `mysql_connect()`, `mysql_query()`…) als deprecated erklärt. Statt dessen die Extensions MySQLi (*MySQL Improved* Extension – für Rückwärts-Kompatibilität Funktionen `mysqli_connect()`, `mysqli_query()`… vorhanden) oder PDO_MySQL verwenden!

Kompatibilität zwischen PHP-Versionen

- Änderungen in PHP 5.6 (unter Anderem):
  - Skalare Ausdrücke mit Konstanten
  - „*Splat*“ Operator „`...`“ zum Entpacken von Arrays in Argumentlisten von Funktionen (wie z.B. in *Ruby*)
  - Exponentiations-Operator „`**`“ (z.B. `2 ** 3 == 8`)
  - Erweiterung des `use`-Operators

Kompatibilität zwischen PHP-Versionen

- Änderungen in PHP 7.0 (unter Anderem):
  - Änderungen im Fehler-Handling (vielfach *Exceptions* statt Laufzeitfehler; Änderungen bei `E_STRICT`)
  - Änderungen in der Behandlung von (komplexen) Variablen(-Konstrukten)
  - Änderungen bei `foreach`
  - Änderungen bei unzulässigen Operationen (unzulässige Oktalkonstanten, bitweise Verschiebungen, Division durch Null)

Kompatibilität zwischen PHP-Versionen

- Änderungen in PHP 7.0 (unter Anderem):
  - Einige (schon länger kaum gebrauchte) Funktionen entfernt (z.B. Unterstützung für PostScript-Schriften in Graphik-Funktionen)
  - ASP- und Script-PHP-Tags entfernt
  - Statische Aufrufe nichtstatischer Funktionen entfernt
  - Mehrfache `default:`-Blöcke in `switch`-Befehlen und mehrfache Verwendung des gleichen Funktionsparameter-Namens unzulässig

Kompatibilität zwischen PHP-Versionen

- Änderungen in PHP 7.0 (unter Anderem):
  - „#“ als Kommentar-Einleitung in `ini`-Dateien unzulässig
  - PHP-4-Konstruktoren (Name = Name der Klasse) als *deprecated* (veraltet) erklärt

Kompatibilität zwischen PHP-Versionen

- Änderungen in PHP 7.1 (unter Anderem):
  - Nullable Types (alternative Verwendung einer Type oder von NULL als Funktions-Parameter oder Ergebnis)
  - Funktionen mit explizitem Ergebnis `void`
  - „`[]`“ als Kurzversion für `list()`
  - Klassenkonstanten können Sichtbarkeit (`public`, `protected`, `private`) haben
  - *Multi Catch Exception Handling*

Kompatibilität zwischen PHP-Versionen

- Änderungen in PHP 7.1 (unter Anderem):
  - Schlüsselwerte in `list()` (oder seiner Kurzversion)
  - Negative Offsets in Strings
  - Asynchrone Verarbeitung von Signalen
  - HTTP/2 Server *Push* Support in `curl`

Kompatibilität zwischen PHP-Versionen

- Änderungen in PHP 7.2:
  - Zahlreiche kleine Verbesserungen in diversen Features und Extensions

Kompatibilität zwischen PHP-Versionen

- Änderungen in PHP 7.3 (unter Anderem):
  - Änderung in der *Heredoc*-Syntax
  - Verbesserte Performance bei der Behandlung von Multibyte-Strings
  - Konstanten sind grundsätzlich *case sensitive*
  - Strings können auch länger als 2 GB sein.

Kompatibilität zwischen PHP-Versionen

- In PHP 5 Scripts mit Error-Setting E_STRICT testen – Scripts, die unter PHP ab Version 5.1 mit E_STRICT korrekt laufen, sollten dies auch unter zukünftigen Versionen von PHP.
- Jedenfalls: Bei jeder Änderung der PHP-Version auf einem Server (oder bei Neu-Installation bestehender Software) einen umfassenden Test vornehmen!

Dynamische Erstellung von Webseiten mit PHP

- Allgemeines zu PHP
- Die Geschichte von PHP
- Kompatibilität zwischen PHP-Versionen
- **Einführung in PHP**
- Erstellung und Testen von PHP-Programmen

Einführung in PHP

- Syntax
- Datentypen
- Variable
- Konstanten
- Ausdrücke
- Operatoren
- Ablaufsteuerung
- Funktionen
- Klassen und Objekte
- *Exceptions*
- Referenzen

Einführung in PHP

- **Syntax**
- Datentypen
- Variable
- Konstanten
- Ausdrücke
- Operatoren
- Ablaufsteuerung
- Funktionen
- Klassen und Objekte
- *Exceptions*
- Referenzen

Die Syntax von PHP

- Für Webserver-seitige Scripts: Eingebettet in HTML-Code
- PHP-Code erzeugt HTML-Befehle oder Teile von HTML-Befehlen
- Beispiel:
  - Ausgabe der aktuellen Temperatur; `$ort` und `$temperatur` sind PHP-Variable; die Funktion `date("H:i:s")` gibt die aktuelle Zeit im Format HH:MM:SS aus

Die Syntax von PHP

- 1. Alternative:
  - Seite besteht im Wesentlichen aus HTML-Code; nur einzelne Daten werden über PHP eingefügt:

```
<html>
<head><title>Aktuelle Temperatur</title></head>
<body>
Temperatur in <?php echo $ort; ?> um <?php echo
date("H:i:s"); ?> Uhr: <?php echo $temperatur;
?> °C.<br>
</body>
</html>
```

  ergibt z.B.:

  Temperatur in Wien um 14:25:39 Uhr: 12.5 °C

Die Syntax von PHP

- 2. Alternative:
  - Gesamte Seite wird durch PHP-Code erstellt:

```
<?php
 echo "<html>\n";
 echo "<head><title>Aktuelle
 Temperatur</title></head>\n";
 echo "<body>\n";
 echo "Temperatur in ".$ort." um ".date("H:i:s").
 " Uhr: ".$temperatur." °C.<br>\n";
 echo "</body>\n";
 echo "</html>\n";
?>
```


Die Syntax von PHP

- 3. Alternative:
  - Vereinfachte Syntax:

```
<?php
 echo "<html>\n";
 echo "<head><title>Aktuelle
 Temperatur</title></head>\n";
 echo "<body>\n";
 echo "Temperatur in $ort um ".date("H:i:s").
 " Uhr: $temperatur °C.<br>\n";
 echo "</body>\n";
 echo "</html>\n";
?>
```

Die Syntax von PHP

- „Einklammerung“ des PHP-Codes:

```
1.<?php echo "Das ist PHP-Code"; ?>
2.<script language="php">
    echo ("Das ist PHP-Code");
  </script>
3.<? echo "Das ist PHP-Code" ?>
4.<?= "Das ist PHP-Code" ?>
5.<% echo "Das ist PHP-Code" %>
6.<%= "Das ist PHP-Code" %>
```


Die Syntax von PHP

- „Einklammerung“ des PHP-Codes:
  - Methode 1: Bevorzugt; XHTML- und XML-kompatibel
  - Methode 2: Für Verwendung mit Tools wie Frontpage; in PHP 7 nicht mehr unterstützt
  - Methoden 3 – 6: Erfordern spezielle Konfiguration des Servers; nicht empfohlen für PHP vor Version 5.4 (Portabilität!)
  - Methoden 3 und 4 werden aber ab PHP 5.4 immer unterstützt
  - Methoden 5 und 6 ab PHP 5.4 nicht mehr unterstützt

Die Syntax von PHP

- Bedingte Ausgabe von HTML-Textblöcken:

```
<?php
    if (date("H") < 12) {
?>
<b>Es ist Vormittag!</b><br>
<?php
    } else {
?>
<b>Es ist Nachmittag!</b><br>
<?php
    }
?>
```


Die Syntax von PHP

- Trennung von PHP-Befehlen: Grundsätzlich durch Strichpunkte („;“)
- Unmittelbar vor einem schließenden PHP-Tag kann der Strichpunkt entfallen

Die Syntax von PHP

- Kommentare im C-, C++- und UNIX-Stil:

```
<?php
echo "Test"; // Einzeiliger C++-Kommentar
/* Dieser Kommentar kann sich über eine
   beliebige Anzahl von Zeilen erstrecken */
echo "Test"; # Einzeiliger UNIX-Kommentar
?>
```

- C++- und UNIX-Kommentare enden mit der Eingabezeile (Zeilenvorschub)
- C-artige Kommentare („`/* ... */`“) nicht schachteln!

Einführung in PHP

- Syntax
- **Datentypen**
- Variable
- Konstanten
- Ausdrücke
- Operatoren
- Ablaufsteuerung
- Funktionen
- Klassen und Objekte
- *Exceptions*
- Referenzen

PHP-Datentypen

- Vier skalare Datentypen:
  - BOOLEsche Daten (*booleans*)
  - Ganzzahlige Daten (*integers*)
  - Gleitkommazahlen (*floats*)
  - Zeichenketten (*strings*)

PHP-Datentypen

- Zwei zusammengesetzte Datentypen
  - Felder (*arrays*)
  - Objekte (*objects*)
- Zwei Spezial-Typen
  - Ressourcen (*resources*)
  - NULL

PHP-Datentypen

- Datentypen werden in der Regel automatisch (aus dem Kontext) bestimmt, in PHP 7 ist explizite Deklaration möglich:

```
<?php
$a = TRUE;           // $a ist boolean
$b = 123;            // $b ist integer
$c = 3.141592;       // $c ist float
$d = "Hello World!"; // $d ist string
?>
```

- Eine explizite Typ-Umwandlung ist aber möglich (➔ PHP-Dokumentation)

PHP-Datentypen

- BOOLEsche Daten (*booleans*):
  - Zwei Werte (in beliebiger Groß- / Kleinschreibung):
    - `true`
    - `false`

PHP-Datentypen

- BOOLEsche Daten (*booleans*):
  - Die folgenden Werte werden in `false` umgewandelt:
    - Integers: `0`
    - Floats: `0.0`
    - Strings: `""` oder `"0"`
    - Arrays: ohne Elemente
    - Objekte: ohne Datenelemente
    - NULL
  - Alles andere gilt als `true`

PHP-Datentypen

- Ganze Zahlen (*integers*)
  - Werte `... -2, -1, 0, 1, 2, ...`
  - Wertebereich in 32-Bit-Implementierungen
    `-(2^31 - 1) ... (2^31 - 1)`,
    in 64-Bit-Implementierungen
    `-(2^63 - 1) ... (2^63 - 1)`
  - PHP-Konstanten `PHP_INT_SIZE` und `PHP_INT_MAX` geben die Größe bzw. den (positiven) Wertebereich in der aktuellen Implementierung an

PHP-Datentypen

- Ganze Zahlen (*integers*)
  - (Dem Betrag nach) größere Werte werden in `float` umgewandelt
  - Das Ergebnis von *Integer*-Divisionen hat *immer* die Type `float` (in PHP 7: Funktion `intdiv()` mit ganzzahligem Ergebnis)
  - Umwandlung von `float` in `integer` erfolgt immer durch Abschneiden der Dezimalstellen (Rundung nach Null) (auch bei `intdiv()`)

PHP-Datentypen

- Ganze Zahlen (*integers*)
  - Darstellung:

dezimal	`123`	`-8`	`0`	`2147483647`
oktal	`0173`	`-010`	`0`	`017777777777`
hexadezimal	`0x7b`	`-0x8` `0xfffffff8`	`0` `0x0`	`0x7fffffff`

Ab PHP 5.4 ist auch binäre Darstellung verfügbar (`0b01011010 = 90`)

PHP-Datentypen

- Ganze Zahlen (*integers*)
  - Bei Hexadezimal-Darstellung sind „`0x7b`“ und „`0X7B`“ (und andere Kombinationen von Groß- und Kleinschreibung) zulässig
  - Unzulässige Oktalkonstanten (Ziffern > 7) werden bis PHP 5 implizit ignoriert (0128 → 012 = 10), ab PHP 7 wird eine Parser-Fehlermeldung ausgegeben

PHP-Datentypen

- Gleitkommazahlen (*floats*)
  - Beispiele:
    - `123.456`
    - `.123`
    - `1.602e-19`
    - `7E3`
  - Wertebereich meist `~-1.8e308 ... ~1.8e308` (64-Bit-IEEE-Format)
  - Auflösung meist ca. 15 – 16 Dezimalstellen

PHP-Datentypen

- Zeichenketten (*strings*)
  - Sequenzen von 8-Bit-Zeichen
  - Volle Unicode-Unterstützung durch Bibliotheks-Funktionen möglich ➔ PHP-Dokumentation
  - Ab PHP 5.4 wird aber UTF-8 unterstützt (und als Default-Zeichensatz verwendet)
  - Längenbegrenzung für *Strings*: 2 Gigabytes (nur bis PHP 7.2)!

PHP-Datentypen

- Zeichenketten (*strings*)
  - Darstellung mit einfachen Anführungszeichen (`'Ich bin ein String'`):
    - *Escape-Codes* (z.B. „\n“) und Variablennamen werden *nicht* aufgelöst
    - Darstellung des einfachen Anführungszeichens („'“) durch „\'“
    - *Backslashes* („\“), die ausgegeben werden sollen, *müssen* nur vor einfachen Anführungszeichen verdoppelt werden („\\'“ → „\'“), sonst nicht

PHP-Datentypen

- Zeichenketten (*strings*)
  - Darstellung mit einfachen Anführungszeichen (`'Ich bin ein String'`):
    - Verdoppelte *Backslashes* werden aber als *ein Backslash* ausgegeben:
      `echo 'You deleted C:\\*.*?';`
      wird ausgegeben als „You deleted C:*.*?“
    - Strings dürfen Zeilenvorschübe enthalten:
      ```
      echo 'The quick brown fox jumps
      over the lazy white dog\'s back';
      ```


PHP-Datentypen

- Zeichenketten (*strings*)
  - Darstellung mit doppelten Anführungszeichen (`"Ich bin auch ein String"`):
    - Variablennamen werden aufgelöst
    - Sonderzeichen können durch *Escape-Codes* dargestellt werden
    - In allen anderen als den auf der nächsten Seite dargestellten Kombinationen (*Escape-Codes*) erscheint der *Backslash* als „\“

Escape-Codes

Code	Bedeutung
`\n`	Zeilenvorschub (*Line Feed*; `0x0a`)
`\r`	Zum Zeilenanfang (*Carriage Return*; `0x0d`)
`\t`	Horizontaler Tabulator (`0x09`)
`\\`	*Backslash* („\“)
`\$`	Dollar-Zeichen („$“)
`\"`	Doppeltes Anführungszeichen
`\<nnn>`	Zeichen mit Oktal-Wert <nnn>
`\x<nn>`	Zeichen mit Hexadezimal-Wert `0x`<nn>

PHP-Datentypen

- Zeichenketten (*strings*)
  - Darstellung als *Heredoc*:

```
<?php
$var = "Variable";
$str = <<<XYZ
Dieser Text wird der Variablen \$str
zugewiesen. Er kann beliebig lang sein und
\"Escape-Codes\" enthalten. $var werden
expandiert.
XYZ
?>
```

  - Details ➔ PHP-Dokumentation

PHP-Datentypen

- Zeichenketten (*strings*)
  - Darstellung als *Nowdoc* (ab PHP 5.3):

```
<?php
$variable = "Variable";
$str = <<<'XYZ'
Dieser Text wird der Variablen $str
zugewiesen. Er kann beliebig lang sein.
$variable werden nicht expandiert,
Zeilenvorschübe und \"Escape-Sequenzen\"
werden unverändert (z.B. als "\n" ausgegeben.
XYZ
?>
```

  - Details ➔ PHP-Dokumentation

String parsing

- Einfache Syntax: Im String eingebettete Variable werden durch ihren Wert ersetzt:

```
<?php
$bier = 'Hirter';
echo "$bier-Bier schmeckt";  // ok
echo "$bierbier schmeckt";   // falsch
echo "${bier}bier schmeckt"; // ok
echo "{$bier}bier schmeckt"; // ok
echo $bier."bier schmeckt";  // ok
?>
```

- Komplexe Syntax ➔ PHP-Dokumentation

PHP-Datentypen

- Zeichenketten (strings)
  - Auswahl einzelner Zeichen:

```
<?php
$str = "abcdefgh";
echo $str[0];               // gibt "a" aus
echo $str{0};               // gibt "a" aus
echo $str[4];               // gibt "e" aus
echo $str{4};               // gibt "e" aus
echo $str[strlen($str)-1];  // gibt "h" aus
echo $str{strlen($str)-1};  // gibt "h" aus
?>
```

PHP-Datentypen

- Zeichenketten (strings)
  - Erlaubte / bevorzugte Syntax:
    - PHP ≤ 3: `$str[0];`
    - PHP 4: `$str{0};` (`$str[0];`)
    - PHP 5: (`$str{0};`) `$str[0];`
      (`$str{0}` bewirkt ab PHP 5.1 mit `E_STRICT` eine Parser-Warnung!)

PHP-Datentypen

- Zeichenketten (*strings*)
  - *String Concatenation*-Operator („.“):

```
<?php
$str = "ist";
echo "Das ".$str." ein "."String!";
// Ausgabe: "Das ist ein String!"
?>
```

  - Zahlreiche Funktionen für die Manipulation und Konversion von Strings ➔ PHP-Dokumentation

PHP-Datentypen

- Zeichenketten (*strings*)
  - In numerischem Kontext können Strings in Zahlen umgewandelt werden:

```
<?php
$a = "1.1" + 1.4;           // $a ist 2.5
$b = 1.4 + "1.1";           // $b ist 2.5
$c = 1 + "1 Bier";          // $c ist 2
$d = "1 Bier" + "noch 1";   // $d ist 1
?>
```

  - Details ➔ PHP-Dokumentation

PHP-Datentypen

- Felder (*arrays*)
  - Verknüpfung von *Werten* (*values*) mit *Schlüsseln* (*keys*)
  - Wahlweise verwendbar als
    - echtes Feld
    - Liste oder Vektor
    - *Hash-Table*
    - Übersetzungstabelle (*Dictionary*)
    - Datensammlung
    - *Stack*
    - Warteschlange

PHP-Datentypen

- Felder (*arrays*)
  - Definition: Mit dem Sprach-Konstrukt `array()`:
    `array([`*`Schlüssel`* `=>] `*`Wert`*
    `, ...`
    `)`
  - *`Schlüssel`* ist entweder ganzzahlig (≥ 0) oder ein String
  - *`Wert`* ist jeder beliebige in PHP gültige Wert

PHP-Datentypen

- Felder (*arrays*)
  - Beispiele:

```
<?php
$a = array ("Banane" => "gelb", "Kiwi" =>
"grün");
echo $a["Banane"];   // "gelb"
echo $a["Kiwi"];     // "grün"

$b = array (10 => 1, 2, 3, 4);
echo $b[10];         // 1
echo $b[13];         // 4
?>
```


PHP-Datentypen

- Felder (*arrays*)
  - Beispiele (Fortsetzung):

```
<?php
$c = array ("Banane" => 4711, 13 => "grün");
echo $c["Banane"];   // 4711
echo $c[13];         // "grün"
?>
```

PHP-Datentypen

- Felder (*arrays*)
  - Ab PHP 5.4 kann „`array()`“ durch „`[]`“ ersetzt werden:

```
<?php
$a = ["Banane" => "gelb", "Kiwi" => "grün"];
echo $a["Banane"];   // "gelb"

$b = [10 => 1, 2, 3, 4];
echo $b[13];         // 4

$c = ["Banane" => 4711, 13 => "grün"];
echo $c[13];         // "grün"
?>
```


PHP-Datentypen

- Felder (*arrays*)
  - Definition / Änderung: mit alternativer Notation:
    `$array[Schlüssel] = Wert;`
    `$array[] = Wert;`
  - *`Schlüssel`* ist entweder ganzzahlig (≥ 0) oder ein String
  - *`Wert`* ist jeder beliebige in PHP gültige Wert
  - Erlaubt Änderung von Feldelementen
  - Zum Löschen von Elementen oder des ganzen Feldes: `unset()`

PHP-Datentypen

- Felder (*arrays*)
  - Beispiel:

```
<?php
$arr = array(5 => 1, 12 => 2);

$arr[] = 56;     // entspricht $arr[13] = 56;
$arr["x"] = 42;  // Neues Element mit
                 // Schlüssel "x"
unset($arr[5]);  // Entfernt erstes Element
                 // (mit Wert 1)
unset($arr);     // Löscht das gesamte Feld
?>
```


PHP-Datentypen

- Felder (*arrays*)
  - Umwandlung anderer Datentypen in Felder:
    - Skalare Typen (*boolean*, *integer*, *float*, *string*, *resource*) → Feld mit einem Element, Schlüssel = 0, Wert = Wert des ursprünglichen Datenelements
    - Objekte: Datenelemente → Array-Elemente; Schlüssel = Name des Datenelements
    - NULL → leeres Array
  - Beispiele und Details: ➔ PHP-Dokumentation

PHP-Datentypen

- Objekte (*objects*)
  - Erstellung von Objekten mit new:

```
<?php
class xyz
{
  function hello()
  {
    echo "Hello world!";
  }
}
$obj = new xyz;   // ein Objekt wird erstellt
$obj->hello();    // führt function hello() aus
?>
```


PHP-Datentypen

- Objekte (*objects*)
  - Umwandlungen:
    - *object* → *object*: Keine Änderung
    - Jede andere Datentype: neues Objekt der Type `stdClass`; ursprüngliche Daten sind im Element `scalar` enthalten:

```
<?php
$obj = (object) 'Hallo';
echo $obj->scalar;  // gibt "Hallo" aus
?>
```

PHP-Datentypen

- Ressourcen (*resources*)
  - Referenz auf externe Objekte (z.B. geöffnete Dateien, *Connections* zu Datenbanken, usw.)
  - Typumwandlungen von Ressourcen sind nicht möglich (und sinnvoll)
  - Nicht mehr benötigte Ressourcen werden automatisch freigegeben

PHP-Datentypen

- NULL
  - Einziger Wert der Type NULL ist `NULL` (in Groß- oder Kleinschreibung)
  - Eine Variable hat den Wert `NULL`
    - nach Zuweisung der Konstanten `NULL`
    - wenn ihr noch kein Wert zugewiesen wurde (Zugriff auf nicht initialisierte Variable!)
    - wenn sie explizit mit `unset()` auf `NULL` gesetzt wurde

PHP-Datentypen

- Typenumwandlung
  - Automatisch: bei der Auswertung von Ausdrücken

```
<?php
$a = '0';       // $a ist "0" (string)
$a += 1;        // $a ist 1 (integer)
$a = $a + 3.14; // $a ist 4.14 (float)
$a -= "3.14";   // $a ist 1 (float)
$a .= " Test";  // $a ist "1 Test" (string)
$a += 1;        // $a ist 2 (integer)
?>
```


PHP-Datentypen

- Typenumwandlung
  - Strings werden bis PHP 5 implizit, ab PHP 5 explizit als Felder von Zeichen betrachtet:

```
<?php
$a = "abcde";
$a[0] = "z";   // → $a == "zbcde";
// in PHP 4 empfohlene Notation,
// nicht mit PHP 5+ kompatibel:
$a{0} = "z"; // → $a == "zbcde";
?>
```

PHP-Datentypen

- Typenumwandlung
  - Explizite Typenumwandlung (*type casts*):
    - `(int)`, `(integer)` → *integer*
    - `(bool)`, `(boolean)` → *boolean*
    - `(float)`, `(double)`, `(real)` → *float*
    - `(string)` → *string*
    - `(array)` → *array*
    - `(object)` → *object*

PHP-Datentypen

- Typenumwandlung
  - Explizite Typenumwandlung von Strings:

```
<?php
$a = 4711;          // integer 4711
$b = (string) $a;   // string "4711"
$c = "$a";          // string "4711"
// $b und $c sind identisch
?>
```

Einführung in PHP

- Syntax
- Datentypen
- **Variable**
- Konstanten
- Ausdrücke
- Operatoren
- Ablaufsteuerung
- Funktionen
- Klassen und Objekte
- *Exceptions*
- Referenzen

Variable

- Variablen-Darstellung: „**$**“, gefolgt vom Namen der Variablen
- Alle Namen in PHP (daher auch die von Variablen) sind *case sensitive* (Groß- und Kleinbuchstaben haben unterschiedliche Bedeutung)!
- Namen beginnen mit einem Buchstaben oder „_“ und können eine beliebige Zahl von Buchstaben, Ziffern und „_“ enthalten

Variable

- „Buchstaben“ sind Groß- und Kleinbuchstaben von „a“ bis „z“ sowie die Zeichen mit den ASCII-Codes 127 bis 255 (Unterschied zu den meisten anderen Programmiersprachen!) – „`$Müßiggang`“ ist in PHP ein gültiger Variablenname!

Variable

- Wertzuweisungen:
  - *by value*: Ergebnis eines Ausdrucks wird in Ziel-Variable kopiert; spätere Änderungen der Quell-Variablen im Ausdruck beeinflussen die Ziel-Variable nicht – ab PHP 3
  - *by reference*: Ziel-Variable verweist auf Quell-Variable, ändert ihren Wert mit der Quell-Variablen (Referenzen dürfen nur auf Variable verweisen, nicht auf Ausdrücke oder Funktionen!) – ab PHP 4

Variable

- Wertzuweisungen:
  - Beispiel für Wert-Zuweisung und Referenz:

```
<?php
$a = 'rot';
$b = $a;    // $b enthält Kopie von $a
$c = &$a;   // $c ist Referenz auf $a
$c = 'grün';
echo $c;    // gibt "grün" aus
echo $a;    // gibt auch "grün" aus
echo $b;    // gibt "rot" aus
?>
```

Variable

- Uninitialisierte Variable: Haben einen ihrer Type entsprechenden Default-Wert:
  - Boolean: `false`
  - Integer, Float: `0`
  - String: "" (leerer String)
  - Array: leeres Array
  - Object: leeres Objekt

Variable

- Vordefinierte Variable
  - PHP-*Superglobals* (Arrays globaler Variablen):
    - `$GLOBALS`
    - `$_SERVER` (früher: `$HTTP_SERVER_VARS`)
    - `$_GET` (früher: `$HTTP_GET_VARS`)
    - `$_POST` (früher: `$HTTP_POST_VARS`)
    - `$_COOKIE` (früher: `$HTTP_COOKIE_VARS`)
    - `$_FILES` (früher: `$HTTP_POST_FILES`)
    - `$_ENV` (früher: `$HTTP_ENV_VARS`)
    - `$_REQUEST`
    - `$_SESSION` (früher: `$HTTP_SESSION_VARS`)

Variable

- `$HTTP_...VARS`-Arrays werden ab PHP 5.4 nicht mehr unterstützt!
- Details zu vordefinierten Variablen ➔ PHP-Dokumentation

Variable

- *Scope* von Variablen (Kontext, in dem sie definiert sind)
  - Normalerweise *ein Scope* (Variable im gesamten PHP-Programm einschließlich mit `require` oder `include` eingebundener Dateien sichtbar)
  - Ab PHP 5.3 ist die Definition eigener *Namespaces* möglich ➔ PHP-Dokumentation

Variable

- *Scope* von Variablen
  - Ausnahme: Benutzerdefinierte Funktionen haben ein eigenes lokales *Scope*:

```
<?php
$a = 1;       // $a ist global

function Test()
{
    echo $a;  // $a ist lokal
}

Test();       // keine Ausgabe ($a ist NULL!)
?>
```

Variable

- *Scope* von Variablen
  - Globale Variable innerhalb der Funktion mit `global` deklarieren (beliebig viele möglich!):

```
<?php
$a = 1;          // $a ist global
function Test()
{
    global $a;
    echo $a;     // globales $a!
}

Test();          // Ausgabe: "1"
?>
```


Variable

- *Scope* von Variablen
  - Mit `global` deklarierte Variable können auch verändert werden:

```
<?php
$a = 1;          // $a ist global
function Test()
{
    global $a;
    $a++;        // globales $a!
}
Test();
echo $a;         // Ausgabe: "2"
?>
```

Variable

- *Scope* von Variablen
  - Alternativer Zugriff auf globale Variable aus einer Funktion:

```
<?php
$a = 1;             // $a ist global
function Test()
{
    $GLOBALS['a']++;   // globales $a!
}
Test();
echo $a;            // Ausgabe: "2"
?>
```

Variable

- *Statische* Variable – "überleben" Funktionsaufrufe
  - Beispiel:

```
<?php
function Test()   // zählt "0, 1, 2…"
{
    static $a = 0;
    echo $a;
    $a++;
}
?>
```

  - Statische Variable dürfen nur mit Konstanten initialisiert werden!

Variable

- *Scope* von Referenzen und statischen Variablen: ➔ PHP-Dokumentation

Variable

- Variable Variable
  - Beispiel:

```
<?php
$a = "Hello";    // gewöhnliche Variable
$$a = "world!";  // setzt Variable $Hello

echo "$a $$a";
echo "$a ${$a}";
echo "$a $Hello";
// alle drei echo-Befehle geben aus:
// "Hello world!"
?>
```

Variable

- Bestimmung der Typen von Variablen
  - Diverse PHP-Funktionen:
    - `gettype()`
    - `is_array()`
    - `is_float()`
    - `is_int()`
    - `is_object()`
    - `is_string()`

Einführung in PHP

- Syntax
- Datentypen
- Variable
- **Konstanten**
- Ausdrücke
- Operatoren
- Ablaufsteuerung
- Funktionen
- Klassen und Objekte
- *Exceptions*
- Referenzen

Konstanten

- Konstante = Name für einen einfachen (konstanten) Wert
- Definiert mit dem Sprachkonstrukt `define()`

```
define ("PI", 3.141592);
define ("UNIVERSITÄT", "TU Wien");
```

- Groß-/Kleinschreibung relevant (*case sensitive*)
- Konvention: Konstantennamen in Großbuchstaben

Konstanten

- Konstanten haben globales *Scope* (sie sind überall im Programm sichtbar)
- Gleiche Regeln für Namen wie bei Variablen:
  - Beginnen mit einem Buchstaben oder „_“
  - Können alle Buchstaben von „a“ – „z“, Zeichen mit ASCII-Codes zwischen 127 und 255, Ziffern und „_“ enthalten

Konstanten

- Den Namen von Konstanten darf *kein* „$“ vorangestellt werden!
- Konstanten und (globale) Variable existieren in unterschiedlichen Namensräumen (*name spaces*) → Konstanten dürfen gleiche Namen wie Variable haben!

Konstanten

- Konstanten dürfen nur *skalare* Daten enthalten (*boolean*, *integer*, *float, string*)
- Ab PHP 7 können auch Arrays von Konstanten definiert werden:

```
define('TIERE', ['Hund', 'Katze',
'Vogel']);
echo TIERE[1];    // gibt "Katze" aus
```

Konstanten

- Undefinierte Konstanten werden mit dem Wert ihres Namens interpretiert
- Definierte Konstanten werden durch ihren Wert ersetzt
- „Dynamische“ Ermittlung des Wertes einer Konstanten: Funktion `constant()`

Konstanten

- Unterschiede zwischen Konstanten und Variablen:
  - Konstanten haben kein vorangestelltes „$“
  - Konstanten dürfen nur mit der Funktion `define()` definiert werden, *nicht* durch Zuweisung
  - Konstanten können überall definiert und verwendet werden
  - Konstanten können nicht verändert oder gelöscht werden
  - Konstanten haben (vor PHP 7) nur skalare Werte

Konstanten

- Verwendung von Konstanten:

```
<?php
define("KONSTANTE", "Hello world.");
echo KONSTANTE;   // Ausgabe von "Hello world."
echo Konstante;   // Ausgabe von "Konstante"
                  // und von einer Fehlermeldung
?>
```


Konstanten

- Vordefinierte Konstanten: ➔ PHP-Dokumentation
- „Magische“ Konstanten – werden automatisch gesetzt (z.B. für Debugging!):
  - `__LINE__`: Aktuelle Zeile eines Programms
  - `__FILE__`: Vollständiger Pfad- und Dateiname der PHP-Datei
  - `__FUNCTION__`: Funktionsname (ab PHP 4.3.0)
  - `__CLASS__`: Name der Klasse (ab PHP 4.3.0)
  - `__METHOD__` : Name der Klassen-Methode (ab PHP 5.0.0)

Konstanten

- „Magische“ Konstanten (Fortsetzung):
  - `__DIR__`: Aktuelles Verzeichnis (ab PHP 5.3.0)
  - `__NAMESPACE__`: Name des aktuellen *Namespaces* (ab PHP 5.3.0)
  - `__TRAIT__`: Aktueller *Trait*-Name (einschließlich *Namespace*) (ab PHP 5.4.0)

Einführung in PHP

- Syntax
- Datentypen
- Variable
- Konstanten
- **Ausdrücke**
- Operatoren
- Ablaufsteuerung
- Funktionen
- Klassen und Objekte
- *Exceptions*
- Referenzen

Ausdrücke

- „Alles, was einen Wert hat“:
  - Konstanten und Variablen
  - Funktionen
  - Ausdrücke mit Operatoren
    - Beispiel:

```
<?php
$a = ($b = 2 * 3);
// Ergebnis von "2 * 3" ist Wert des
// Ausdrucks "$b = 2 * 3" und wird $a
// zugewiesen
?>
```


Ausdrücke

- Spezialfall: Inkrement- und Dekrement-Operatoren
  - Präfix-Operatoren:

```
<?php
$a = $b = 2;
$c = ++$a;  // setzt $c und $a auf 3
$d = --$b;  // setzt $d und $b auf 1
?>
```

Ausdrücke

- Spezialfall: Inkrement- und Dekrement-Operatoren
  - Postfix-Operatoren:

```
<?php
$a = $b = 2;
$c = $a++;  // setzt $c auf 2 und $a auf 3
$d = $b--;  // setzt $d auf 2 und $b auf 1
?>
```

Ausdrücke

- Spezialfall: Inkrement- und Dekrement-Operatoren
  - Gilt auch für Argumente in Funktionsaufrufen:

```
<?php

function ausgabe ($x) { echo $x; }

$a = $b = 1;
ausgabe (++$a);  // gibt "2" aus
ausgabe ($b++);  // gibt "1" aus
echo "$a, $b";   // gibt "2, 2" aus
?>
```

Einführung in PHP

- Syntax
- Datentypen
- Variable
- Konstanten
- Ausdrücke
- **Operatoren**
- Ablaufsteuerung
- Funktionen
- Klassen und Objekte
- *Exceptions*
- Referenzen

Operatoren

- Operator-Präzedenz *(Operator Precedence* – „2 + 3*4“ → 14)
- Jeder Operator hat eine fix vorgegebene Position in der nach Präzedenz gereihten Liste der Operatoren (➔ PHP-Dokumentation)

Operatoren

- Arithmetische Operatoren:
  - + … Addition
  - - … Subtraktion
  - * … Multiplikation
  - / … Division (*immer* mit Gleitkommaergebnis!)
  - % … Modulo
  - ** … Exponentiation (ab PHP 5.6)

Operatoren

- Inkrement- und Dekrement- Operatoren:
  - ++$a … Prefix-Inkrement
  - $a++ … Postfix-Inkrement
  - --$a … Prefix-Dekrement
  - $a-- … Postfix-Dekrement

Operatoren

- Stringverbindungs-Operator:
  - . … verbindet zwei Strings

```
<?php
$a = "Hello"." "."world!";
  // $a wird auf "Hello world!" gesetzt
?>
```

Operatoren

- Zuweisungs-Operatoren:
  - Zuweisungs-Operatoren *kopieren* den Wert des Ausdrucks auf der rechten Seite in die Variable auf der linken Seite
  - = … gewöhnliche Zuweisung
    - Beispiel:

```
<?php
$a = ($b = 4) + 5;
// setzt $b auf 4 und $a auf 9
?>
```

Operatoren

- Zuweisungs-Operatoren:
  - *op*= … kombinierter Zuweisungsoperator
  - *op* kann jeder binäre arithmetische und String-Operator sein

Operatoren

- Zuweisungs-Operatoren:
  - *op*= … kombinierter Zuweisungsoperator
    - Beispiel:

```
<?php
$a = 2;
$a += 3;
// entspricht $a = $a + 3; $a → 5
$b = "Hello";
$b .= " world!";
// entspricht $b = $b." world!";
// $b → "Hello world!"
?>
```

Operatoren

- Bitweise Operatoren:
  - & … Und
  - | … Oder
  - ^ … Exklusiv-Oder
  - ~ … Negation
  - << … Linksverschiebung ($a << $b entspricht $a*(2 hoch $b))
  - >> … Rechtsverschiebung ($a >> $b entspricht $a/(2 hoch $b))

Operatoren

- Bitweise Operatoren:
  - Funktionieren auch für Strings:

```
<?php
echo 12 ^ 9;
  // gibt '5' aus (1100b^1001b=0101b)
echo "12" ^ "9";
  // gibt Backspace (ASCII 8) aus:
  // ('1' (ASCII 49)) ^ ('9' (ASCII 57)) = #8
echo "hallo" ^ "hello";
  // gibt die ASCII-Werte #0 #4 #0 #0 #0 aus:
  // 'a' ^ 'e' = #4
?>
```

Operatoren

- Logische Operatoren (verknüpfen Boole'sche Ausdrücke):
  - `and` … Und
  - `or` … Oder
  - `xor` … Exklusiv-Oder
  - `!` … Negation
  - `&&` … Und (höhere Präzedenz als `and`)
  - `||` … Oder (höhere Präzedenz als `or`)

Operatoren

- Vergleichsoperatoren:
  - `==` … Gleich
  - `===` … Identisch (gleich und gleiche Type)
  - `!=` … Ungleich
  - `<>` … Ungleich
  - `!==` … Nicht identisch (ungleich oder unterschiedliche Type)
  - `<=>` … Ternärer Vergleichsoperator („Spaceship operator“ – ab PHP 7)

Operatoren

- Ternärer Vergleichsoperator („Spaceship operator“)
  - Gibt als Ergebnis -1, 0 oder 1 zurück
  - Funktioniert für ganze und Gleitkommazahlen sowie für Strings:

```
echo 1 <=> 1;   // 0
echo 1 <=> 2;   // -1
echo 2 <=> 1;   // 1
```


Operatoren

- Vergleichsoperatoren:
  - > … Größer
  - < … Kleiner
  - >= … Größer oder gleich
  - <= … Kleiner oder gleich

Operatoren

- Ternärer Operator „?:“:
  `$a = ausdr1 ? ausdr2 : ausdr3;`
  - $a → *ausdr2*, wenn *ausdr1* == TRUE
  - $a → *ausdr3*, wenn *ausdr1* == FALSE
- Kurzversion ab PHP 5.3:
  `$a = ausdr1 ?: ausdr3;`
  - $a → *ausdr1*, wenn *ausdr1* == TRUE
  - $a → *ausdr3*, wenn *ausdr1* == FALSE

Operatoren

- Operator für Test auf NULL „??“ (ab PHP 7):
  - Gibt seinen ersten Operanden zurück, wenn dieser gesetzt und nicht NULL ist, ansonsten den zweiten:
    ```
    $user = $_GET['user'] ?? 'niemand';
    ```
    äquivalent zu
    ```
    $user = isset($_GET['user']) ?
    $_GET['user'] : 'niemand';
    ```
  - Kann auch verschachtelt werden:
    ```
    $user = $_GET['user'] ?? $_POST['user']
    ?? 'niemand';
    ```

Operatoren

- Array-Operator „+“:
  - Hängt das Array, das als rechter Operand angegeben ist, an das linke Array an
  - Felder mit gleichnamigen Schlüsseln werden dabei nicht überschrieben

Operatoren

- Array-Operator „+“:
  - Beispiel:

```
<?php
$a = array ("a" => "Apfel", "b" => "Birne");
$b = array ("a" => "Orange", "b" => "Erdbeere",
"c" => "Kirsche");
$c = $a + $b;
var_dump ($c);
?>
```

Operatoren

- Array-Operator „+“:
  - Beispiel (Fortsetzung):

```
<?php
/* var_dump() gibt aus:
array(3) {
   ["a"]=>
   string(5) "Apfel"
   ["b"]=>
   string(5) "Birne"
   ["c"]=>
   string(7) "Kirsche"
} */
?>
```


Operatoren

- Fehlerausgabesteuerungs-Operator „@“:
  - Unterdrückt die Ausgabe von Laufzeit-Fehlermeldungen für jenen (ausführbaren) Ausdruck, dem er vorangestellt wird
  - Hat keinen Einfluss auf die Ausgabe von Fehlermeldungen bei Syntax-Fehlern
  - Wenn die Boole‘sche Konstante `track_errors` in `php.ini` aktiviert ist, wird die letzte Fehlermeldung in der Variablen `$php_errormsg` gespeichert

Operatoren

- Systembefehlsausführungs-Operator „``“:
  - „`“ nicht mit einfachem Anführungszeichen „'“ verwechseln!
  - Führt den zwischen den beiden „`“ eingeschlossenen Systembefehl aus und gibt die Ausgabe des Systembefehls als Ergebnis zurück:

```
<?php
$dirlist = `ls –al`; // Directory-Listing
echo "<pre>$dirlist</pre>\n";
?>
```


Einführung in PHP

- Syntax
- Datentypen
- Variable
- Konstanten
- Ausdrücke
- Operatoren
- **Ablaufsteuerung**
- Funktionen
- Klassen und Objekte
- *Exceptions*
- Referenzen

Ablaufsteuerung

- PHP-Script = Folge von *Befehlen*
- Befehle:
  - Zuweisungen
  - Gruppe von Befehlen
  - Leerer Befehl
  - Bedingte Befehle
  - Schleifen
  - Sonstige Steuerungsfunktionen
  - Funktionsaufrufe

Ablaufsteuerung

- Gruppe von Befehlen
  - Befehle, die in geschwungenen Klammern („{…}“)eingeschlossen sind, werden wie *ein* Befehl behandelt:

```
<?php
if ($x > 0)
{
    echo "\$x ist größer als Null ($x)!\n";
    $x = 0;
    echo "\$x wurde auf Null gesetzt.\n";
}
?>
```

Ablaufsteuerung

- Leerer Befehl
  - Besteht nur aus dem abschließenden „;“ oder aus „{}“:

```
<?php
if ($x >= 0)
    ;        // leerer Befehl
else
    echo "\$x ist kleiner als Null ($x)!\n";
// als Alternative für "if ($x < 0)"
?>
```

Ablaufsteuerung

- Bedingte Ausführung von Programmteilen: `if-elseif-else`-Konstrukt:

```
if (Ausdruck)
    Befehl
```

oder

```
if (Ausdruck)
    Befehl1
else
    Befehl2
```

Ablaufsteuerung

- `if-elseif-else`-Konstrukt (Fortsetzung):

```
if (Ausdruck1)
    Befehl1
elseif (Ausdruck2)
    Befehl2
else
    Befehl3
```


Ablaufsteuerung

- `if-elseif-else`-Konstrukt:
  - Beispiel: Setze `$a` je nach Vorzeichen von `$x` auf `-1`, `0` oder `+1`:

```
<?php
if ($x > 0)
    $a = 1;
elseif ($x == 0)
    $a = 0;
else
    $a = -1;
?>
```

Ablaufsteuerung

- Alternative Syntax (gilt außer für `if` auch für `switch`, `while`, `for` und `foreach`-Konstrukte):
  - Beispiel:

```
<?php if ($note == 1): ?>
<b>Sie bekommen die Note &quot;Sehr
gut&quot;!</b> <br>
Sie haben erfolgreich HTML-Code erstellt, der
nur dann ausgeführt wird, wenn Ihre Note gleich
&quot;1&quot; ist!
<?php endif; ?>
```

Ablaufsteuerung

- Alternative Syntax
  - Bedingter Block beginnt mit „`:`“ statt „`{`“ und endet mit:

```
if:          endif;
switch:      endswitch;
while:       endwhile;
for:         endfor;
foreach:     endforeach;
```

  statt mit „`}`“

Ablaufsteuerung

- Alternative Syntax
  - Beispiel:

```
<?php
if ($a == 5):
    print "a ist gleich 5";
    print "...";
elseif ($a == 6):
    print "a ist gleich 6";
    print "!!!";
else:
    print "a ist weder 5 noch 6";
endif;
?>
```

Ablaufsteuerung

- Bedingte Ausführung von Programmteilen: `switch`-Konstrukt:

```
switch (Ausdruck)
{
    case x:
        Befehle
        break;
    case y:
        Befehle
        break;
    default:
        Befehle
}
```

Ablaufsteuerung

- `switch`-Konstrukt – Beispiel:

```
<?php
switch ($a)
{
    case 1:
        echo "\$a ist gleich 1!\n";
        break;
    case 2:
        echo "\$a ist gleich 2!\n";
        break;
    default:
        echo "\$a ist weder 1 noch 2!\n";
}
?>
```


Ablaufsteuerung

- `switch`-Konstrukt – Beispiel:

```
<?php
switch ($a)
{
    case 1:
    case 2:
        echo "\$a ist 1 oder 2!\n";
    case 3:
        echo "\$a ist 1, 2 oder 3!\n";
        break;
    case 4:
        echo "\$a ist gleich 4!\n";
}
?>
```

Ablaufsteuerung

- `switch`-Konstrukt – alternative Syntax:

```
<?php
switch ($a):
    case 1:
    case 2:
        echo "\$a ist 1 oder 2!\n";
    case 3:
        echo "\$a ist 1, 2 oder 3!\n";
        break;
    case 4:
        echo "\$a ist gleich 4!\n";
endswitch;
?>
```


Ablaufsteuerung

- Schleifen: `while`-Konstrukt:
  `while (`*`Ausdruck`*`)` *`Befehl`*
  - Beispiel:

```
<?php
$i = 1;
echo "<h1>Wir zählen bis 10:</h1>\n";
while ($i <= 10)
{
    echo "<b>".$i."</b><br>\n";
    $i++;
}
?>
```

Ablaufsteuerung

- Schleifen: `while`-Konstrukt:
  - Beispiel für alternative Syntax:

```
<?php
$i = 0;
echo "<h1>Quadrate ganzer Zahlen:</h1>\n";
while ($i < 100):
    $j = $i * $i;
    echo "Das Quadrat von $i ist $j.<br>\n";
    $i++;
endwhile;
?>
```

Ablaufsteuerung

- Schleifen: `do while`-Konstrukt:
  `do` *`Befehl`* `while (`*`Ausdruck`*`)`
  - Beispiel:

```
<?php
$i = 0;
echo "<h1>Wir zählen bis 10:</h1>\n";
do
    echo "<b>".(++$i)."</b><br>\n";
while ($i < 10);
?>
```

  - Keine alternative Syntax!

Ablaufsteuerung

- Schleifen: `for`-Konstrukt:
  `for (`*`Befehl1; Ausdruck; Befehl2`*`)` *`Befehl3`*
  - Beispiel:

```
<?php
echo "<h1>Wir zählen bis 10:</h1>\n";
for ($i = 1; $i <= 10; $i++)
    echo "<b>$i</b><br>\n";

// oder: Zählen in Zweierschritten von 2 – 20
for ($i=1, $j=2; $i<=10; print $i*$j, $i++);
// Befehl3 ist hier der leere Befehl
?>
```


Ablaufsteuerung

- Schleifen: `for`-Konstrukt:
  - Beispiel für alternative Syntax:

```
<?php
echo "<h1>Quadrate ganzer Zahlen:</h1>\n";
for ($i = 0; $i < 100; $i++):
    $j = $i * $i;
    echo "Das Quadrat von $i ist $j.<br>\n";
endfor;
?>
```

Ablaufsteuerung

- Endlosschleifen:
  - Beispiele:

```
<?php
while (1)
{
    echo "Ich laufe...<br>\n";
    if (abbruch())  // eine Abbruchbedingung
        break;
}
?>
```


Ablaufsteuerung

- Endlosschleifen:
  - Beispiele:

```
<?php
for (;;)
{
    echo "Ich laufe auch...<br>\n";
    if (abbruch())  // eine Abbruchbedingung
        break;
}
?>
```

Ablaufsteuerung

- Schleifen: `foreach`-Konstrukt:
  `foreach(`*`Feld-Ausdruck`*` as $value)` *`Befehl`*
  `foreach(`*`Feld-Ausdruck`*` as $key => $value)` *`Befehl`*
  - `foreach`-Schleifen durchlaufen alle Elemente eines Feldes und weisen die Werte der Elemente der Variablen `$value` und optional die zugehörigen Schlüssel der Variablen `$key` zu
  - Erweiterte Funktionalität in zukünftigen Versionen von PHP vorgesehen
  - Nähere Details: ➔ PHP-Dokumentation

Ablaufsteuerung

- Abbruch von Schleifen und `switch`-Konstrukten: `break`
  - `break` erlaubt einen optionalen numerischen Parameter (z.B. „`break 3;`“), der angibt, aus wie vielen ineinander geschachtelten Schleifen oder `switch`-Konstrukten herausgesprungen werden soll
  - Der numerische Parameter muss aber konstant und darf nicht gleich 0 sein!

Ablaufsteuerung

- `break` – Beispiel:

```
<?php
for ($i = 0; ; $i++)  // würde endlos laufen
{
   switch ($i)
   {
      case 10:
         echo "10 geschafft!<br>\n";
         break;   // oder: break 1;
      case 20:
         echo "Fertig!<br>\n";
         break 2; // verlässt for-Schleife
   }
}
?>
```

Ablaufsteuerung

- Überspringen des Restes einer Schleife: `continue`
  - `continue` erlaubt einen optionalen numerischen Parameter (z.B. „`continue 3;`“), der angibt, in wie vielen ineinander geschachtelten Schleifen an das Ende der Schleife gesprungen werden soll (Details ➔ PHP-Dokumentation

Ablaufsteuerung

- `continue` – Beispiel:

```
<?php
for ($i = 0; $i < 100; $i++)
{
    echo "Wir sind bei $i angelangt.<br>\n";
    if ($i % 7)  // nur für Vielfache von 7 false
        continue;
    echo "$i ist ein Vielfaches von 7!<br>\n";
}
?>
```

Ablaufsteuerung

- Rücksprung aus einer Funktion oder eingebundenem Programmcode: `return`
  `return [`*`Ausdruck`*`]` oder `return([`*`Ausdruck`*`])`
  - Innerhalb einer Funktion: `return` beendet die Ausführung der Funktion und gibt den Wert des optionalen Ausdrucks als Ergebnis zurück:

```
<?php
function quadrat ($x)
{
    return $x*$x;
}
?>
```

Ablaufsteuerung

- `return`
  - Im globalen *Scope*: `return` beendet das Script
  - In einem mit `include()` oder `require()` eingebundenen Script: `return` beendet das eingebundene Script und gibt den Wert des Ausdrucks als Ergebnis von `include()` zurück
- Klammern bei `return()` sind optional

Ablaufsteuerung

- Sprung mit `goto` (ab PHP 5.3):
  - Sprungziel muss innerhalb derselben Datei und desselben Kontexts (Funktion, Konstrukt…) liegen!
  - Sprung aus einer Schleife heraus ist aber zulässig
  - Der *Label* (das Sprungziel) muss eine Konstante sein und wird case sensitive interpretiert
  - Details: ➔ PHP-Dokumentation

Ablaufsteuerung

- Einbinden anderer Skript-Dateien: `require` und `include`
  `require` *`string`* oder `require(`*`string`*`)`
  `include` *`string`* oder `include(`*`string`*`)`
  - `require` und `include` verhalten sich identisch; Ausnahme bei Nicht-Verfügbarkeit der eingebundenen Datei:
    - `require`: Fataler Fehler
    - `include`: Parser-Warnung

Ablaufsteuerung

- `require` und `include`
  - Die Klammern bei `require()` und `include()` sind optional
  - Wird `require` oder `include` in einer Schleife aufgerufen, so wird die Datei nur *einmal* eingebunden, aber der Code bei jedem Schleifendurchlauf ausgeführt
  - Mit `require` oder `include` eingebundener Code übernimmt das *Scope*, innerhalb dessen `require` oder `include` aufgerufen wurden

Ablaufsteuerung

- `require` und `include` – Beispiel (Teil 1):

Datei `incl.php`:

```
<?php
$a = "Include-";
$b = "Nr. 1";
?>
```

Datei `test.php`:

```
<?php
echo "{$a}Test $b";    // "Test "
include "incl.php";
echo "{$a}Test $b";    // "Include-Test Nr. 1"
?>
```


Ablaufsteuerung

- `require` und `include` – Beispiel (Teil 2):

Datei `test1.php`:

```
<?php
function func()
{
    global $a;
    include ('incl.php');
    echo "{$a}Test $b";
}

func();                 // "Include-Test Nr. 1"
echo "{$a}Test $b";     // "Include-Test "
?>
```

Ablaufsteuerung

- `require` und `include`
  - Bei bedingt ausgeführten `require`- oder `include`-Befehlen diese in „`{ ... }`“ setzen:

```
<?php
if ($bedingung)
{
    include "file_a.php";
}
else
{
    include "file_b.php";
}
?>
```


Ablaufsteuerung

- `require` und `include`
  - Die einzubindende Datei kann über einen absoluten Pfad oder ohne Pfad-Angabe spezifiziert werden.
  - (Nur) im letzteren Fall sucht PHP *erst*
    - in dem mit der System-Variablen `include_path` (jedenfalls `"."`) spezifizierten Pfad relativ zum aktuellen Arbeitsverzeichnis, und *dann*
    - in dem mit `include_path` spezifizierten Pfad relativ zum Verzeichnis des aktuellen Skripts
  - Damit sind interessante Konstrukte realisierbar.

Ablaufsteuerung

- `require` und `include`
  - Die eingebundene Datei kann auch eine HTTP-URL sein: ➔ PHP-Dokumentation
  - Beim Einbinden einer Datei fällt der Parser vom PHP- in den HTML-Mode – „<?php ... ?>“-*Tags* nicht vergessen!
  - Eingebundener Code kann ab PHP 4 mit `return` einen Wert zurückgeben → Ergebnis von `require()` oder `include()` (➔ PHP-Dokumentation)

Ablaufsteuerung

- Einmaliges Einbinden anderer Skript-Dateien: `require_once()` und `include_once()`
  - Gleiche Syntax und Funktionalität wie `require` und `include`; einzubindende Datei wird jedoch *genau einmal* eingebunden (➔ PHP-Dokumentation)

Ablaufsteuerung

- Ausführungs-Direktiven für einen Block von Code festlegen: `declare`
  - Details ➔ PHP-Dokumentation

Einführung in PHP

- Syntax
- Datentypen
- Variable
- Konstanten
- Ausdrücke
- Operatoren
- Ablaufsteuerung
- **Funktionen**
- Klassen und Objekte
- *Exceptions*
- Referenzen

Funktionen

- Definition von Funktionen
  - Funktionen dürfen beliebigen Code, einschließlich der Definition neuer Funktionen und Klassen, enthalten
  - Für Namen von Funktionen gelten die gleichen Regeln wie für Variablen-Namen; sie werden aber *case insensitive* behandelt!
  - *Nicht* unterstützt werden *Function Overloading* und Änderungen der Definition einmal definierter Funktionen

Funktionen

- Definition von Funktionen
  - Voreingestellte Argumente seit PHP 3, Argumentlisten mit variabler Länge seit PHP 4 verfügbar
  - Funktionen haben unabhängig vom Ort ihrer Definition immer globales *Scope* (also auch innerhalb einer Funktion definierte Funktionen!)
  - Rekursive Aufrufe sind zulässig; Rekursionstiefe sollte nicht über 100 – 200 liegen

Funktionen

- Definition von Funktionen
  - In PHP 3 mussten Funktionen definiert sein, bevor sie aufgerufen werden konnten, nicht jedoch ab PHP 4
  - Ausnahme: Funktionen, die innerhalb anderer Funktionen oder bedingter Code-Blöcke (z.B. in mit bedingten `include`-Befehlen eingebundenen Dateien) definiert werden, sind erst verfügbar, nachdem der Code ausgeführt wurde, innerhalb dessen sie definiert sind

Funktionen

- Funktionsargumente
  - Beliebige Zahl von Argumenten (einschließlich null), getrennt durch Kommas
  - Übergabe von Argumenten als
    - Wert (*by value*) – Default
    - Referenz (*by reference*)
    - Voreingestellte Argumente

Funktionen

- Argumentübergabe als Wert
  - Funktion verwendet eine *Kopie* des Arguments:

```
<?php
function test ($arg)
{
    echo "\$i ist ".++$arg;
}

$i = 1;
test ($i);              // "$i ist 2"
echo "\$i ist ".$i;     // "$i ist 1"
?>
```


Funktionen

- Argumentübergabe als Referenz
  - Funktion ändert das *Original* des Arguments:

```
<?php
function test (& $arg)
{
    echo "\$i ist ".++$arg;
}

$i = 1;
test ($i); // KEIN "&"! - "$i ist 2"
echo "\$i ist ".$i;     // "$i ist 2"
?>
```

Funktionen

- Voreingestellte Argumente
  - Argumente können einen Standardwert haben:

```
<?php
function test ($i = 3, $j = 5)
{
    echo "\$i = $i, \$j = $j\n";
}

test (1, 2);      // "$i = 1, $j = 2"
test (1);         // "$i = 1, $j = 5"
test ();          // "$i = 3, $j = 5"
?>
```


Funktionen

- Voreingestellte Argumente
  - Im Aufruf nicht definierte Argumente werden auf den voreingestellten Wert gesetzt
  - Voreingestellte Argumente werden immer vom Ende der Argumentliste ausgehend angewendet
  - In unserem Beispiel ist es daher nicht möglich, den voreingestellten Wert von `$i` zu verwenden, wenn nur *ein* Argument übergeben wurde – Argumentlisten entsprechend sortieren!
  - Ab PHP 5: auch voreingestellte Werte für Referenz-Argumente

Funktionen

- Argumentlisten mit variabler Länge
  - Ab PHP 4 verfügbar
  - Details ➔ PHP-Dokumentation

Funktionen

- Rückgabewerte
  - `return` mit einem beliebigen Ausdruck:

```
<?php
function quadrat ($num)
{
    return $num * $num;
}

echo quadrat (4);      // gibt "16" aus
?>
```

Funktionen

- Rückgabewerte
  - Rückgabe mehrerer Werte als Array:

```
<?php
function primzahlen ()
{
    return array (1, 2, 3, 5, 7, 11, 13);
}

$a = primzahlen ();
echo $a[4];      // gibt "7" aus
?>
```

Funktionen

- Rückgabewerte
  - Rückgabe von Referenzen – „&“ vor dem Funktionsnamen in *Definition und Aufruf* der Funktion:

```
<?php
function & ref_test ()
{
    return $eine_referenz;
}

$neue_referenz = & ref_test ();
?>
```

Funktionen

- Variable als Funktionen
  - Beispiel:

```
<?php
function test1 () { echo "Test #1"; }
function test2 () { echo "Test #2"; }
function test3 () { echo "Test #3"; }
$a = 'test1';
$a();         // ruft Funktion test1 auf
$a = 'test2';
$a();         // ruft Funktion test2 auf
$a = 'test3';
$a();         // ruft Funktion test3 auf
?>
```


Funktionen

- Variable als Funktionen
  - Gilt auch analog für Methoden einer Klasse (➔ PHP-Dokumentation)

Funktionen

- Eingebaute Funktionen
  - Zahlreiche „eingebaute“ Funktionen von PHP (➔ PHP-Dokumentation)
  - Manche „eingebauten“ Funktionen benötigen spezielle „Extensions“ von PHP (z.B. Funktionen für Zugriffe auf Datenbanken, Funktionen zur Bearbeitung von Bildern) (➔ PHP-Dokumentation)
  - Eingebaute Funktionen verwenden compilierten (C- oder C++-) Code → viel bessere Performance als selbst programmierte Script-Funktionen!

Einführung in PHP

- Syntax
- Datentypen
- Variable
- Konstanten
- Ausdrücke
- Operatoren
- Ablaufsteuerung
- Funktionen
- **Klassen und Objekte**
- *Exceptions*
- Referenzen

Klassen und Objekte

- Klasse: Sammlung von Variablen und Funktionen, die auf diese Variablen zugreifen
- Objekt: Variable, die nach dem „Bauplan“ einer Klasse erstellt wurde und auf die Funktionen der Klasse zugreifen kann
- Klassen müssen *definiert* werden, bevor sie verwendet werden können
- Klassen und Objekte werden bis PHP 4 und ab PHP 5 unterschiedlich behandelt. PHP 5 hat ein volles Objekt-Modell!

Klassen und Objekte

- Beispiel für Definition einer Klasse (Teil 1):

```
<?php
class Einkaufswagen
{
   public $waren;  // Waren im Einkaufswagen

   // $n Artikel mit Nummer $artnr hinzufügen

   public function dazu ($artnr, $n)
   {
      $this->waren[$artnr] += $n;
   }
// ... weiter auf nächster Seite
```

Klassen und Objekte

- Beispiel für Definition einer Klasse (Teil 2):

```
    // $n Artikel mit $artnr herausnehmen
    public function heraus ($artnr, $n)
    {
        if ($this->waren[$artnr] >= $n)
        {
            $this->waren[$artnr] -= $n;
            return true;
        }
        else
            return false;
    }
}
?>
```


Klassen und Objekte

- Klassen- und Funktionsnamen:
  - Klasse `stdClass` wird intern von PHP verwendet und ist reserviert
  - Funktionsnamen, die mit „___“ (2 x „_“!) beginnen, werden als Namen „magischer“ Funktionen interpretiert und gelten als reserviert
  - „Magische“ Funktionen müssen vom Programmierer erstellt werden; sie werden unter bestimmten Voraussetzungen automatisch ausgeführt

Klassen und Objekte

- Initialisierung von Datenelementen einer Klasse:
  - Direkt mit einer Konstanten
  - Unter Verwendung einer Konstruktor-Funktion mit beliebigen Ausdrücken
  - Ab PHP 5 ist der Name der Konstruktor-Funktion mit „`__construct()`“ festgelegt

Klassen und Objekte

- Initialisierung von Datenelementen einer Klasse:

```
<?php
class Test
{
   public $a = 4711;          // mit einer Konstanten
   public $b = array ("Apfel", "Birne"); // konstant!
   public $c, $d;
   public function __construct()    // Konstruktor
   {
      $this->c = date ("Y-m-d");
      $this->d = $_SERVER["HTTP_USER_AGENT"];
   }
}
?>
```

Klassen und Objekte

- Initialisierung mit globalen Variablen:

```
<?php
$string = "Hello World!";
class Test0
{
    public $a;
    public $b;
    public function __construct()
    {
        $this->a = $string;          // falsch!
        $this->b = $GLOBALS['string'];
    }
}
?>
```


Klassen und Objekte

- Initialisierung mit globalen Variablen – alternative Methode:

```
<?php
$string = "Hello World!";
class Test0
{
    public $a;
    public $b;
    public function __construct()
    {
        global $string;
        $this->a = $string;
    }
}
?>
```

Klassen und Objekte

- Konstruktor mit voreingestelltem Argument:

```
<?php
class Test1
{
    var $a;
    public function __construct($arg = 3.14)
    {
        $this->a = $arg;
    }
}
?>
```

- Konstruktoren *können* Argumente haben, sollten aber auch ohne solche aufgerufen werden können

Klassen und Objekte

- Objekte einer Klasse:
  - Unterschiedliche Daten-Inhalte, aber gleiche Funktionen
  - Erstellt mit new:

```
<?php
$x = new Test1;
$y = new Test1("Hello World!");

echo $x->a;    // Ausgabe: "3.14"
echo $y->a;    // Ausgabe: "Hello World!"
?>
```

Klassen und Objekte

- Objekte einer Klasse:
  - Bei der Erstellung eines Objekts einer Klasse mit `new` wird, sofern vorhanden, der Konstruktor der Klasse automatisch ausgeführt
  - Der Konstruktor kann aber im Fehlerfall die Erstellung eines neuen Objekts verhindern (mittels einer *Exception*)

Klassen und Objekte

- Datenelemente und Klassenelement-Funktionen:

```
<?php
$wagen1 = new Einkaufswagen;
$wagen1->dazu ("4711", 12);
$wagen1->heraus ("4711", 2);

$wagen2 = new Einkaufswagen;
$wagen2->dazu ("0815", 100);
$wagen2->heraus ("0815", 69);
?>
```

Klassen und Objekte

- Auswahl von Datenelementen und Klassenelement-Funktionen: Operator „->“
  - Nur *ein* „$“!:

```
<?php
$wagen1->dazu(...);  // NICHT $wagen1->$dazu
echo $x->a;          // NICHT $x->$a

// das ist aber zulässig
$element = "a";
echo $x->$element;
?>
```


Klassen und Objekte

- Auswahl von Datenelementen und Klassenelement-Funktionen:
  - Innerhalb der Klassendefinition: Pseudo-Variable `$this` („Aktuelles Objekt“)

```
<?php
...
function dazu ($artnr, $n)
{
    $this->waren[$artnr] += $n;
}
...
?>
```

Klassen und Objekte

- Bei Zuweisung eines mit new erstellten Objekts an eine Variable verweist diese ab PHP 5 auf das *gleiche* Objekt:

```
<?php
$wagen = new Einkaufswagen();
$wagen1 = $wagen;
$wagen->dazu(4711, 10);
echo $wagen1->waren[4711]; // gibt "10" aus!
?>
```

Klassen und Objekte

- Gilt auch für Pseudo-Variable `$this`:

```
<?php
function dazu ($artnr, $n)
{
    $c = $this;
    $c->waren[$artnr] += $n;
}
?>
```

- „Echte“ Kopien mit `clone` (➔ PHP-Dokumentation)

```
<?php
$wagen2 = clone $wagen;
?>
```

Klassen und Objekte

- Abgeleitete Klassen: `extends`
  - Übernehmen („erben“) Datenelemente und Funktionen einer Basisklasse
  - Führen zusätzliche Funktionalität (Datenelemente, Funktionen) ein
  - Objekte einer abgeleiteten Klasse können daher auf die Datenelemente und Funktionen der Basisklasse *und* auf die der abgeleiteten Klasse zugreifen

Klassen und Objekte

- Abgeleitete Klassen: `extends`
  - Von einer abgeleiteten Klasse können wiederum beliebig viele Klassen mit `extends` abgeleitet werden
  - Regel: Wird ein Datenelement oder eine Funktion nicht in der eigenen Klasse gefunden, wird in der Basisklasse danach gesucht

Klassen und Objekte

- Abgeleitete Klassen: Beispiel (1. Teil):

```
<?php
class Mein_Einkaufswagen extends Einkaufswagen
{
    public $gehört;

    public function set_gehört ($name)
    {
        $this->gehört = $name;
    }
}
?>
```


Klassen und Objekte

- Abgeleitete Klassen: Beispiel (2. Teil):

```
<?php
$wagen = new Mein_Einkaufswagen;
$wagen->set_gehört("Karl");
    // Funktion aus abgeleiteter Klasse

$wagen->dazu("1234", 12);
    // Funktion aus Basisklasse

echo $wagen->gehört;
    // Datenelement aus abgeleiteter Klasse
?>
```

Klassen und Objekte

- Abgeleitete Klassen: `extends`
  - Nur einfache Ableitung (nur *eine* Basisklasse)
  - Gleichnamige Methoden oder Datenelemente in abgeleiteter Klasse ersetzen jene der Basisklasse, außer Methode in der Basisklasse wurde mit `final` deklariert
  - Zugriff auf Methoden oder Datenelemente der Basisklasse mit `parent::`

Klassen und Objekte

- Abgeleitete Klassen: `parent::` (Teil 1):

```
<?php
class Basis
{
   public function melde() { echo "Basis<br>"; }
}
class Abgeleitet extends Basis
{
   public function melde()
   {
      echo "Abgeleitet<br>";
      parent::melde();
   }
}
```

Klassen und Objekte

- Abgeleitete Klassen: `parent::` (Teil 2):

```
$a = new Abgeleitet();
$a->Melde();

// Ausgabe:
// Abgeleitet
// Basis
```


Klassen und Objekte

- Automatisches Laden von Dateien mit Definitionen von Klassen und Objekten – `__autoload()`:
  - `__autoload()`-Funktion kann selbst definiert werden; wird am Beginn jedes Scripts ausgeführt:

```
<?php
function __autoload($Klasse)
{
   require_once $Klasse.'.php';
}
$obj1 = new Class1();  // lädt "Class1.php"
$obj2 = new Class2();  // lädt "Class2.php"
?>
```

Klassen und Objekte

- Konstruktoren – `__construct()`:

```
<?php
class Basis {
  function __construct() { echo "Basisklasse\n"; }
}
class Abgeleitet extends Basis {
  function __construct() {
    parent::__construct();
    echo "Abgeleitete Klasse\n";
    }
}
$obj = new Basis();
$obj1 = new Abgeleitet(); ?>
```


Klassen und Objekte

- Konstruktoren werden immer bei der Erstellung eines neuen Objekts aufgerufen.
- Konstruktoren abgeleiteter Klassen rufen *nicht* den Konstruktor der Basisklasse auf – selbst explizit aufrufen („`parent::__construct();`“)
- Wenn keine Funktion `__construct()` existiert, interpretiert PHP 5 eine Methode mit gleichem Namen wie die Klasse, in der sie definiert ist, als Konstruktor (Kompatibilität zu PHP 4)
- Diese Kompatibilitäts-Funktionalität wird in PHP 7 nicht mehr unterstützt!

Klassen und Objekte

- Destruktoren – `__destruct()`:
  - Werden ausgeführt, wenn alle Referenzen auf ein Objekt entfernt sind (spätestens am Ende des Scripts)
  - Können auch explizit aufgerufen werden
  - Destruktoren abgeleiteter Klassen rufen *nicht* den Destruktor der Basisklasse auf – `parent::destruct()` explizit im Destruktor der abgeleiteten Klasse aufrufen!

Klassen und Objekte

- Sichtbarkeit – `public`, `protected` und `private`:
  - `public`: Mit `public` deklarierte Elemente und Methoden sind von überall sichtbar und in abgeleiteten Klassen neu definierbar.
  - `protected`: Mit `protected` deklarierte Elemente und Methoden sind nur innerhalb der Klasse sowie in von dieser abgeleiteten Klassen sichtbar und können dort auch neu definiert werden.

Klassen und Objekte

- Sichtbarkeit – `public`, `protected` und `private`:
  - `private`: Mit `private` deklarierte Elemente und Methoden sind nur innerhalb der Klasse sichtbar. Sie können in abgeleiteten Klassen nicht neu definiert werden!

Klassen und Objekte

- Sichtbarkeit:
  - `var`: Das PHP 4-Schlüsselwort „`var`“ wird von PHP 5 als identisch zu „`public`“ interpretiert.
  - `var` führt aber in PHP 5 vor 5.1.3 zu einer `E_STRICT`-Warnung!
  - In PHP 7 wird `var` noch immer als reserviertes Keyword geführt (trotz Entfernung der PHP 4-Kompatibilität); seine Interpretation ist aber derzeit nicht spezifiziert.

Klassen und Objekte

- Statische Elemente und Methoden – `static`:
  - Sind ohne Objekt der Klasse verfügbar
  - Sichtbarkeit (Default „`public`“) muss vor „`static`“ stehen
  - Statische Datenelemente dürfen nur unter Verwendung des Klassennamens und von „`::`“ angesprochen werden, statische Methoden auch unter Verwendung eines Objekts der Klasse

Klassen und Objekte

- Klassenkonstanten – `const`:
  - Echte Konstanten (ohne „$“ in der Definition)
  - Dürfen nur mit Klassennamen und „`::`“ angesprochen werden

Klassen und Objekte

- *Scope Resolution* – `::`, `self` und `parent`:
  - „`::`“ = „*Paamayim Nekudotayim*“ (Hebräisch für „`::`“)
  - `self` und `parent` beziehen sich auf die aktuelle und die Basisklasse

```
<?php
class Basis
{
   const Konstante = 'Eine Konstante';
}
echo Basis::Konstante;
// Fortsetzung auf nächster Seite
```


Klassen und Objekte

- *Scope Resolution* – `::`, `self` und `parent`:

```
class Abgeleitet extends Basis
{
   public static $statisch = 'statische Variable';
   public static function Paamayim() {
      echo parent::Konstante . "\n";
      echo self::$statisch . "\n";
   }
}

Abgeleitet::Paamayim();
?>
```

Klassen und Objekte

- Abspeichern von Objekten: `serialize()` und `unserialize()`
  - `serialize()` fasst die Datenelemente eines Objekts für nachfolgende Abspeicherung zusammen
  - `unserialize()` erstellt aus den mit `serialize()` abgespeicherten Daten ein neues Objekt
  - Nur Datenelemente und der Klassenname, nicht aber Funktionen werden abgespeichert

Klassen und Objekte

- Abspeichern von Objekten: `serialize()` und `unserialize()`
  - Vor dem Aufruf von `unserialize()` muss die zugehörige Klasse definiert worden sein (`include()` oder `require()` verwenden!)

Klassen und Objekte

- `serialize()` und `unserialize()`
  - Datei ClassA.inc:

```
<?php
class A
{
    public $eins = 1;
    public function show ()
    {
        echo $this->eins;
    }
}
?>
```

Klassen und Objekte

- `serialize()` und `unserialize()`
  - Datei `Seite1.php`:

```
<?php
include ("ClassA.inc");
$a = new A;
$a->eins = 2;
$s = serialize ($a);
// Abspeichern von $s in einer Datei "store"
$fp = fopen ("store", "w");
fputs ($fp, $s);
fclose ($fp);
?>
```

Klassen und Objekte

- `serialize()` und `unserialize()`
  - Datei `Seite2.php`:

```
<?php
include ("ClassA.inc");
$s = implode ("", @file ("store"));
/* file() liest die Datei in ein Array (eine
   Zeile in ein Element); implode() macht einen
   String daraus; einfacher:
   $s = @file_get_contents ("store"); */
$a = unserialize ($s);
// $a ist nun ein Objekt der Klasse A
$a->show(); // gibt "2" aus
?>
```


Klassen und Objekte

- Abspeichern von Objekten:
  - Bei Verwendung von Sessions und der Funktion `session_register()` werden Objekte automatisch am Ende jeder PHP-Seite mit `serialize()` abgespeichert und zu Beginn jeder Seite mit `unserialize()` wieder hergestellt
  - Konsequenz: Alle Klassen müssen auf allen Seiten definiert sein, egal ob dort verwendet oder nicht! (`include()` verwenden!)
  - `session_register()` wird ab PHP 5.4 aber nicht mehr unterstützt

Klassen und Objekte

- Abspeichern von Objekten: `__sleep()` und `__wakeup()`
  - Wenn in der Definition einer Klasse vorhanden, wird von `serialize()` vor dem Zusammenpacken der Daten `__sleep()` aufgerufen
  - Wenn in der Definition einer Klasse vorhanden, wird von `unserialize()` nach dem Entpacken der Daten `__wakeup()` aufgerufen

Klassen und Objekte (PHP 3, PHP 4)

- (Reduzierte) OOP-Funktionalität bereits ab PHP 3 verfügbar
- Wesentliche Unterschiede zu PHP 5:
  - Keine steuerbare Sichtbarkeit; Deklaration von Datenelementen mit `var` (entspricht `public` in PHP 5) und von Klassenelement-Funktionen mit `function` (ohne Zusatz)
  - Objekte werden kopiert und nicht als Referenz übergeben

Klassen und Objekte (PHP 3, PHP 4)

- Wesentliche Unterschiede zu PHP 5 (Fortsetzung):
  - Initialisierung von Datenelementen der Klasse:
    - PHP 4: Konstruktor = Funktion mit gleichem Namen wie die Klasse, *in der sie definiert ist*.
    - PHP 3: Konstruktor = Funktion mit gleichem Namen wie die *aktuelle* Klasse.
  - Keine Destruktoren
  - Einschränkungen bei Klassenkonstanten und statischen Methoden

Klassen und Objekte

- Weitere PHP 5-OOP-Features (➔ PHP-Dokumentation)
  - Abstrakte Klassen („`abstract`“)
  - Templates („`interface`“, „`implements`“)
  - Überladen von Datenelementen („`__get`“, „`__set`“, „`__isset`“ und „`__unset`“)
  - Überladen von Methoden („`__call`“)
  - Objekt-Iterationen (mit z.B. `foreach`; `Iterator`)
  - *Patterns* (`factory()`, `singleton()`)
  - *Reflection Classes*
  - *Type Hinting*

Klassen und Objekte

- Magische Methoden – automatisch in bestimmter Situation aufgerufen (➔ PHP-Dokumentation):
  - `__construct()`, `__destruct()`
  - `__call()`, `__get()`, `__set()`, `__isset()`, `__unset()`
  - `__sleep()`, `__wakeup()`
  - `__toString()`
  - `__set_state()`
  - `__clone()`
  - `__autoload()`

Klassen und Objekte

- Features objektorientierter Programmierung im Vergleich

Feature	PHP 3	PHP 4	PHP 5+	C++
Klassen und Objekte	Ja	Ja	Ja	Ja
Einfache Ableitung	Ja	Ja	Ja	Ja
Mehrfache Ableitung	Nein	Nein	Nein	Ja
Steuerbare Sichtbarkeit	Nein	Nein	Ja	Ja
Konstruktoren	Ja	Ja	Ja	Ja
Destruktoren	Nein	Nein	Ja	Ja
Autom. Konstruktor-/Destruktoraufruf der Basisklasse	Nein	Nein	Nein	Ja

Klassen und Objekte

- Features objektorientierter Programmierung im Vergleich

Feature	PHP 3	PHP 4	PHP 5+	C++
Überladene Funktionen	Nein	Nein	Ja	Ja
Überladene Operatoren	Nein	Nein	Nein	Ja
Kopierkonstruktoren	Nein	Nein	Ja	Ja
Templates	Nein	Nein	Ja	Ja
Statische Datenelemente und Funktionen	Nein	Nein	Ja	Ja

Einführung in PHP

- Syntax
- Datentypen
- Variable
- Konstanten
- Ausdrücke
- Operatoren
- Ablaufsteuerung
- Funktionen
- Klassen und Objekte
- ***Exceptions***
- Referenzen

Exceptions

- Ab PHP 5 Funktionalität zur Behandlung von *Exceptions*
- Eingebaute *Exception*-Klasse, kann durch eigene abgeleitete *Exception*-Klassen erweitert werden
- In den neueren PHP 5-Versionen und in PHP 7 werden in zunehmendem Maß interne Laufzeitfehlermeldungen durch *Exceptions* ersetzt
- ➔ PHP-Dokumentation

Einführung in PHP

- Syntax
- Datentypen
- Variable
- Konstanten
- Ausdrücke
- Operatoren
- Ablaufsteuerung
- Funktionen
- Klassen und Objekte
- *Exceptions*
- **Referenzen**

Referenzen

- Eine Referenz verweist auf den gleichen Speicherplatz wie die Variable, mit der sie initialisiert wurde:

```
<?php
$a = 4711;
$b = & $a;   // $b ist eine Referenz
    // $b und $a zeigen auf gleiche Daten
$b++;
echo $a;     // gibt "4712" aus
?>
```


Referenzen

- Referenzen können auch mit Ausdrücken mit new und mit Funktionen, die Referenzen als Ergebnis liefern, initialisiert werden (➔ PHP-Dokumentation)
- Referenzen mit „&“ auf Ergebnis von new sind in PHP 5 redundant (weil new dort eine Referenz generiert) → E_STRICT-Fehlermeldung
- Typische Verwendung: Übergabe von Werten an Funktionen
  - Effizienter (Argumente werden nicht kopiert)
  - Aktuelle Argumente können von der Funktion global verändert werden

Referenzen

- Funktions-*Aufruf* mit Referenz:

```
<?php
function func1 ($arg)    // KEINE Referenz
{ echo ++$arg; }

function func2 (& $arg)  // Referenz-Funktion
{ echo ++$arg; }

$a = $b = 0;
func1 ($a);   // gibt "1" aus
func2 ($b);   // gibt auch "1" aus
echo $a;      // gibt "0" aus
echo $b;      // gibt "1" aus
?>
```


Referenzen

- Funktions-Aufruf mit Referenzen – zulässige Argumente:
  - Variable (`func2($a)`)
  - Mit new erstellte Objekte (`func2(new Klasse)`)
  - Funktionen, die Referenz-Ergebnisse haben
  - Alle anderen Argumente sind ungültig!
- Bei Funktionen mit Referenz-Argumenten muss bei der *Deklaration* der Funktion der „&“-Operator angegeben werden (`function func2 (& $arg)`); eine Angabe des „&“-Operators beim *Aufruf* ist ab PHP 5.3 nicht mehr zulässig und ein Syntax-Fehler!

Referenzen

- Funktionen mit Referenz-*Ergebnis*:

```
<?php
function & func () // Achtung: "&"
{
    static $a;
    return $a;     // eine REFERENZ auf $a
}

$b = & func();     // Achtung: wieder "&"
    // $b zeigt auf gleiche Daten wie $a
$b = "Hello world!";
$c = & func();
echo $c;           // gibt "Hello world!" aus
?>
```


Dynamische Erstellung von Webseiten mit PHP

- Allgemeines zu PHP
- Die Geschichte von PHP
- Kompatibilität zwischen PHP-Versionen
- Einführung in PHP
- **Erstellung und Testen von PHP-Programmen**

Erstellung von PHP-Programmen

- Erstellung: Mit jedem beliebigen ASCII-Editor
- Zweckmäßige Editor-Funktionen:
  - *Syntax Highlighting*
  - Überprüfungsmöglichkeiten für runde „( )“, eckige „[ ]“ und geschwungene Klammern „{}“
  - Unterstützung für Erstellung von HTML-*Tags*
  - Sinnvolle Tabulator-Funktionen

Erstellung von PHP-Programmen

- Verwendung von *Web-Authoring*-Software (*Adobe Dreamweaver*, *Microsoft Expression Web*)
  - Hilfreich bei Erstellung komplexer HTML-Seiten (mit wenig eingebettetem PHP-Code)
  - Funktionalität zum Editieren des HTML-Quellcodes muss aber vorhanden sein!
  - *Dreamweaver* bietet volle Editier-Funktionalität für HTML- und PHP-Code

Erstellung von PHP-Programmen

- Hinweise zur Gestaltung des Quellcodes
  - Lesbarkeit des *PHP-Quellcodes* sicherstellen
    - Einrückungen
    - Zeilenvorschübe
    - Kommentare

Erstellung von PHP-Programmen

- Hinweise zur Gestaltung des Quellcodes
  - Lesbarkeit des gesamten (Original- und PHP-generierten) *HTML-Codes* sicherstellen (für Debugging-Zwecke):
    - Bei PHP-generiertem HTML-Code Zeilenvorschübe (mit „`echo "\n";`“) vorsehen
    - Einrückungen im HTML-Code (z.B. bei Tabellen) sparsam verwenden – mühsam aus PHP heraus zu generieren!

Erstellung von PHP-Programmen

- Hinweise zur Gestaltung des Quellcodes
  - Bei Seiten mit großem Anteil an PHP-Code ist es oft übersichtlicher, den gesamten HTML-Seitencode aus PHP heraus zu erstellen (statt oftmaligem Wechsel zwischen HTML und PHP)

Testen von PHP-Programmen

- Typische Fehlerquellen
- Syntaktischer Test
- Logischer Test

Testen von PHP-Programmen

- **Typische Fehlerquellen**
- Syntaktischer Test
- Logischer Test

Typische Fehlerquellen

- Vergessen:
  - Schließende Anführungszeichen
  - Strichpunkte
  - Geschwungene Klammern
- Schreibfehler in Variablennamen
  - Schreibfehler → neue uninitialisierte Variable (= `NULL`)
  - Nicht automatisch detektierbar (keine Deklaration von Variablen!) – mit `E_NOTICE` verursacht Lesen einer uninitialisierten Variablen aber eine Fehlermeldung

Typische Fehlerquellen

- Irrtümliche Mehrfach-Verwendung der selben (globalen) Variablen
  - (Prozedurale) PHP-Programme sind oft nicht strukturiert – gesamter Programmcode liegt im globalen *Scope*
  - (Speziell für C-/C++-Programmierer:) Vorsicht bei Laufvariablen in ineinander geschachtelten Schleifen!

Typische Fehlerquellen

- Verwendung von PHP-Funktionen
  - Verhalten von PHP-Funktionen in Grenzbereichen ihrer Definition kann sich bei Wechsel zwischen Software-Versionen unerwartet ändern!
  - Ganze Familien von Funktionen können bei Wechsel zu einer neuen PHP-Version nicht mehr verfügbar sein (POSIX *Regular Expression*, klassische MySQL-Funktionen)!
  - Syntax-Änderungen bei Versionswechsel beachten (Deklaration von Klassen-Datenelementen `var` → `public`; Zeichenauswahl aus Strings `[]` → `{}` → `[]`)

Testen von PHP-Programmen

- Typische Fehlerquellen
- **Syntaktischer Test**
- Logischer Test

Syntaktischer Test

- Erfolgt automatisch bei der Ausführung einer Seite
  - Webserver mit aktiviertem PHP-Plug-In ist erforderlich!
  - Häufiger Fehler: PHP-Seiten werden einfach im Browser geöffnet – PHP-Code wird dabei nicht ausgeführt!
  - Lokal laufende Entwicklungsumgebung (z.B. XAMPP) kann hilfreich bis unersetzbar sein!

Syntaktischer Test

- PHP-Parser liest den *gesamten* PHP-Code, auch Code-Teile in bedingten Konstrukten
- Im Fall syntaktischer Fehler wird die Seitenausgabe mit einer Fehlermeldung abgebrochen
- Vorsicht: Im Zweifelsfall HTML-Quellcode scheinbar leerer Ausgabeseiten anschauen!
- Partiell ausgegebene Seiten können seit PHP 4 aber nur mehr passieren bei
  - Fatalen Laufzeitfehlern
  - Syntax-Fehlern in bedingt eingebundenen Dateien

Syntaktischer Test

- PHP-Fehlermeldungen enthalten Zeilennummern des PHP-Quellcodes
- Fehler werden aber oft erst einige Zeilen unter der fehlerhaften entdeckt (z.B. bei fehlenden schließenden Anführungszeichen oder Strichpunkten)

Testen von PHP-Programmen

- Typische Fehlerquellen
- Syntaktischer Test
- **Logischer Test**

Logischer Test

- Überprüfung des generierten HTML-Codes
- Ausgabe der Werte von Variablen
- Spezielle Debugging-Methoden

Logischer Test

- Überprüfung des generierten HTML-Codes
  - Im Browser Quelltext der Seite anzeigen lassen
  - Manche Browser zeigen unvollständige Seitenelemente (z.B. Tabellen) überhaupt nicht an!
  - Quelltext der Seite enthält oft Hinweise auf Ablauf des PHP-Programms und/oder Werte von PHP-Variablen

Logischer Test

- Ausgabe der Werte von Variablen
  - Einfach (z.B. „echo“-Befehle)
  - Gezielt bei „Problemstellen“ einsetzbar
  - ABER: Probleme bei Ausgaben in bestimmten HTML-Konstrukten (z.B. Tabellen)
  - Seiten-Design wird empfindlich gestört

Logischer Test

- Ausgabe der Werte von Variablen
  - Variante: Werte von kritischen Parametern werden (mehr oder weniger) permanent als HTML-Kommentar ausgegeben (Vorsicht: Sicherheitsrisiko!):

```
<?php
if ($debug)
    echo "<!-- x = $x -->\n";
?>
```

  - Testausgaben, die im Programmcode verbleiben sollen, eventuell wie oben „abschaltbar“ machen

Logischer Test

- Ausgabe der Werte von Variablen
  - Ausgabezeilen in unvollständigen Tabellen o.ä. werden oft nicht oder nicht an der erwarteten Stelle angezeigt – HTML-Quellcode inspizieren!
  - Ausgaben, die *vor* einem Aufruf der Funktion `header()` (Übergabe von HTTP-Parametern oder Aufruf einer neuen HTML- oder PHP-Seite) erfolgen, haben fatale Fehlermeldungen zur Folge (und `header()` wird nicht ausgeführt) – eventuell *Output Buffering* aktivieren

Logischer Test

- *Output Buffering*
  - Seiteninhalt wird (zur Verbesserung der Performance) erst dann zum Client geschickt, wenn eine bestimmte Datenmenge erreicht ist oder die Seite komplett erstellt wurde
  - In neueren Implementierungen von PHP ist *Output Buffering* auf 4 KBytes gesetzt
  - *Output Buffering* kann aber für manche Anwendungen kontraproduktiv sein (z.B. bei Status-Ausgaben während umfangreicher Operationen)

Logischer Test

- Spezielle Debugging-Methoden
  - Debugging-Ausgaben nicht auf HTML-Seite, sondern in eine Hilfs-Datei:

```
<?php
function debuglog ($line)
{
  $fp = fopen("debug.log","a");
  fwrite($fp,date("H:i:s")." Zeile $line\r\n");
  fclose($fp);
}
?>
```

Logischer Test

- Spezielle Debugging-Methoden
  - An geeigneten Stellen der PHP-Seite dann:

```
<?php
// beliebiger PHP-Programmcode
debuglog(__LINE__.": x=$x, y=$y, z=$z");
?>
```

  - Sinnvoll: Zeilennummer und Zeit angeben
  - Debug-Datei kann mit jedem ASCII-Editor (oder Browser) eingesehen werden

Logischer Test

- Spezielle Debugging-Methoden
  - Testen auf nicht initialisierte Variable (`NULL`):
    - In der Debug-Ausgabe Variable mit irgendwelchen Zeichen „einklammern“:
      `echo "*$x*";`
    - Programm mit Error-Setting `E_STRICT | E_NOTICE` ausführen → Fehlermeldung bei undefinierten Variablen (auch bei undefinierten Aufruf-Parametern!)

Logischer Test

- Spezielle Debugging-Methoden
  - Testen auf nicht initialisierte Variable (`NULL`):
    - Die Werte `NULL`, `""`, `"0"` und `0` können, müssen sich aber nicht gleich verhalten (je nach Kontext) – fallweise Prüfung auf Identität verwenden:

```
if ($x === NULL) echo "\$x = NULL";
if ($x === "") echo "\$x = leerer String";
```

    - Identitäts-Prüfung („`===`“) und nicht Gleichheits-Operator („`==`“) verwenden: Typumwandlung bewirkt z.B. „`"" == 0` → `TRUE`“!