

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

The PHP Language

Internet Applications, ID1354

Contents

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

Introduction

Types, Operators
and Expressions

Arrays

Functions

Objects

Namespaces

Exception Handling

PHPDoc

Section

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

Introduction

Types, Operators
and Expressions

Arrays

Functions

Objects

Namespaces

Exception Handling

PHPDoc

The PHP Language

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



Introduction

Types, Operators
and Expressions

Arrays

Functions

Objects

Namespaces

Exception Handling

PHPDoc

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.



Introduction

Types, Operators
and Expressions

Arrays

Functions

Objects

Namespaces

Exception Handling

PHPDoc

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**.



Introduction

Types, Operators
and Expressions

Arrays

Functions

Objects

Namespaces

Exception Handling

PHPDoc

Overview of PHP

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

Introduction

Types, Operators
and Expressions

Arrays

Functions

Objects

Namespaces

Exception Handling

PHPDoc

Overview of PHP

- ▶ By far the **most used server-side** programming language.
- ▶ **Purely interpreted**, there is no compiler.

Introduction

Types, Operators
and Expressions

Arrays

Functions

Objects

Namespaces

Exception Handling

PHPDoc

Overview of PHP

- ▶ By far the **most used server-side** programming language.
- ▶ **Purely interpreted**, there is no compiler.
- ▶ **Object-oriented with class-based inheritance**, like Java, but using objects is optional.

[Introduction](#)[Types, Operators and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

Overview of PHP

- ▶ By far the **most used server-side** programming language.
- ▶ **Purely interpreted**, there is no compiler.
- ▶ **Object-oriented with class-based inheritance**, like Java, but using objects is optional.
- ▶ PHP files can **contain HTML and PHP**.

Introduction

Types, Operators
and Expressions

Arrays

Functions

Objects

Namespaces

Exception Handling

PHPDoc

Overview of PHP

- ▶ By far the **most used server-side** programming language.
- ▶ **Purely interpreted**, there is no compiler.
- ▶ **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**

Introduction

Types, Operators
and Expressions

Arrays

Functions

Objects

Namespaces

Exception Handling

PHPDoc

Overview of PHP

- ▶ By far the **most used server-side** programming language.
- ▶ **Purely interpreted**, there is no compiler.
- ▶ **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.

Introduction

Types, Operators
and Expressions

Arrays

Functions

Objects

Namespaces

Exception Handling

PHPDoc

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**.

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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**.

Introduction

Types, Operators
and Expressions

Arrays

Functions

Objects

Namespaces

Exception Handling

PHPDoc

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).

[Introduction](#)[Types, Operators and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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**ppache, **M**ySQL and **P**HP. These together form a **complete web server**.

[Introduction](#)[Types, Operators and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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**ppache, **M**ySQL and **P**HP. These together form a **complete web server**.
 - ▶ **UniformServer** , <http://www.uniformserver.com/>, is a WAMP pack that is **easy to install**.

[Introduction](#)[Types, Operators and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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.

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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.

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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>
```

[Introduction](#)[Types, Operators and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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`

[Introduction](#)[Types, Operators and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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.

[Introduction](#)[Types, Operators and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

Accessing PHP Files (Cont'd)

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

Introduction

Types, Operators
and Expressions

Arrays

Functions

Objects

Namespaces

Exception Handling

PHPDoc

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;`.

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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**.

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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.

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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.

[Introduction](#)[Types, Operators and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

Naming Conventions

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

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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**

[Introduction](#)[Types, Operators and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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**

[Introduction](#)[Types, Operators and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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**

[Introduction](#)[Types, Operators and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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** names are written either in camel case, **myFirstVar**, or with underscores, **my_first_var**.

[Introduction](#)[Types, Operators and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

Naming Conventions (Cont'd)

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

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

Naming Conventions (Cont'd)

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

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

Introduction

Types, Operators
and Expressions

Arrays

Functions

Objects

Namespaces

Exception Handling

PHPDoc

Question 1

Section

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

Introduction

**Types, Operators
and Expressions**

Arrays

Functions

Objects

Namespaces

Exception Handling

PHPDoc

Comments

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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.

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

Identifiers

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

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

Variables

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

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

Variables

- ▶ Variables are represented by a **dollar sign**, \$, followed by the **name** of the variable.
- ▶ There are no variable declarations, PHP is **dynamically typed**.

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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**, and changes type when needed.

```
$year = "in the eighties"; //year is a string.  
$year = 84; //year is a number.
```

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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**, and changes type when needed.

```
$year = "in the eighties"; //year is a string.  
$year = 84; //year is a number.
```

- ▶ A variable that has never been assigned a value is **unbound** and has the value **NULL**

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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**, and changes type when needed.

```
$year = "in the eighties"; //year is a string.  
$year = 84; //year is a number.
```

- ▶ A variable that has never been assigned a value is **unbound** and has the value **NULL**
- ▶ The **unset** function sets a variable to **NULL**

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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**, and changes type when needed.

```
$year = "in the eighties"; //year is a string.  
$year = 84; //year is a number.
```

- ▶ 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**

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

Predefined Variables

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

[Introduction](#)[Types, Operators and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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.

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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.

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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**.

[Introduction](#)[Types, Operators and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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**.

[Introduction](#)[Types, Operators and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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

[Introduction](#)[Types, Operators and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

Constants

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

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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;
```

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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 \$

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

Magic Constants

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

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

Magic Constants

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

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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.

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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.

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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.

[Introduction](#)[Types, Operators and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

Types

- ▶ There are **eight different types**.

Introduction

**Types, Operators
and Expressions**

Arrays

Functions

Objects

Namespaces

Exception Handling

PHPDoc

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**.

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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**.

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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).

[Introduction](#)[Types, Operators and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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.

[Introduction](#)[Types, Operators and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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.

[Introduction](#)[Types, Operators and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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.

[Introduction](#)[Types, Operators and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

The String Type

- ▶ A string consists of **one-byte characters**.

Introduction

Types, Operators
and Expressions

Arrays

Functions

Objects

Namespaces

Exception Handling

PHPDoc

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
```

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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
```

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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.

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

String Concatenation

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

- ▶ The string concatenation operator is a dot,
•

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


Arithmetic Operators

- ▶ The usual $+$, $-$, $*$, $/$ and $\%$.

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

Arithmetic Operators

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

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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.

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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.

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

Arithmetic and String Functions

- ▶ Some available [arithmetic functions](#) are **floor**, **ceil**, **round**, **abs**, **min**, **max**, **rand**.

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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**.

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

Cast (Explicit Conversion)

- ▶ Three ways to specify an explicit conversion.

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

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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 three 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);
```

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

Output

- ▶ Output from the PHP program is **included in the out stream** from server to browser.
- ▶ There are three 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**.

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

Debug Output to Console

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

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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>";  
}
```

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

Error Messages From Interpreter

- ▶ The [PHP interpreter's output](#), for example exception reports, goes to the web server's log.

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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. You might be able to find it by executing a php file with the statement **phpinfo()**, and looking for **APACHE_LOG_DIR** in the output.
 - ▶ On my Ubuntu/Apache platform, the log is in **/var/log/apache2/error.log**.

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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. You might be able to find it by executing a php file with the statement `phpinfo()`, and looking for **APACHE_LOG_DIR** in the output.
 - ▶ 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.

[Introduction](#)[Types, Operators and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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. You might be able to find it by executing a php file with the statement **`phpinfo()`**, and looking for **`APACHE_LOG_DIR`** in the output.
 - ▶ 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.

[Introduction](#)[Types, Operators and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

Comparison Operators

- ▶ The usual six comparison operators: `==` , `!=` , `<` , `>` , `<=` , `>=`

[Introduction](#)[Types, Operators and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

Comparison Operators

- ▶ The usual six comparison operators: `==` , `!=` , `<` , `>` , `<=` , `>=`
- ▶ Operands are converted if necessary

[Introduction](#)[Types, Operators and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

Comparison Operators

- ▶ The usual six comparison operators: `==` , `!=` , `<` , `>` , `<=` , `>=`
- ▶ Operands are converted if necessary
 - ▶ If one operand is a string and one is a number, the string is converted to a number.

[Introduction](#)[Types, Operators and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

Comparison Operators

- ▶ The usual six comparison operators: `==` , `!=` , `<` , `>` , `<=` , `>=`
- ▶ Operands are converted if necessary
 - ▶ If one operand is a string and one is a number, the string is converted to a number.
 - ▶ If one operand is a boolean or **null**, both operands are converted to booleans.

[Introduction](#)[Types, Operators and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

Comparison Operators

- ▶ The **usual six comparison operators**: `==` , `!=` , `<` , `>` , `<=` , `>=`
- ▶ Operands are converted if necessary
 - ▶ If one operand is a string and one is a number, the string is converted to a number.
 - ▶ If one operand is a boolean or `null`, both operands are converted to booleans.
- ▶ The **unusual two comparison operators**, `===` and `!==`, behave as `==` and `!=`, except that **no conversions** are done. The expression can only be true if the operands have the same type.

[Introduction](#)[Types, Operators and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

Logical Operators

- ▶ The **logical operators** come 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**

[Introduction](#)[Types, Operators and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

Control Statements

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

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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.

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

Question 2

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

Question 3

Section

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

Introduction

Types, Operators
and Expressions

Arrays

Functions

Objects

Namespaces

Exception Handling

PHPDoc

The Array Type

- ▶ Not like arrays of any other language.

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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.

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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.

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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  
)
```

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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  
]
```

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

Internal Array Structure

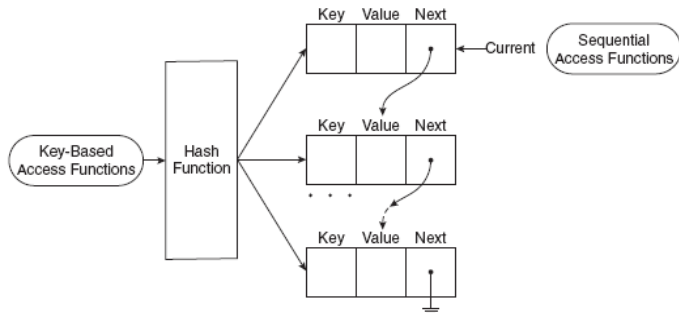


Figure from Sebesta: *Programming the World Wide Web*

Introduction

Types, Operators
and Expressions

Arrays

Functions

Objects

Namespaces

Exception Handling

PHPDoc

Keys and Values

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

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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");
```

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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**.

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

Access Elements Using Brackets

- ▶ Array elements are accessed using **brackets**.

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

Access Elements Using Brackets

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

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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
```

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

Some Array Functions

`array_keys($arr)` Extracts all keys.

Introduction

Types, Operators
and Expressions

Arrays

Functions

Objects

Namespaces

Exception Handling

PHPDoc

Some Array Functions

`array_keys($arr)` Extracts all keys.

`array_values($arr)` Extracts all values.

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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.

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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.

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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**

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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**

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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; ";  
}
```

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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;";  
}
```

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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.

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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;
```

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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**

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

Question 4

Section

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

Introduction

Types, Operators
and Expressions

Arrays

Functions

Objects

Namespaces

Exception Handling

PHPDoc

To Define a Function

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

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

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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**.

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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.

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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.

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

Arguments

- ▶ Arguments are by default passed **by value**.

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

Arguments

- ▶ Arguments are by default passed **by value**.
- ▶ To pass **by reference**, 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.'
```

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

Arguments

- ▶ Arguments are by default passed **by value**.
- ▶ To pass **by reference**, 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'
```

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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'
```

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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'
```

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

Anonymous Functions

- ▶ An anonymous function is **defined in an expression**, instead of a declaration.

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

Anonymous Functions

- ▶ An anonymous function is **defined in an expression**, instead of a declaration.
- ▶ The **reference** to the anonymous function is **stored in a variable**, which can then be used to invoke the function.

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

Anonymous Functions

- ▶ An anonymous function is **defined in an expression**, instead of a declaration.
- ▶ The **reference** to the anonymous function is **stored in a variable**, which can then be used to invoke the function.

```
▶ $myFunc = function($a, $b) {  
    return $a + $b;  
};  
$myFunc(4, 3); //Returns 7
```

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

Closures

- ▶ A closure means that a nested function can **access a variable in its outer function** after the outer is **closed**.

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

Closures

- ▶ A closure means that a nested function can **access a variable in its outer function** after the outer is **closed**.
- ▶ The variable in the outer function will have the value it had when the outer function created the nested function.

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

Closures

- ▶ A closure means that a nested function can **access a variable in its outer function** after the outer is **closed**.
- ▶ The variable in the outer function will have the value it had when the outer function created the nested function.

```
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**.

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

Internal Functions

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

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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`

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

Type Hinting

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

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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.

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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) {  
}
```

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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**.

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

Question 5

Section

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

Introduction

Types, Operators
and Expressions

Arrays

Functions

Objects

Namespaces

Exception Handling

PHPDoc

The Object Model

- ▶ The object model is **class based** as in Java.

Introduction

Types, Operators
and Expressions

Arrays

Functions

Objects

Namespaces

Exception Handling

PHPDoc

The Object Model

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

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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 = 'some value';  
  
    public function displayVar() {  
        echo $this->var;  
    }  
}  
  
$instance = new SimpleClass();  
echo $instance->displayVar();
```

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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 = 'some value';  
  
    public function displayVar() {  
        echo $this->var;  
    }  
}
```

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

- ▶ Note 1: The syntax for **method call** is ->

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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 = 'some 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**.

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

Properties, Methods and Visibility

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

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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.

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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**.

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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.

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

Constructors

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

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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;  
    }  
}
```

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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 \$**

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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.

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

Destructors

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

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

Destructors

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

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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();
```

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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.

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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**.

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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'
```

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

The Scope Resolution Operator and Late Static Binding

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

[Introduction](#)[Types, Operators and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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**.

[Introduction](#)[Types, Operators and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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**.

[Introduction](#)[Types, Operators and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

Object Iteration

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

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)[\) {](#)

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";  
}
```

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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**.

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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**.

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

Object Serialization

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

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

Object Serialization

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

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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.

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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.

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

Object Serialization, Example

```
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
}
```

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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.

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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.

[Introduction](#)[Types, Operators and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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';  
});
```

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

Question 6

Section

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

Introduction

Types, Operators
and Expressions

Arrays

Functions

Objects

Namespaces

Exception Handling

PHPDoc

Namespaces

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

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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.

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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.

[Introduction](#)[Types, Operators and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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**.

[Introduction](#)[Types, Operators and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

Declaring Namespaces

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

```
namespace \Model;
```

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

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

Importing Namespaces

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

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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**.

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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

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

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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**.

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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**.

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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**.

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

The Global Namespace

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

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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.

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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.

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

Section

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

Introduction

Types, Operators
and Expressions

Arrays

Functions

Objects

Namespaces

Exception Handling

PHPDoc

Exceptions

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

[Introduction](#)[Types, Operators and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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.

[Introduction](#)[Types, Operators and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

Exceptions

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

[Introduction](#)[Types, Operators and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

Exceptions

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

[Introduction](#)[Types, Operators and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

Section

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

Introduction

Types, Operators
and Expressions

Arrays

Functions

Objects

Namespaces

Exception Handling

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();
}
```

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

Generating Documentation

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

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)

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.

[Introduction](#)[Types, Operators
and Expressions](#)[Arrays](#)[Functions](#)[Objects](#)[Namespaces](#)[Exception Handling](#)[PHPDoc](#)