The PHP Language

Internet Applications, ID1354

Contents

Section

- Introduction to PHP
- Types, Operators and Expressions
- Arrays
- Functions
- The Object Model
- Namespaces
- Exception Handling
- Documentation With PHPDoc

The PHP Language

- PHP development was started by Rasmus Lerdorf in 1994.
- Developed to allow him to track visitors to his web site.

The PHP Language

- PHP development was started by Rasmus Lerdorf in 1994.
- Developed to allow him to track visitors to his web site.

- PHP is an open-source product, developed by the PHP group.

The PHP Language

- PHP development was started by Rasmus Lerdorf in 1994.
- Developed to allow him to track visitors to his web site.

- PHP is an open-source product, developed by the PHP group.
- PHP was originally an acronym for Personal Home Page, but later became PHP Hypertext Preprocessor.

Overview of PHP

- By far the most used server-side programming language.

Overview of PHP

- By far the most used server-side programming language.
- Dynamically typed, like JavaScript.

Overview of PHP

- By far the most used server-side programming language.
- Dynamically typed, like JavaScript.
- Purely interpreted, like JavaScript.

Overview of PHP

- By far the most used server-side programming language.
- Dynamically typed, like JavaScript.
- Purely interpreted, like JavaScript.
- Object-oriented with class-based inheritance, like Java, but using objects is optional.

Overview of PHP

- By far the most used server-side programming language.
- Dynamically typed, like JavaScript.
- Purely interpreted, like JavaScript.
- Object-oriented with class-based inheritance, like Java, but using objects is optional.
- PHP files can contain HTML and PHP.

Overview of PHP

- By far the most used server-side programming language.
- Dynamically typed, like JavaScript.
- Purely interpreted, like JavaScript.
- Object-oriented with class-based inheritance, like Java, but using objects is optional.
- PHP files can contain HTML and PHP.
- PHP files have extension **`.php`**

Overview of PHP

- By far the most used server-side programming language.
- Dynamically typed, like JavaScript.
- Purely interpreted, like JavaScript.
- Object-oriented with class-based inheritance, like Java, but using objects is optional.
- PHP files can contain HTML and PHP.
- PHP files have extension `.php`
- There are many different versions of PHP, and they differ quite a lot. This presentation follows the latest versions.

Installation

- The PHP interpreter must be integrated in the web server.
  - Therefore, installation depends on server, see `http://php.net/manual/en/install.php` for instructions.

Installation

- The PHP interpreter must be integrated in the web server.
  - Therefore, installation depends on server, see `http://php.net/manual/en/install.php` for instructions.
- Consider installing a WAMP/LAMP/MAMP pack.

Installation

- The PHP interpreter must be integrated in the web server.
  - Therefore, installation depends on server, see `http://php.net/manual/en/install.php` for instructions.
- Consider installing a WAMP/LAMP/MAMP pack.
  - The first letter is you operating system (**W**indows, **L**inux or **M**acOS).

Installation

- The PHP interpreter must be integrated in the web server.
  - Therefore, installation depends on server, see `http://php.net/manual/en/install.php` for instructions.
- Consider installing a WAMP/LAMP/MAMP pack.
  - The first letter is you operating system (**W**indows, **L**inux or **M**acOS).
  - The other letters means **A**pache, **M**ySQL and **P**HP. These together form a complete web server.

Installation

- The PHP interpreter must be integrated in the web server.
  - Therefore, installation depends on server, see `http://php.net/manual/en/install.php` for instructions.
- Consider installing a WAMP/LAMP/MAMP pack.
  - The first letter is you operating system (**W**indows, **L**inux or **M**acOS).
  - The other letters means **A**pache, **M**ySQL and **P**HP. These together form a complete web server.
  - EasyPHP , **`http://www.easyphp.org/`**, is a WAMP pack that is easy to install.

Syntax

- A PHP file can contain both PHP and HTML.

Syntax

- A PHP file can contain both PHP and HTML.
- HTML is passed to the browser, PHP is executed on the server, and the resulting output is passed to the browser.

Syntax

- A PHP file can contain both PHP and HTML.
- HTML is passed to the browser, PHP is executed on the server, and the resulting output is passed to the browser.
- PHP code is embedded between **`<?php`** and **`?>`** tags.
  - You might want to omit the closing tag since it produces a space in the output.

The First Example

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title></title>
    </head>
    <body>
        <h1>
            <?php
            echo 'Mixing PHP and'
            . ' HTML in the same '
            . 'file this way gives '
            . 'bad cohesion. TRY TO'
            . ' AVOID THAT!';
            ?>
        </h1>
    </body>
</html>
```

Accessing PHP Files

- A PHP file is accessed with a HTTP request with a matching URL, just like a HTML file is accessed,
  `http://myserver.se/path/to/thephpfile.php`

Accessing PHP Files

- A PHP file is accessed with a HTTP request with a matching URL, just like a HTML file is accessed, `http://myserver.se/path/to/thephpfile.php`
- Execution just starts from the beginning of the specified PHP file, there is nothing like a main method.

Accessing PHP Files (Cont'd)

- To call code in other PHP files, it is necessary to include those files.

Accessing PHP Files (Cont'd)

- To call code in other PHP files, it is necessary to include those files.
- Files are included with the **`include`** construct, **`include anotherfile.php;`**.

Accessing PHP Files (Cont'd)

- To call code in other PHP files, it is necessary to include those files.
- Files are included with the **`include`** construct, **`include anotherfile.php;`**.
- The interpreter will look for files at the specified file path, at specified include paths, in the calling file's directory, and in the current working directory.

Accessing PHP Files (Cont'd')

- **`include`** will emit a warning if it cannot find a file. There is also the **`require`** construct which works like **`include`** but emits a fatal error if the specified file is not found.

Accessing PHP Files (Cont'd')

- **`include`** will emit a warning if it cannot find a file. There is also the **`require`** construct which works like **`include`** but emits a fatal error if the specified file is not found.
- **`include_once`** and **`require_once`** works like **`include`** and **`require`**, except that the same file is included only once even if it is specified in multiple inclusion statements.

Naming Conventions

- There is no globally accepted naming convention as in for example Java, but the following is quite common.

Naming Conventions

- There is no globally accepted naming convention as in for example Java, but the following is quite common.
- Class and interface names are written in PascalCase, **`MyFirstClass`**

Naming Conventions

- There is no globally accepted naming convention as in for example Java, but the following is quite common.
- Class and interface names are written in PascalCase, `MyFirstClass`
- Method names are written in camelCase, `myFirstMethod`

Naming Conventions

- There is no globally accepted naming convention as in for example Java, but the following is quite common.
- Class and interface names are written in PascalCase, **`MyFirstClass`**
- Method names are written in camelCase, **`myFirstMethod`**
- Functions, which are methods placed outside classes, are named with underscore, **`my_first_function`**

Naming Conventions

- There is no globally accepted naming convention as in for example Java, but the following is quite common.
- Class and interface names are written in PascalCase, `MyFirstClass`
- Method names are written in camelCase, `myFirstMethod`
- Functions, which are methods placed outside classes, are named with underscore, `my_first_function`
- Variables are named with underscore, `my_first_var`, both inside and outside classes.

Naming Conventions (Cont'd)

- Constant names are written in upper case with underscores,
  `MY_FIRST_CONSTANT`

Naming Conventions (Cont'd)

- Constant names are written in upper case with underscores, **`MY_FIRST_CONSTANT`**
- Namespace, which corresponds to packages, are named in CamelCase, **`MyFirstNamespace`**

Section

- Introduction to PHP
- Types, Operators and Expressions
- Arrays
- Functions
- The Object Model
- Namespaces
- Exception Handling
- Documentation With PHPDoc

Comments

There are three different kinds of comments:

```
// Single line comment

# Single line comment

/*
   Multiple line comment
*/
```

Identifiers

- A valid identifier starts with a letter or underscore, followed by any number of letters, numbers, or underscores.

Identifiers

- A valid identifier starts with a letter or underscore, followed by any number of letters, numbers, or underscores.
- Identifiers are case sensitive.

Variables

- Variables are represented by a dollar sign, **$**, followed by the name of the variable.

Variables

- Variables are represented by a dollar sign, `$`, followed by the name of the variable.
- There are no variable declarations, PHP is dynamically typed.
  - A variable is created and assigned an appropriate type when it is first used, much the same way as in JavaScript.

Variables

- Variables are represented by a dollar sign, `$`, followed by the name of the variable.
- There are no variable declarations, PHP is dynamically typed.
  - A variable is created and assigned an appropriate type when it is first used, much the same way as in JavaScript.
- A variable that has never been assigned a value is unbound and has the value **`NULL`**

Variables

- Variables are represented by a dollar sign, `$`, followed by the name of the variable.
- There are no variable declarations, PHP is dynamically typed.
  - A variable is created and assigned an appropriate type when it is first used, much the same way as in JavaScript.
- A variable that has never been assigned a value is unbound and has the value **`NULL`**
- The **`unset`** function sets a variable to **`NULL`**

Variables

- Variables are represented by a dollar sign, `$`, followed by the name of the variable.
- There are no variable declarations, PHP is dynamically typed.
  - A variable is created and assigned an appropriate type when it is first used, much the same way as in JavaScript.
- A variable that has never been assigned a value is unbound and has the value **`NULL`**
- The **`unset`** function sets a variable to **`NULL`**
- The **`isset`** function is used to determine whether a variable is **`NULL`**

Predefined Variables

- PHP has many predefined variables, that are always available to a PHP program, for example:

Predefined Variables

- PHP has many predefined variables, that are always available to a PHP program, for example:
- **`$_GET`** An array with all HTTP GET variables.

Predefined Variables

- PHP has many predefined variables, that are always available to a PHP program, for example:
- **`$_GET`** An array with all HTTP GET variables.
- **`$_POST`** An array with all HTTP POST variables.

Predefined Variables

- PHP has many predefined variables, that are always available to a PHP program, for example:
- **`$_GET`** An array with all HTTP GET variables.
- **`$_POST`** An array with all HTTP POST variables.
- **`$_SESSION`** An array with all session variables.

Predefined Variables

- PHP has many predefined variables, that are always available to a PHP program, for example:
- **`$_GET`** An array with all HTTP GET variables.
- **`$_POST`** An array with all HTTP POST variables.
- **`$_SESSION`** An array with all session variables.
- **`$_COOKIE`** An array with all HTTP Cookies.

Predefined Variables

- PHP has many predefined variables, that are always available to a PHP program, for example:
- **`$_GET`** An array with all HTTP GET variables.
- **`$_POST`** An array with all HTTP POST variables.
- **`$_SESSION`** An array with all session variables.
- **`$_COOKIE`** An array with all HTTP Cookies.
- These are called superglobals, and are always accessible, regardless of scope

Constants

- Constants can be defined with the **`const`** and **`define`** constructs.

Constants

- Constants can be defined with the **`const`** and **`define`** constructs.
- The following two examples are equal.

```
define("GREETING", "Hello world");
echo GREETING;
```

```
const GREETING = "Hello World";
echo GREETING;
```

Constants

- Constants can be defined with the **const** and **define** constructs.
- The following two examples are equal.

```
define("GREETING", "Hello world");
echo GREETING;
```

```
const GREETING = "Hello World";
echo GREETING;
```

- Note that constant names are not prefixed with **$**

Magic Constants

- There are built-in magic constants that are always available.

Magic Constants

- There are built-in magic constants that are always available.
- Some magic constants follow.

Magic Constants

- There are built-in magic constants that are always available.
- Some magic constants follow.

__FILE__ Path and name of the currently executing PHP file.

Magic Constants

- There are built-in magic constants that are always available.
- Some magic constants follow.

__FILE__	Path and name of the currently executing PHP file.
__DIR__	Path to directory with the currently executing PHP file.

Magic Constants

- There are built-in magic constants that are always available.
- Some magic constants follow.

__FILE__	Path and name of the currently executing PHP file.
__DIR__	Path to directory with the currently executing PHP file.
__FUNCTION__	Name of the currently executing function.

Types

- There are eight different types.

Types

- There are eight different types.
  - Four scalar types, **`boolean`** (**`true`** or **`false`**; 0, “”, and “0” are false, others true) **`integer`** (platform-dependent size), **`double`** (platform-dependent size), **`string`**.

Types

- There are eight different types.
  - Four scalar types, **`boolean`** (**`true`** or **`false`**; 0, “”, and “0” are false, others true) **`integer`** (platform-dependent size), **`double`** (platform-dependent size), **`string`**.
  - Two compound types, **`array`** and **`object`**.

Types

- There are eight different types.
  - Four scalar types, **`boolean`** (**`true`** or **`false`**; 0, “”, and “0” are false, others true) **`integer`** (platform-dependent size), **`double`** (platform-dependent size), **`string`**.
  - Two compound types, **`array`** and **`object`**.
  - Two special types, **`resource`** (a reference to an external resource, like a database) and **`NULL`** (the value of an unbound variable).

Types

- There are eight different types.
  - Four scalar types, **`boolean`** (**`true`** or **`false`**; 0, "", and "0" are false, others true) **`integer`** (platform-dependent size), **`double`** (platform-dependent size), **`string`**.
  - Two compound types, **`array`** and **`object`**.
  - Two special types, **`resource`** (a reference to an external resource, like a database) and **`NULL`** (the value of an unbound variable).
- To print type and value of an expression, use the **`var_dump`** function.

Types

- There are eight different types.
  - Four scalar types, **`boolean`** (**`true`** or **`false`**; 0, "", and "0" are false, others true) **`integer`** (platform-dependent size), **`double`** (platform-dependent size), **`string`**.
  - Two compound types, **`array`** and **`object`**.
  - Two special types, **`resource`** (a reference to an external resource, like a database) and **`NULL`** (the value of an unbound variable).
- To print type and value of an expression, use the **`var_dump`** function.
- To get a human-readable representation of a type, use the **`gettype`** function.

Types

- There are eight different types.
  - Four scalar types, **`boolean`** (**`true`** or **`false`**; 0, "", and "0" are false, others true) **`integer`** (platform-dependent size), **`double`** (platform-dependent size), **`string`**.
  - Two compound types, **`array`** and **`object`**.
  - Two special types, **`resource`** (a reference to an external resource, like a database) and **`NULL`** (the value of an unbound variable).
- To print type and value of an expression, use the **`var_dump`** function.
- To get a human-readable representation of a type, use the **`gettype`** function.
- To check for a certain type, use the **`is_`**<type> functions.

The String Type

- A string consists of one-byte characters.

The String Type

- A string consists of one-byte characters.
- Variables and escape sequences are not expanded with single-quoted string literals.

```
$a = 2;
echo 'The value is \n $a';
// Prints: The value is \n $a
```

The String Type

- A string consists of one-byte characters.
- Variables and escape sequences are not expanded with single-quoted string literals.

```
$a = 2;
echo 'The value is \n $a';
// Prints: The value is \n $a
```

- Variables and escape sequences are expanded with double-quoted string literals.

```
$a = 2;
echo "The value is \n $a";
// Prints: The value is
// 2
```

The String Type

- A string consists of one-byte characters.
- Variables and escape sequences are not expanded with single-quoted string literals.

```
$a = 2;
echo 'The value is \n $a';
// Prints: The value is \n $a
```

- Variables and escape sequences are expanded with double-quoted string literals.

```
$a = 2;
echo "The value is \n $a";
// Prints: The value is
// 2
```

- Note that **\n** is expanded to a line break, not to a **
** tag.

String Concatenation

- The string concatenation operator is a dot, .

```
$what = "Hello";
$who =  "World!";
echo $what . " " . $who;
```

Arithmetic Operators

- The usual `+`, `-`, `*`, `/` and `%`.

Arithmetic Operators

- The usual `+`, `-`, `*`, `/` and `%`.
- If the result of integer division is not an integer, a double is returned.

Arithmetic Operators

- The usual `+`, `-`, `*`, `/` and `%`.
- If the result of integer division is not an integer, a double is returned.
- Any integer operation that results in overflow produces a double.

Arithmetic Operators

- The usual `+`, `-`, `*`, `/` and `%`.
- If the result of integer division is not an integer, a double is returned.
- Any integer operation that results in overflow produces a double.
- The modulus operator, `%`, coerces its operands to integer.

Arithmetic and String Functions

- Some available arithmetic functions are **`floor`**, **`ceil`**, **`round`**, **`abs`**, **`min`**, **`max`**, **`rand`**.

Arithmetic and String Functions

- Some available arithmetic functions are **`floor`**, **`ceil`**, **`round`**, **`abs`**, **`min`**, **`max`**, **`rand`**.
- Some available string functions are **`strlen`**, **`strcmp`**, **`strpos`**, **`substr`**, **`strlen`**, **`chop`**, **`trim`**.

Cast (Explicit Conversion)

- Three ways to specify an explicit conversion.

```
(int)$total
intval($total)
settype($total, "integer")
```

Output

- Output from the PHP program is included in the out stream from server to browser.

Output

- Output from the PHP program is included in the out stream from server to browser.
- There are tree ways to generate output. The first two, **`print`** and **`echo`**, differ only in that **`print`** has a return value.

```
$what = "Hello";
$who =  "World!";
echo $what . " " . $who;
print($what . " " . $who);
```

Output

- Output from the PHP program is included in the out stream from server to browser.
- There are tree ways to generate output. The first two, **`print`** and **`echo`**, differ only in that **`print`** has a return value.

```
$what = "Hello";
$who =  "World!";
echo $what . " " . $who;
print($what . " " . $who);
```

- The third way, **`printf`**, has the same formatting flags as the C function **`printf`**.

Debug Output to Console

- Output for development purposes, for example to track the flow through the program, should not appear in the web page.

Debug Output to Console

- Output for development purposes, for example to track the flow through the program, should not appear in the web page.
- Such output should be directed to the JavaScript console. The following function creates JavaScript code that prints the specified string to the console.

```
function cons($param) {
    echo "<script>" .
            "console.log('$param');" .
         "</script>";
}
```

Error Messages From Interpreter

- The PHP interpreter's output, for example exception reports, goes to the web server's log.

Error Messages From Interpreter

- The PHP interpreter's output, for example exception reports, goes to the web server's log.
- The location of that log depends on server and operating system.
  - On my Ubuntu/Apache platform, the log is in **`/var/log/apache2/error.log`**.

Error Messages From Interpreter

- The PHP interpreter's output, for example exception reports, goes to the web server's log.
- The location of that log depends on server and operating system.
  - On my Ubuntu/Apache platform, the log is in `/var/log/apache2/error.log`.
- It is **strongly recommended** to locate this log since that is where you will see if your PHP program crashed.

Error Messages From Interpreter

- The PHP interpreter's output, for example exception reports, goes to the web server's log.
- The location of that log depends on server and operating system.
  - On my Ubuntu/Apache platform, the log is in `/var/log/apache2/error.log`.
- It is **strongly recommended** to locate this log since that is where you will see if your PHP program crashed.
- Remember that PHP programs are not compiled. The only way to be notified of coding errors is through the above mentioned log.

Relational and Logical Operators

- The relational operators are the same as in JavaScript, including `===` and `!==`.

Relational and Logical Operators

- The relational operators are the same as in JavaScript, including `===` and `!==`.
- The logical operators comes in two flavors. The difference is that number one has higher precedence than assignment operators while number two has lower.
  1. `&&`, `!` and `||`
  2. `and`, `or` and `xor`

Control Statements

- The following control statements behave as in Java, **`if`**, **`else`**, **`else if`**, **`while`**, **`do-while`**, **`for`**, **`switch`**, **`break`** and **`continue`**.

Control Statements

- The following control statements behave as in Java, **`if`**, **`else`**, **`else if`**, **`while`**, **`do-while`**, **`for`**, **`switch`**, **`break`** and **`continue`**.
- There is also the **`foreach`** statement which is different from Java. It will be covered below, after arrays.

Section

- Introduction to PHP
- Types, Operators and Expressions
- Arrays
- Functions
- The Object Model
- Namespaces
- Exception Handling
- Documentation With PHPDoc

The Array Type

- Not like arrays of any other language.

The Array Type

- Not like arrays of any other language.
- A PHP array is actually an ordered map.
  - A map is a type that associates values to keys.
  - Ordered means elements are located at indexes.

The Array Type

- Not like arrays of any other language.
- A PHP array is actually an ordered map.
  - A map is a type that associates values to keys.
  - Ordered means elements are located at indexes.
- This means arrays can be used for many different data structures, like lists and hash tables.

To Create an Array

- Arrays can be created with the **`array()`** construct, which takes comma-separated **`key => value`** pairs as arguments.

```
$my_array = array(
    3       => "value1",
    "key2" => 38
)
```

To Create an Array

- Arrays can be created with the **`array()`** construct, which takes comma-separated **`key => value`** pairs as arguments.

```
$my_array = array(
    3       => "value1",
    "key2" => 38
)
```

- Arrays can also be created with the short array syntax, **`[]`**

```
$my_array = [
    3       => "value1",
    "key2" => 38
]
```

Internal Array Structure

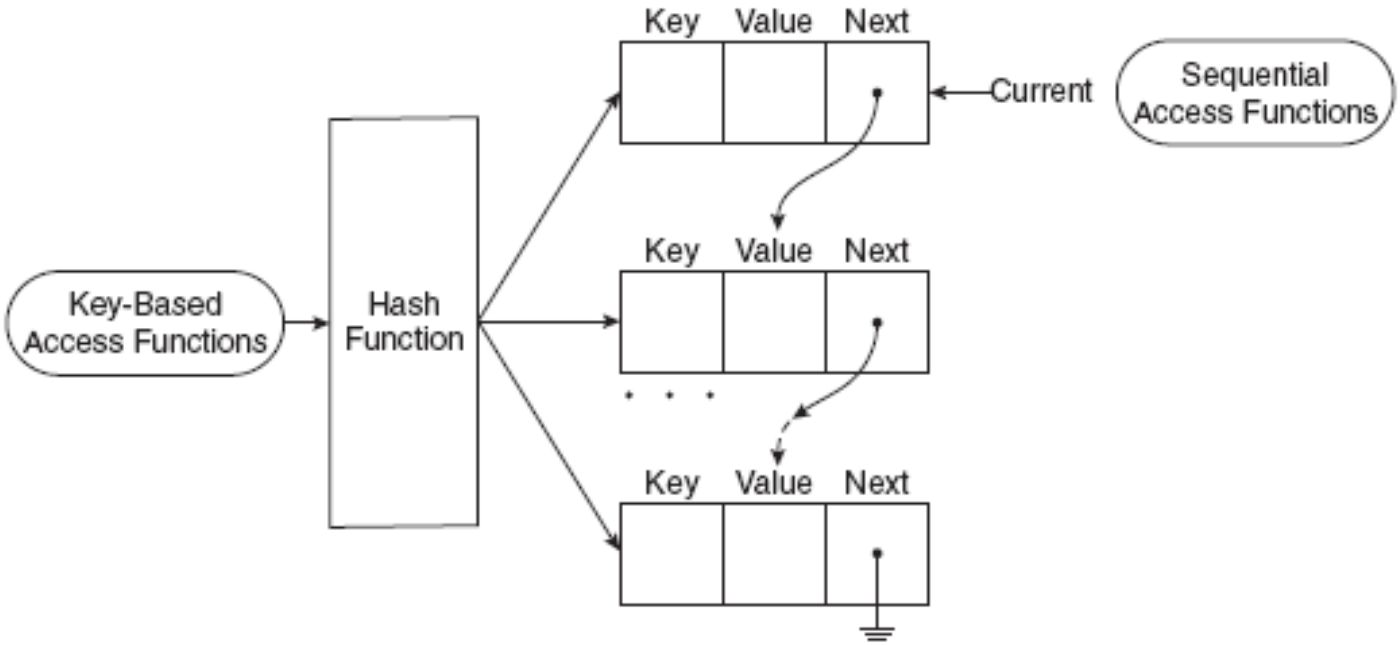

Figure from Sebesta: *Programming the World Wide Web*

Keys and Values

- The key must be an integer or a string, the value can be any type.

Keys and Values

- The key must be an integer or a string, the value can be any type.
- Omitted keys, as below, are assigned the integer that is nearest higher than the highest previous integer key, or zero if there is no previous integer key.

```
$array = array("foo", "bar", "hi", "there");
```

Keys and Values

- The key must be an integer or a string, the value can be any type.
- Omitted keys, as below, are assigned the integer that is nearest higher than the highest previous integer key, or zero if there is no previous integer key.

```
$array = array("foo", "bar", "hi", "there");
```

- Assigning to a key that already has a value means the old value is overwritten.

Access Elements Using Brackets

- Array elements are accessed using brackets.

Access Elements Using Brackets

- Array elements are accessed using brackets.
- If an element with the specified key does not exist, it is created.

Access Elements Using Brackets

- Array elements are accessed using brackets.
- If an element with the specified key does not exist, it is created.
- If the array itself does not exist, it is created.

```
$arr[1] = "hi";
echo "$arr[1]"; // Prints: hi
```

Some Array Functions

`array_keys($arr)` Extracts all keys.

Some Array Functions

`array_keys($arr)` Extracts all keys.

`array_values($arr)` Extracts all values.

Some Array Functions

`array_keys($arr)` Extracts all keys.

`array_values($arr)` Extracts all values.

`array_key_exists($arr)` Tests if there is a key with the specified value.

Some Array Functions

`array_keys($arr)` Extracts all keys.

`array_values($arr)` Extracts all values.

`array_key_exists($arr)` Tests if there is a key with the specified value.

`sizeof($arr)` Returns the number of elements.

Some Array Functions

`array_keys($arr)` Extracts all keys.

`array_values($arr)` Extracts all values.

`array_key_exists($arr)` Tests if there is a key with the specified value.

`sizeof($arr)` Returns the number of elements.

`explode($delim, $str)` Returns an array with the elements of the string `$str` split at the delimiter `$delim`

Some Array Functions

`array_keys($arr)` Extracts all keys.

`array_values($arr)` Extracts all values.

`array_key_exists($arr)` Tests if there is a key with the specified value.

`sizeof($arr)` Returns the number of elements.

`explode($delim, $str)` Returns an array with the elements of the string `$str` split at the delimiter `$delim`

`implode($glue, $arr)` Returns a string with the elements of the array `$arr` separated by `$glue`

Iterating Over Elements

- An array can be iterated with the **`foreach`** construct.

```
foreach ($arr as $value) {
    echo("$value");
}
```

```
foreach ($arr as $key => $value) {
    echo "Key: $key, Value: $value; ";
}
```

Iterating Over Elements (Cont'd)

- Also the **`while`** construct can be used.

```
while (list(, $value) = each($arr)) {
    echo("$value");
}
```

```
while (list($key, $value) = each($arr)) {
    echo "Key: $key, Value: $value;";
}
```

Iterating Over Elements (Cont'd)

- Also the **`while`** construct can be used.

```
while (list(, $value) = each($arr)) {
    echo("$value");
}
```

```
while (list($key, $value) = each($arr)) {
    echo "Key: $key, Value: $value;";
}
```

- **`each`** returns the current key/value pair and advances the cursor.

Iterating Over Elements (Cont'd)

- Also the **while** construct can be used.

```
while (list(, $value) = each($arr)) {
    echo("$value");
}
```

```
while (list($key, $value) = each($arr)) {
    echo "Key: $key, Value: $value;";
}
```

- **each** returns the current key/value pair and advances the cursor.
- **list** assigns multiple values from an array.

```
$arr = array('a', 'b', 'c');
list($elem1, $elem2, $elem3) = $arr;
```

Iterating Over Elements (Cont'd)

- Also the **while** construct can be used.

```
while (list(, $value) = each($arr)) {
    echo("$value");
}
```

```
while (list($key, $value) = each($arr)) {
    echo "Key: $key, Value: $value;";
}
```

- **each** returns the current key/value pair and advances the cursor.
- **list** assigns multiple values from an array.

```
$arr = array('a', 'b', 'c');
list($elem1, $elem2, $elem3) = $arr;
```

- Other useful functions are **reset**, **next**, **prev**, **current**

Section

- Introduction to PHP
- Types, Operators and Expressions
- Arrays
- Functions
- The Object Model
- Namespaces
- Exception Handling
- Documentation With PHPDoc

To Define a Function

- Functions are defined with the **`function`** keyword.

```
function sum($op1, $op2) {
    return $op1 + $op2;
}
```

To Define a Function

- Functions are defined with the **`function`** keyword.

```
function sum($op1, $op2) {
    return $op1 + $op2;
}
```

- Any valid PHP code may appear inside a function, even other functions and class definitions.

To Define a Function

- Functions are defined with the **`function`** keyword.

```
function sum($op1, $op2) {
    return $op1 + $op2;
}
```

- Any valid PHP code may appear inside a function, even other functions and class definitions.
- All functions and classes have global scope, they can be called outside a function even if they were defined inside.

To Define a Function

- Functions are defined with the **`function`** keyword.

```
function sum($op1, $op2) {
    return $op1 + $op2;
}
```

- Any valid PHP code may appear inside a function, even other functions and class definitions.
- All functions and classes have global scope, they can be called outside a function even if they were defined inside.
- Functions need not be defined before they are referenced.

Arguments

- Arguments are by default passed by reference.

Arguments

- Arguments are by default passed by reference.
- To pass by value, prepend an ampersand, **&**, to the argument.

```
function add_a_dot(&$string) {
    $string .= '.';
}

$str = 'My name is Olle';
add_a_dot($str);
echo $str;    // prints 'My name is Olle.'
```

Arguments

- Arguments are by default passed by reference.
- To pass by value, prepend an ampersand, `&`, to the argument.

```
function add_a_dot(&$string) {
    $string .= '.';
}

$str = 'My name is Olle';
add_a_dot($str);
echo $str;    // prints 'My name is Olle.'
```

- There can be default argument values.

```
function add_two($op1, $op2=2) {
    return $op1 + $op2;
}
echo add_two(3); // prints '5'
```

Variable-Length Argument List

- A variable-length argument list is implemented with the functions **`func_num_args()`**, **`func_get_arg()`** and **`func_get_args()`**.

```
function sum() {
    $acc = 0;
    foreach (func_get_args() as $n) {
        $acc += $n;
    }
    return $acc;
}

echo sum(1, 2, 3, 4); // prints '10'
```

Variable Function

A variable function is a function whose name is the value of a variable. Variables with appended parentheses are treated as variable functions.

```
function foo() {
    echo "In foo";
}

function bar() {
    echo "In bar";
}

$func = 'foo';
$func();         // prints 'In foo'

$func = 'bar';
$func();  // prints 'In bar'
```

Anonymous Functions and Closures

- Anonymous functions and closures behaves very much as in JavaScript.

```
function outer($param) {
    return function() use ($param) {
        echo "Inner got '$param'";
    };
}

$func = outer('Hi!');
echo $func(); //prints Inner got 'Hi!'
```

Anonymous Functions and Closures

- Anonymous functions and closures behaves very much as in JavaScript.

```
function outer($param) {
    return function() use ($param) {
        echo "Inner got '$param'";
    };
}

$func = outer('Hi!');
echo $func(); //prints Inner got 'Hi!'
```

- As can be seen above, closures are defined with the construct **`use`**.

Internal Functions

- There are many internal (built-in) functions, and also many PHP extensions with yet more functions.

Internal Functions

- There are many internal (built-in) functions, and also many PHP extensions with yet more functions.
- Reference manual for internal functions can be found at

  `http://php.net/manual/en/funcref.php`

  `http://www.w3schools.com/php/default.asp`

Type Hinting

- Type hinting is a way to introduce type safety in the otherwise type unsafe PHP language.

Type Hinting

- Type hinting is a way to introduce type safety in the otherwise type unsafe PHP language.
- Forces parameters to be of the specified class or interface, or to be an array or a function.

Type Hinting

- Type hinting is a way to introduce type safety in the otherwise type unsafe PHP language.
- Forces parameters to be of the specified class or interface, or to be an array or a function.
- The following code forces the **`param`** parameter to be an instance of the class **`MyClass`**.

```
function test(MyClass $param) {
}
```

Type Hinting

- Type hinting is a way to introduce type safety in the otherwise type unsafe PHP language.
- Forces parameters to be of the specified class or interface, or to be an array or a function.
- The following code forces the **`param`** parameter to be an instance of the class **`MyClass`**.

```
function test(MyClass $param) {
}
```

- Type hinting can not be used for primitive types such as **`integer`** or **`string`**.

Section

- Introduction to PHP
- Types, Operators and Expressions
- Arrays
- Functions
- The Object Model
- Namespaces
- Exception Handling
- Documentation With PHPDoc

The Object Model

- The object model is class based as in Java.

The Object Model

- The object model is class based as in Java.
- Class and interface definitions, inheritance, implementation and instantiation is similar to Java.

The Object Model

- The object model is class based as in Java.
- Class and interface definitions, inheritance, implementation and instantiation is similar to Java.
- ```
  class SimpleClass {
      private $var = 'a default value';

      public function displayVar() {
          echo $this->var;
      }
  }

  $instance = new SimpleClass();
  echo $instance->displayVar();
  ```
  ```

The Object Model

- The object model is class based as in Java.
- Class and interface definitions, inheritance, implementation and instantiation is similar to Java.
- ```
  class SimpleClass {
      private $var = 'a default value';

      public function displayVar() {
          echo $this->var;
      }
  }

  $instance = new SimpleClass();
  echo $instance->displayVar();
  ```
- Note 1: The syntax for method call is `->`
  ```

The Object Model

- The object model is class based as in Java.
- Class and interface definitions, inheritance, implementation and instantiation is similar to Java.
- ```
  class SimpleClass {
      private $var = 'a default value';

      public function displayVar() {
          echo $this->var;
      }
  }

  $instance = new SimpleClass();
  echo $instance->displayVar();
  ```
- Note 1: The syntax for method call is ->
- Note 2: It is not possible to specify a visibility for the class itself, all classes are public.
  ```

Properties, Methods and Visibility

- Properties and methods are as in Java.
  See previous slide for an example.

Properties, Methods and Visibility

- Properties and methods are as in Java. See previous slide for an example.
- The visibilities are **`public`**, **`protected`** and **`private`**. The meanings are the same as in Java.

Properties, Methods and Visibility

- Properties and methods are as in Java. See previous slide for an example.
- The visibilities are **`public`**, **`protected`** and **`private`**. The meanings are the same as in Java.
- The default visibility is **`public`**.

Properties, Methods and Visibility

- Properties and methods are as in Java. See previous slide for an example.
- The visibilities are **`public`**, **`protected`** and **`private`**. The meanings are the same as in Java.
- The default visibility is **`public`**.
- There is no package private visibility since there are no packages.

Constructors

- Constructors work the same ways as in Java, but they are always called **`__construct`**.

Constructors

- Constructors work the same ways as in Java, but they are always called `__construct`.
- ```
  class SimpleClass {
      private $var;

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

Constructors

- Constructors work the same ways as in Java, but they are always called `__construct`.
- ```
  class SimpleClass {
      private $var;

      public function __construct($var) {
          $this->var = $var;
      }
  }
  ```
- Note 1: Also the **`this`** variable is prefixed with **`$`**
  ```

Constructors

- Constructors work the same ways as in Java, but they are always called `__construct`.
- ```
  class SimpleClass {
      private $var;

      public function __construct($var) {
          $this->var = $var;
      }
  }
  ```
- Note 1: Also the `this` variable is prefixed with `$`
- Note 2: There is no overloading, there can be only one constructor per class.
  ```

Destructors

- Unlike Java, the destructor is called when the last reference to the object is removed.

Destructors

- Unlike Java, the destructor is called when the last reference to the object is removed.
- The destructor is called **`__destruct`**

Destructors

- Unlike Java, the destructor is called when the last reference to the object is removed.
- The destructor is called **`__destruct`**
- ```
  class SimpleClass {
      public function __destruct() {
          echo 'running destructor';
      }
  }

  $instance = new SimpleClass();
  ```
  ```

Destructors

- Unlike Java, the destructor is called when the last reference to the object is removed.
- The destructor is called **`__destruct`**
- ```
  class SimpleClass {
      public function __destruct() {
          echo 'running destructor';
      }
  }

  $instance = new SimpleClass();
  ```
- The code above prints **`running destructor`** since the last reference to the object is removed when program ends.
  ```

Destructors

- Unlike Java, the destructor is called when the last reference to the object is removed.
- The destructor is called **`__destruct`**
- ```
  class SimpleClass {
      public function __destruct() {
          echo 'running destructor';
      }
  }

  $instance = new SimpleClass();
  ```
- The code above prints **`running destructor`** since the last reference to the object is removed when program ends.
- Note that the destructor can not take any parameters.
  ```

`self` and `static`

`self` is resolved to the class where it is written, **`static`** is resolved to the called class.

```
class SuperClass {
    public static function whoAreYouSelf() {
        self::me();
    }

    public static function whoAreYouStatic() {
        static::me();
    }

    protected static function me() {
        echo "I am SuperClass";
    }
}

class SubClass extends SuperClass {
    protected static function me() {
        echo "I am SubClass";
    }
}

SubClass::whoAreYouSelf();   //prints 'I am SuperClass'
SubClass::whoAreYouStatic(); //prints 'I am SubClass'
```

The Scope Resolution Operator and Late Static Binding

- The double colon used on the previous slide is called the scope resolution operator.

The Scope Resolution Operator and Late Static Binding

- The double colon used on the previous slide is called the scope resolution operator.
- Used to specify which class to use.

The Scope Resolution Operator and Late Static Binding

- The double colon used on the previous slide is called the scope resolution operator.
- Used to specify which class to use.
- Using **`static`**, as illustrated on the previous slide, is late static binding, which means that the scope is the called class.

Object Iteration

- It is possible to iterate over fields in an object, as if the object was an array.

Object Iteration

- It is possible to iterate over fields in an object, as if the object was an array.
- Only visible fields, as specified by the visibility, will occur in the iteration.

```
class Person {
    public $name;
    public $phone;
    public $address;

    public function __construct($name, $phone, $address) {
        $this->address = $address;
        $this->phone = $phone;
        $this->name = $name;
    }
}

$stina = new Person("Stina", "1234567", "at home");
foreach ($stina as $key => $value) {
    echo "$key: $value";
}
```

Comparing Objects

- The comparison operator, **==**, considers two object instances equal if they have the same attributes and values, and are instances of the same class.

Comparing Objects

- The comparison operator, ==, considers two object instances equal if they have the same attributes and values, and are instances of the same class.
- The identity operator, ===, considers instances to be equal only if they refer to the same instance and same class.

Object Serialization

- The **`serialize`** function returns a string containing a representation of any PHP value.

Object Serialization

- The **`serialize`** function returns a string containing a representation of any PHP value.
- The **`unserialize`** function recreates the original values.

Object Serialization

- The `serialize` function returns a string containing a representation of any PHP value.
- The `unserialize` function recreates the original values.
- Serializing an object will save all variable values plus the class name of that object.

Object Serialization

- The `serialize` function returns a string containing a representation of any PHP value.
- The `unserialize` function recreates the original values.
- Serializing an object will save all variable values plus the class name of that object.
- To unserialize an object, the class definition of that object needs to be present.

Object Serialization, Example

```php
class Person {
    public $name;
    public $phone;
    public $address;

    public function __construct($name, $phone, $address) {
        $this->address = $address;
        $this->phone = $phone;
        $this->name = $name;
    }
}

$stina = new Person("Stina", "1234567", "at home");
$serialized = serialize($stina);
// prints O:6:"Person":3:{s:4:"name";s:5:"Stina";
//        s:5:"phone";s:7:"1234567";
//        s:7:"address";s:7:"at home";}
echo $serialized;

$someone = unserialize($serialized);
foreach ($someone as $key => $value) {
    echo "$key: $value"; //Same output as before serializing
}
```

Autoloading Classes

- Normally, each class is placed in a file with the same name as the class, plus the extension `.php`. This means we are forced to write one `require_once` statement for each used class.

Autoloading Classes

- Normally, each class is placed in a file with the same name as the class, plus the extension `.php`. This means we are forced to write one `require_once` statement for each used class.
- To avoid these long `require_once` listings, it is possible to register an autoload function, that is called whenever a previously unloaded class is used.

Autoloading Classes

- Normally, each class is placed in a file with the same name as the class, plus the extension `.php`. This means we are forced to write one `require_once` statement for each used class.
- To avoid these long `require_once` listings, it is possible to register an autoload function, that is called whenever a previously unloaded class is used.

```
spl_autoload_register(function ($class) {
    include 'classes/' .
            \str_replace('\\', '/', $class) .
            '.php';
});
```

Section

- Introduction to PHP
- Types, Operators and Expressions
- Arrays
- Functions
- The Object Model
- Namespaces
- Exception Handling
- Documentation With PHPDoc

Namespaces

- Namespaces are used to structure the program, as packages are used in Java.

Namespaces

- Namespaces are used to structure the program, as packages are used in Java.
- A namespace does not affect visibility, there is no package private visibility.

Namespaces

- Namespaces are used to structure the program, as packages are used in Java.
- A namespace does not affect visibility, there is no package private visibility.
- Namespaces define name spaces, the same symbol (e.g., class) can exist in different namespaces.

Namespaces

- Namespaces are used to structure the program, as packages are used in Java.
- A namespace does not affect visibility, there is no package private visibility.
- Namespaces define name spaces, the same symbol (e.g., class) can exist in different namespaces.
- Namespaces also structure the program and thereby improve cohesion.
  - If, for example, the MVC architecture is used, there should be the namespaces **`Model`**, **`View`** and **`Controller`**.

Declaring Namespaces

Namespaces are declared with the **`namespace`** keyword, placed first in a file.

```
namespace Model;
```

```
namespace \MyProject\Model\Payment;
```

Importing Namespaces

- Namespaces are imported with the **`use`** keyword and aliased with **`alias`**.

Importing Namespaces

- Namespaces are imported with the **`use`** keyword and aliased with **`alias`**.
- The following examples assume there is a namespace **`\MyProject\Model\Payment`**, which contains the class **`SomeClass`**.

Importing Namespaces

- Namespaces are imported with the **`use`** keyword and aliased with **`alias`**.
- The following examples assume there is a namespace **`\MyProject\Model\Payment`**, which contains the class **`SomeClass`**.

```
use \MyProject\Model\Payment as Pay;
new Pay\SomeClass();
//Instantiates \MyProject\Model\Payment\SomeClass
```

Importing Namespaces

- Namespaces are imported with the **use** keyword and aliased with **alias**.
- The following examples assume there is a namespace **\MyProject\Model\Payment**, which contains the class **SomeClass**.

```
use \MyProject\Model\Payment as Pay;
new Pay\SomeClass();
//Instantiates \MyProject\Model\Payment\SomeClass
```

```
use \MyProject\Model\Payment;
new Payment\SomeClass();
//Instantiates \MyProject\Model\Payment\SomeClass
```

Importing Namespaces

- Namespaces are imported with the **use** keyword and aliased with **alias**.
- The following examples assume there is a namespace **\MyProject\Model\Payment**, which contains the class **SomeClass**.

```
use \MyProject\Model\Payment as Pay;
new Pay\SomeClass();
//Instantiates \MyProject\Model\Payment\SomeClass
```

```
use \MyProject\Model\Payment;
new Payment\SomeClass();
//Instantiates \MyProject\Model\Payment\SomeClass
```

```
use \MyProject\Model\Payment;
new SomeClass(); //NOT ALLOWED!!
```

Namespaces are Hierarchical

- If the namespace **`A\B\C`** is imported as **`C`**, a call to **`C\D\E`** is translated to **`A\B\C\D\E`**.

Namespaces are Hierarchical

- If the namespace **A\B\C** is imported as **C**, a call to **C\D\E** is translated to **A\B\C\D\E**.
- A call to **C\D\E** within namespace **A\B** is translated to **A\B\C\D\E**.

Namespaces are Hierarchical

- If the namespace **A\B\C** is imported as **C**, a call to **C\D\E** is translated to **A\B\C\D\E**.
- A call to **C\D\E** within namespace **A\B** is translated to **A\B\C\D\E**.
- A call to **\C\D\E** within namespace **A\B** is translated to **C\D\E**.

The Global Namespace

- In a file without any namespace definition, all classes and functions are placed in the global namespace.

The Global Namespace

- In a file without any namespace definition, all classes and functions are placed in the global namespace.
- Prefixing a name with \ will specify that the name is required from the global namespace.

The Global Namespace

- In a file without any namespace definition, all classes and functions are placed in the global namespace.
- Prefixing a name with \ will specify that the name is required from the global namespace.
- It is good practice to prefix all functions in the global space with \, even though the interpreter always looks for functions in the global space before failing.

Section

- Introduction to PHP
- Types, Operators and Expressions
- Arrays
- Functions
- The Object Model
- Namespaces
- Exception Handling
- Documentation With PHPDoc

Exceptions

- PHP exception handling works exactly like runtime exceptions in Java. There are no checked exceptions i PHP.

Exceptions

- PHP exception handling works exactly like runtime exceptions in Java. There are no checked exceptions i PHP.
- The constructs **`throw`**, **`try`**, **`catch`** and **`finally`** have the same meaning as in Java.

Exceptions

- PHP exception handling works exactly like runtime exceptions in Java. There are no checked exceptions i PHP.
- The constructs **`throw`**, **`try`**, **`catch`** and **`finally`** have the same meaning as in Java.
- There is no **`throws`** construct since there are no checked exceptions.

Exceptions

- PHP exception handling works exactly like runtime exceptions in Java. There are no checked exceptions i PHP.
- The constructs **`throw`**, **`try`**, **`catch`** and **`finally`** have the same meaning as in Java.
- There is no **`throws`** construct since there are no checked exceptions.
- Custom exception classes shall extend the class **`Exception`**, which is in the global namespace.

Section

- Introduction to PHP
- Types, Operators and Expressions
- Arrays
- Functions
- The Object Model
- Namespaces
- Exception Handling
- Documentation With PHPDoc

Writing PHPDoc

- For simple use cases, PHPDoc is very much like Javadoc.

```
/**
 * Creates a cache with the specified layout.
 *
 * @param \Csim\Model\CacheLayout $layout The layout
 *          of the cache that shall be created.
 * @return \Csim\Model\SimulationState The state of
 *          the newly created, empty, cahce.
 */
public function
        defineCache(\Csim\Model\CacheLayout $layout) {
    $this->cache = new \Csim\Model\Cache($layout);
    return $this->cache->getState();
}
```

Generating Documentation

- To generate the HTML files with documentation, it is necessary to install a third-party tool.

Generating Documentation

- To generate the HTML files with documentation, it is necessary to install a third-party tool.
- Use for example ApiGen, **`http://apigen.org/`**, which can be integrated with NetBeans.