OpenKnowTech & CC ICT-SUD

An introduction to the language (PHP 5)... in fact just the theory needed to fully understand the presented code examples.

PHP vs others

- Cons:
  - inconsistencies in naming and syntax
  - not a pure object oriented language
- Pros:
  - much popular, supported by many ISPs
  - web oriented and good for database access
  - backward compatibility assured
  - script language but quite fast, dynamic
  - completely open source, easily extendable in C

PHP Manual

- It is the official, authoritative and most complete documentation and it's available in both online and downloadable versions
- How to get help on a particular control structure, feature, function, etc: http://www.php.net/SEARCHKEY
- More about the PHP site shortcuts at:

  http://www.php.net/urlhowto.php

PEAR

- PEAR (PHP Extension and Application Repository) is a framework and distribution system for reusable PHP components
- Provides a structured library of code usually in an OO style and a foundation for developing new modules
- Promotes a standard coding style
- Takes care of distributing and managing code packages through a package manager

Identifier names

- Anything that goes into the global namespace should be prefixed or suffixed with an uncommon 3-4 letter word. E.g.:
  - MyPx_someFunc()
  - Foo_Date
  - ALIB_CONSTANT_VAL
  - $asdf_dbh
- For more info see Userland naming guide
- As of PHP 5.3.0 namespaces are available.

String literals

- Single quoted:
  - to specify a literal single quote, escape it: \'
  - to specify a literal backslash before a single quote, or at the end of the string, double it: \\
  - no variables and escape sequences will be expanded, thus are faster
- Double quoted:
  - Escape sequences are interpreted. The most common are: \n \r \t \\ \$ \”
  - Variable names will be expanded

Cookies 

- Mechanism for storing data in the remote browser in order to track or identify users
- Sessions usually use cookies to store the unique session id on the user side, because they are safer and more efficient than URL propagation
- Storing a username in a cookie is insecure

bool **setcookie** (string *$name* [, string *$value* [, int *$expire* = *0* [, string *$path* [, string *$domain* [, bool *$secure* = *false* [, bool *$httponly* = *false*]]]]]])

Session Handling

- (Optional) You can set session parameters only before session_start():

  session_name('MYAPP');

  session_id('global');
- session_start();
- After session_start() you can get current session name/id:

  session_name(); by default returns PHPSESSID

  session_id(); e.g.returns etgk2s3ccjd77nbkca982o7dr1

Session Variables

- **write.php**

```
<?php
session_start();      # mandatory
$_SESSION['counter'] = 1;
?>
```

- **read.php** (run it more than once)

```
<?php
session_start();      # mandatory
# it displays 1 at the 1th run, 2 at the 2nd, etc...
echo $_SESSION['counter']++;
?>
```

- Check for existence: `if (isset($_SESSION['counter']))`
- Unregister: `unset($_SESSION['counter']);`

Login and redirects

```php
<?php

# Code for Logging a User.

# Filter and validate username and password
if ( successful login ) {
    # Setting Session.
    session_start();
    $_SESSION['user'] = user name;

    # Redirecting to the logged page.
    $host = $_SERVER['HTTP_HOST'];
    $uri = rtrim(dirname($_SERVER['PHP_SELF']), '/\\');
    header('Location: http://$host$uri/index.php');
    exit;
} else {
    # Wrong username or Password. Show error here.
}

?>
```

Custom pages

```php
<?php

# Code for differentiating Guest and Logged members.

session_start();

if (isset($_SESSION['user']))  {
    # Code for Logged members.

    # Identifying the user.
    $user = $_SESSION['user'];

    # Information for the user.
} else {
    # Code to show Guests.
}

?>
```

Logout

```php
<?php

session_start();

if (isset($_SESSION['user']))  {
    # Unset all of the session variables.
    $_SESSION = array();

    /* Also delete the session cookie.
       This assumes sessions use cookies. */

    $params = session_get_cookie_params();
    setcookie(session_name(), '', time() - 42000,
        $params['path'], $params['domain'],
        $params['secure'], $params['httponly']

    );

    # Finally, destroy the session.
    session_destroy();

    # Redirect as in login.
} else {
    # Error: you must first log in.
}

?>
```

Classes and Objects: con/destructors

- PHP5 has a full object model. Nothing to envy Java/C++ for. Except nested and friend classes, but we can live without them in PHP.
- Constructors:
  - Called on each newly-created object. Initialize the object as needed before use
  - If the child class defines a con/destructor, the parent con/destructor is not called implicitly; if and where you want to call it use: parent::__con/destruct(...) within the child con/destructor

Inheritance

- Syntax for creating a subclass (same as Java):

  class mountainBike extends bicycle { … }

- Classes must be defined before they are used
- As Java there is no support for multiple inheritance, you can use interfaces instead and implement multiple interfaces in a class via the implements keyword
- Any class that contains at least one abstract method must also be abstract (as in Java)

Polymorphism

```
class Base {
    function method() { echo "base\n"; }
}


class Child1 extends Base {
    function method() { echo "child1\n"; }
}


class Child2 extends Base {
    function method() { echo "child2\n"; }
}

$base = new Base();
$base->method(); # base

$child1 = new Child1();
$child1->method(); # child1

$child2 = new Child2();
$child2->method(); # child2
```

Late Binding

```
interface Int {
    function behaviour();
}

class Base {
    function method() {
        $this->behaviour();
    }
}

class Sub extends Base implements Int {
    function behaviour() {
        echo 'Late binding works!';
    }
}

$sub = new Sub();
$sub->method(); # Late binding works!
```

Fail-safe late binding

```
interface Int {
    function behaviour();
}

class Base {
    function method() {
        $this->behaviour();
    }

    function behaviour() { }
}

class Sub1 extends Base implements Int {
    function behaviour() { }
}

class Sub2 extends Base {

}

$sub2 = new Sub2();
$sub2->behaviour();
```

Overloading methods?

```
class Overloading {
    function overloaded() {
        return "overloaded called\n";
    }

    function overloaded($arg) {
        echo "overloaded($arg) called\n";
    }
}

$o = new Overloading();
echo $o->overloaded();
$o->overloaded(1);
```

Would not compile: Fatal error: Cannot redeclare Overloading::overloaded()

Trick to implement overloading

```
class Overloading {
    function __call($method, $args) {
        if ($method == 'overloaded') {
            if (count($args) == 0) {
                return $this->noArg();
            } elseif (count($args) == 1) {
                $this->oneArg($args[0]);
            } else {
                # Do nothing. You can also trigger an error here.
                return FALSE;
            }
        }
    }
    protected function noArg() { return "noArg called\n"; }
    protected function oneArg($arg) { echo "oneArg($arg) called\n"; }
}

$o = new Overloading();
echo $o->overloaded();
$o->overloaded(1);
```

Getting the name of a class

- The __CLASS__ magic constant always contains the class name that it is called in (i.e. it's in). In global context, it's an empty string.
- The string **get_class** ([object *$object*]) function returns the name of the class of the given object. When called in a method:
  - without any arguments (static and non-static methods): same as __CLASS__ (both are substituted at bytecode compile time)
  - with $this argument (only for non-static methods): the class of the object the method is called on

How to get the name of the class where a static method call was made against? Use get_called_class().

Inspecting variables

PHP does not have an internal debugging facility, but has four external debuggers.

- Dumps information about a variable:
  void **var_dump** (mixed *$expression* [, mixed *$expression* [, *$...*]])
- Outputs or returns a parsable string representation of a variable:
  mixed **var_export** (mixed *$expression* [, bool *$return* = false])
- Prints human-readable information about a variable:
  mixed **print_r** (mixed *$expression* [, bool *$return* = *false*])

Late Static Bindings

- TODO

The callback pseudo-type

- Example values:

```
'function_name', array($obj, 'methodName'),
array('ClassName', 'staticMethodName'),
array('ClassName::staticMethodName')
```

  and closures...

- Make an array of strings all uppercase:

```
$array = array_map('strtoupper', $array);
```

- Sort an array by the length of its values:

```
function cmp($a, $b){
    return strlen($b) - strlen($a);
}

usort($array, 'cmp');
```

Lambda Functions

string **create_function** (string *$args* , string *$code*)

Creates an anonymous function at runtime from the parameters passed, and returns a unique name for it (i.e. 'lambda_1') or FALSE on error. Being a string, the return value can be used as a callback.

```
usort($array,
    create_function('$a,$b',
        'return strlen($b) - strlen($a);'
    )
);
```

Closures

function [**&**] (**[**formal parameters]) [use (**$var1**, **$var2**, **&$refvar**)**]** **{ ... }**

```
usort($array, function($a, $b) {
    return strlen($b) - strlen($a);
});
```

TODO: example with “use”

Type Hinting

```
function test(MyClass $typehintexample = NULL) {}
function array_average(array $arr) { ... }
function average($val) {
        if (is_numeric($val)) return $val;
        /* a warning will be issued if we get
        here and $val isn't an array */
        return array_sum($val) / count($val);
}
```

- If the type does not match a PHP Catchable fatal error occurs (it's not an exception)

PHP Arrays

- A PHP array is a ordered map optimized for several uses:
  numerical (indexed), associative or mixed array - multidimensional array - list (vector) - hash table – dictionary – collection – stack – queue – tree – coffe machine :-)
- Keys may only be integers or strings
- Values may be any value of any type, including other arrays
- No fixed number of values, no single type!
- Can be created/modified with square bracket syntax. A rich library of array functions is available.

Kludges for constant arrays

- static $CONSTARRAY = array(...);

  ... Class::$CONSTARRAY ...
- static function CONSTARRAY() { return array(...); }

  ... Class::CONSTARRAY() ...
- define('CONSTARRAY', serialize(array(...)));

  ... unserialize(CONSTARRAY) ...

  define('CONSTARRAY', 'return ' . var_export(array(...), 1) . ';');

  ... eval(CONSTARRAY) ...

PHP Exception Handling

- Runtime exception handling in PHP resembles that of Java, but somewhat simplified
- Unhandled exceptions are always forwarded; catching and re-throwing them within a catch block is allowed (as in Java). Only needed if you want to change the exception object to forward up.
- There is no throws clause, that is there's only one exception type: unchecked
- Exception class (PHP) =Throwable class (Java)
- PHP has no finally block – this is a hole in the language

PDO: PHP Data Objects

- Provides a data-access abstraction layer not a database abstraction
- Bundled with PHP as of version 5.1 – current stable version is 5.3.1
- Don't use PDO::exec and PDO::quote to build SQL statement using string interpolation
- Use prepared statements with bound parameters: faster and secure

PDO: prepared statements

```
$dbh = new PDO('mysql:dbname=testdb;host=localhost', 'user', 'pwd');
$sth = $dbh->prepare('... ? … ? …');
        or - you cannot mix ? and :name in the same query:
$sth = $dbh->prepare('... :name … :othername …');
$sth->bindParam/Value(1, ...); $sth->bindParam(2, ...); or:
$sth->bindParam/Value(':name', ...); opt. add param type
$sth->execute();
shortcut for input only and all-strings parameters:
$sth->execute(array(..., ...)); or:
$sth->execute(array(':name' => ..., ':othername' => ...));
```

PDO: some param/value data types

bool PDOStatement::bindValue(mixed $parameter, mixed $value [, int $data_type = PDO::PARAM_STR])

bool PDOStatement::bindParam(mixed $parameter, mixed $value [, int $data_type = PDO::PARAM_STR [, int $length]])

PDO::PARAM_BOOL	boolean
PDO::PARAM_NULL	NULL
PDO::PARAM_INT	integer
PDO::PARAM_STR	string (default)

PDO::PARAM_INPUT_OUTPUT OR-bitwise with one of the above to bind and INOUT parameter (only useful with stored procedures). A length must follow.

include/require quirks

- These statemens includes and evaluates the specified file. Slower _once versions are available.
- Use require if you want a missing file to halt processing of the page; include if you want your script to continue regardless
- Path defined: absolute, relative: if an included script contains another include with a relative path, note this is only based off of the first script current working directory, and not that of the included script
- No path is given: include_path will be used

Autoloading Classes

```
function __autoload($class_name) {
    require_once $class_name . '.php';
}
```

- The autoload function must be defined in main scope in "\" namespace.
- To keep things simple, it is advisable to keep source files in the filesystem in a directory tree correspondent to that of namespaces.

PHP Namespaces

- Declaration: namespace MyWebapp; (that is \MyWebapp)
- Namespaces can nest into others:

  namespace CompanyName\ProjectName\Library\Database;
- Before, to avoid name clashes, long prefixes were used: CompanyName_ProjectName_Library_Database_ClassName
- Namespace aliasing allows to shorten long qualified names:

  use CompanyName\ProjectName\Library\Database [as Database];

  $obj = new Database\ClassName();
- Single classes can also be aliased (not constants or functions):

  use CompanyName\ProjectName\Library\Database\ClassName as C;

  $obj = new C();

Three types of names

- Unqualified:

  Database (namespace)

  ClassName (class)

- Qualified:

  Library\Database (namespace)

  Database\Classname (class)

- Fully-qualified:

  \CompanyName\ProjectName\Library\Database (namespace)

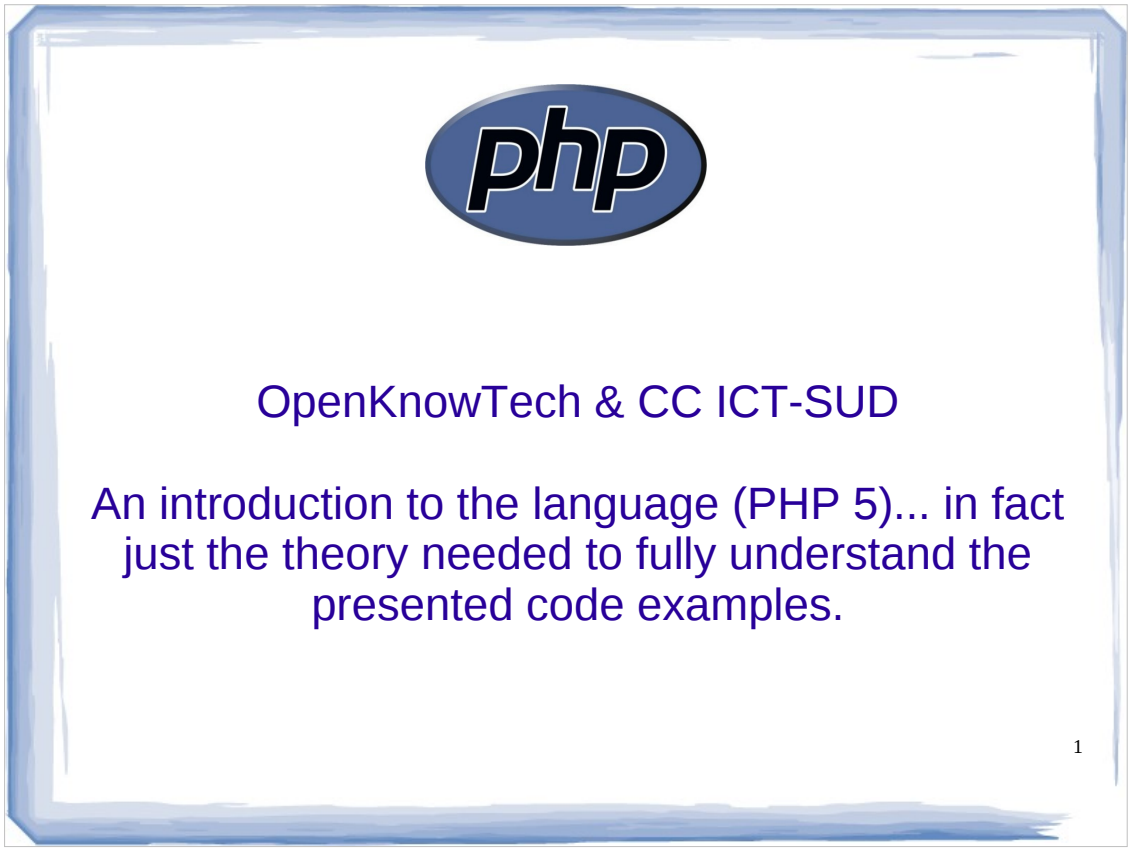
  \CompanyName\ProjectName\Library\Database\ClassName (class)

Data Filtering

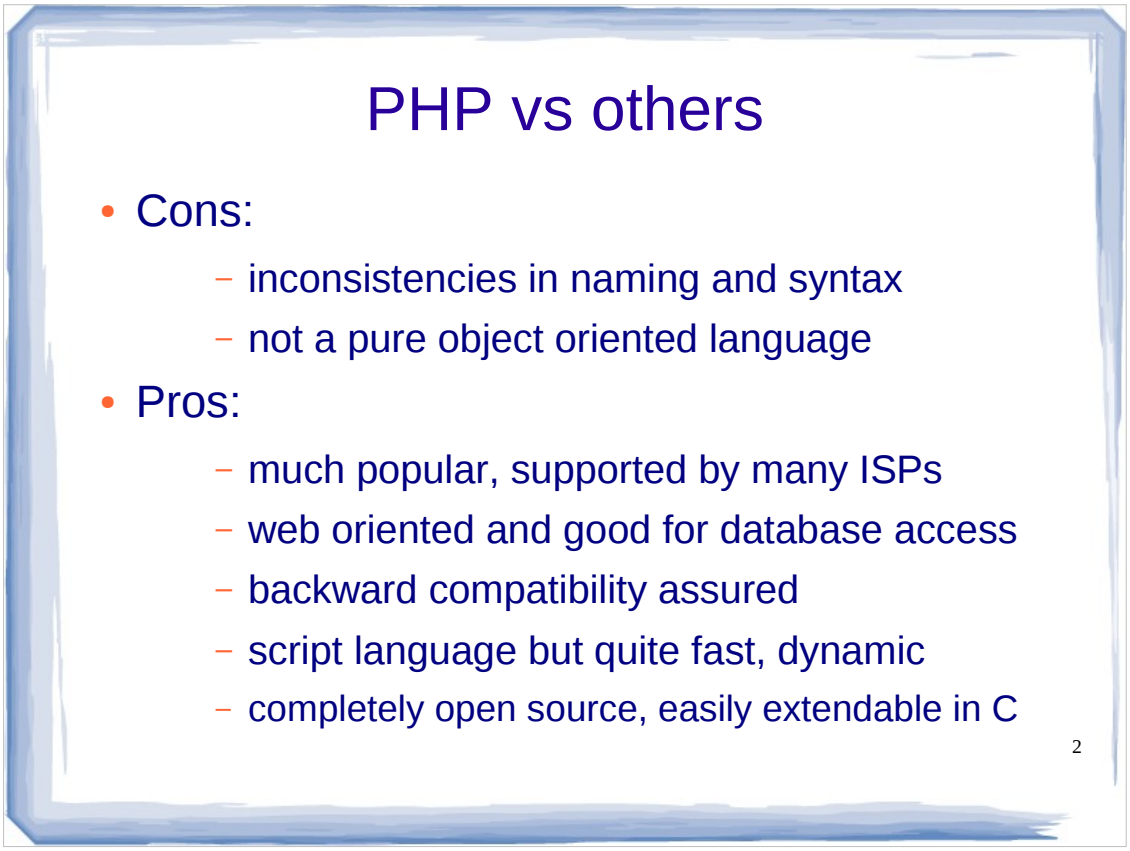
- Validation: check if the data meets certain qualifications: BOOLEAN, EMAIL, FLOAT, INT, IP, REGEXP, URL
- Sanitization: alter the data, e.g. by removing undesired characters, applying URL-encoding, adding slashes, stripping tags, HTML-escaping

mixed **filter_input** (int *$type* , string *$variable_name* [, int *$filter* = *FILTER_DEFAULT* [, mixed *$options*]])

mixed **filter_input_array** (int *$type* [, mixed *$definition*])



PHP: Hypertext Preprocessor



The cons are mainly due to the evolving nature of the language and its history: PHP3 was only procedural. PHP4 added a rudimentary object model. We have to wait for PHP5 for PHP to become a real object-oriented language, but not pure. You can still use procedural programming or mix it with OOP.

PHP syntax is not uniform and has been influenced by various languages as Perl, C, C++.

PHP is easy to learn and use than most other languages. Allows rapid development and prototyping.

PHP3 code still works with PHP5, only a few minor tweaks may be required.

Normally PHP is parsed and compiled at run time, i.e. every time a user accesses the webpage, not at the page's first view or at design time. After that it runs in a virtual machine (Zend Engine), as Java. If more speed is needed you can: use a php compiler to store script in binary format and/or a bytecode optimizer to reduce execution time and/or an opcode cache (OPC) to cache a compiled form of a PHP script in shared memory (PHP6 will have OPC built-in). Code scramblers (obfuscators) are also available, but a good license/lawer or not giving out the code and instead run a hosted service may work better :-)

Java is not entirely open source, some parts are still untouchable and Sun decide what goes into Java, not the community.

PHP Manual

- It is the official, authoritative and most complete documentation and it's available in both online and downloadable versions
- How to get help on a particular control structure, feature, function, etc: http://www.php.net/SEARCHKEY
- More about the PHP site shortcuts at:

  http://www.php.net/urlhowto.php

PEAR

- PEAR (PHP Extension and Application Repository) is a framework and distribution system for reusable PHP components
- Provides a structured library of code usually in an OO style and a foundation for developing new modules
- Promotes a standard coding style
- Takes care of distributing and managing code packages through a package manager

To keep things simple and practice the standard PHP functions before all, we won't use PEAR for our project here.

Identifier names

- Anything that goes into the global namespace should be prefixed or suffixed with an uncommon 3-4 letter word. E.g.:
  - MyPx_someFunc()
  - Foo_Date
  - ALIB_CONSTANT_VAL
  - $asdf_dbh
- For more info see Userland naming guide
- As of PHP 5.3.0 namespaces are available.

5

It doesn't matter what convention you choose for your identifier: the most important thing is to be consistent with it.

Four popular conventions for names of functions, classes, variables, constants are: underscore_between_words, alltogetherwords, CamelCase, ALL_UPPERCASE

To avoid name clashing between code you create and internal PHP or thirdy part library identifiers now and in the future, prefixes/suffixes or namespaces should be used.

We will use namespaces because prexifing/suffixing makes names longer. Namespace in PHP are defined and used in a similar fashion as paths to directory in a filesystem (they can be nested, absolute, relative, there is a current namespace concept akin to current directory).

String literals

- Single quoted:
  - to specify a literal single quote, escape it: \'
  - to specify a literal backslash before a single quote, or at the end of the string, double it: \\
  - no variables and escape sequences will be expanded, thus are faster
- Double quoted:
  - Escape sequences are interpreted. The most common are: \n \r \t \\ \$ \”
  - Variable names will be expanded

6

In scripting languages variables are preceded by a dollar sign to the purpose of allowing a simple syntax of interpolation of variables into strings. Some languages (e.g. BOO of .NET) allow interpolating entire expressions into strings. PHP only parses variables within strings (scalar variable values, array values or object properties). E.g.:

“I ate $fruits[banana] bananas in all my life.”

Outside a string, the key must be quoted otherwise will be interpreted as a constant (and if undefined, PHP will issue an error notice):

```
echo $fruits['banana'];
```

At the moment PHP has partial UTF-8 support. Many string functions regard a single character as a byte. PHP version 6 will have full UTF-8 support. For historical reasons dots were used for string concatenation and thus cannot be used for object navigation, for that PHP uses ->

Cookies

- Mechanism for storing data in the remote browser in order to track or identify users
- Sessions usually use cookies to store the unique session id on the user side, because they are safer and more efficient than URL propagation
- Storing a username in a cookie is insecure

bool **setcookie** (string *$name* [, string *$value* [, int *$expire* = *0* [, string *$path* [, string *$domain* [, bool *$secure* = *false* [, bool *$httponly* = *false*]]]]]])

7

PHP sessions also supports url id propagation. Cookies are a more efficient and safer way, since they are temporary, don't show up in your browsing history, and can't accidentally be mailed to someone with the url, no need to append the session id to every link and form submission etc.

A cookie usually stores only the session id, all the session data are stored on the server. Much safer, you can store as much data as you like and data do not have to be retrasmitted between client and server on each request.

Because PHP abstracts the storage method from the programmatic interface to session management, different storage strategies can be used (files, memory, tables in a database, etc.)

Implementing sessions through cookies by storing the user id (aka username or login) in a cookie would be insecure because everyone who knows someone else's username will be able to act as that user or even logging him/her out. This is why hard-to-guess 26-letters long identifier are used for session values in PHP by default.

Session Handling

- (Optional) You can set session parameters only before session_start():

  session_name('MYAPP');

  session_id('global');
- session_start();
- After session_start() you can get current session name/id:

  session_name(); by default returns PHPSESSID

  session_id(); e.g.returns etgk2s3ccjd77nbkca982o7dr1

A session has a name (used as a cookie name) and an id (cookie value). By default the name is PHPSESSID. Names can contain only alphanumeric characters and cannot consist of digits only; they should be short and descriptive (i.e. for users with enabled cookie warnings).

Normally session ids (SIDs) are uniquely randomly generated for private sessions (private per browser instance i.e. session cookie); e.g). Setting it to a constant allows for global sessions (global to all clients) Some session handlers place syntax restrictions on SID, e.g. for the file session handler they must match the following regex: [-a-zA-Z0-9,]+

session_start() creates a session or resumes the current one (recreates the prior saved environment) if a specific session ID has been sent with the request by the browser (through a cookie or the get/post method). Normally session are not auto-started to avoid overhead for the scripts that do not need them.

Session Variables

- **write.php**

```
<?php
session_start();     # mandatory
$_SESSION['counter'] = 1;
?>
```

- **read.php** (run it more than once)

```
<?php
session_start();     # mandatory
# it displays 1 at the 1th run, 2 at the 2nd, etc...
echo $_SESSION['counter']++;
?>
```

- Check for existence: `if (isset($_SESSION['counter']))`
- Unregister: `unset($_SESSION['counter']);`

9

To register a variable with the current session just add a new key to the $_SESSION superglobal array. These keys cannot start with a number and must start with a letter or _, as any other PHP variable name.

All registered variables are serialized after the request finishes.

Resource variables (e.g. external resource handlers as db connections, opened files) are garbage collected during execution and automatically destroyed at the of the request. Thus cannot be serialized or registered with a session. They must be re-created by each script that needs them.

Login and redirects

```php
<?php

# Code for Logging a User.

# Filter and validate username and password
if ( successful login ) {
    # Setting Session.
    session_start();
    $_SESSION['user'] = user name;

    # Redirecting to the logged page.
    $host = $_SERVER['HTTP_HOST'];
    $uri = rtrim(dirname($_SERVER['PHP_SELF']), '/\\');
    header('Location: http://$host$uri/index.php');
    exit;
} else {
    # Wrong username or Password. Show error here.
}

?>
```


Here is how to create logged sessions.

There should be no output sent to the browser before header(), not even spaces or blank lines.

HTTP/1.1 requires an absolute URL as argument to Location: including the scheme, hostname and absolute path, but some clients accept relative URLs.

Custom pages

```php
<?php
# Code for differentiating Guest and Logged members.
session_start();
if (isset($_SESSION['user']))  {
    # Code for Logged members.
    # Identifying the user.
    $user = $_SESSION['user'];
    # Information for the user.
} else {
    # Code to show Guests.
}
?>
```

Logout

```php
<?php

session_start();

if (isset($_SESSION['user']))  {
    # Unset all of the session variables.
    $_SESSION = array();

    /* Also delete the session cookie.
       This assumes sessions use cookies. */

    $params = session_get_cookie_params();
    setcookie(session_name(), '', time() - 42000,
        $params['path'], $params['domain'],
        $params['secure'], $params['httponly']

    );

    # Finally, destroy the session.
    session_destroy();

    # Redirect as in login.
} else {
    # Error: you must first log in.
}

?>
```


session_destroy() destroys all of the data associated with the current session in the session storage. It does not unset any of the global variables associated with the session, or unset the session cookie. The first will be freed when the script terminates. For the last, if a cookie is used to propagate the session id (default behaviour), you need to delete the client-side session cookie by re-sending it with the same parameters but an empty value and a past expiration date.

Do not forget to initialize the session you want to destroy prior to destroying it by using session_start('NAME') where 'NAME' is the session name if different from default.

Note no output must be written before calling setcookie otherwise it will fail.

Classes and Objects: con/destructors

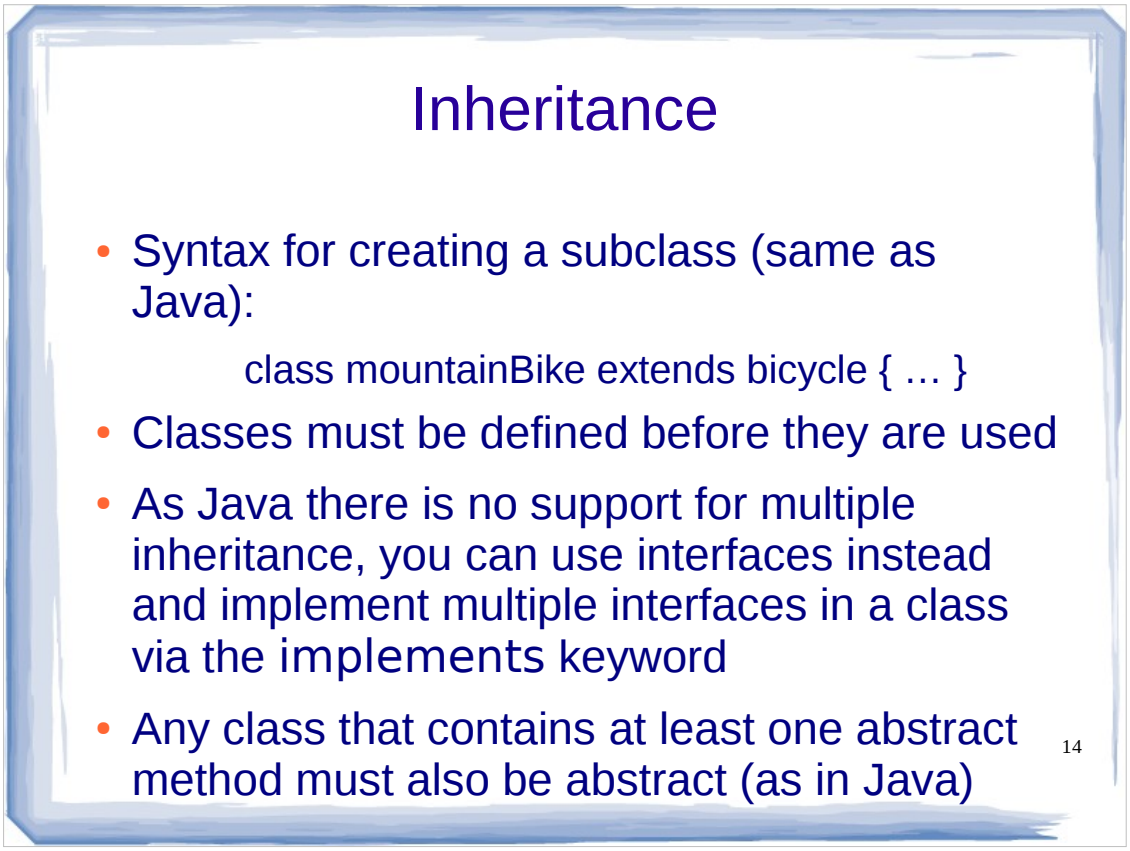
- PHP5 has a full object model. Nothing to envy Java/C++ for. Except nested and friend classes, but we can live without them in PHP.
- Constructors:
  - Called on each newly-created object. Initialize the object as needed before use
  - If the child class defines a con/destructor, the parent con/destructor is not called implicitly; if and where you want to call it use: parent::__con/destruct(...) within the child con/destructor

13

Differences with Java: if there is no explicit call to a constructor in the first line of constructor, the compiler will insert a call to the parameterless constructor of the parent, which must be defined otherwise a compilation error is produced. In PHP constructors are never called implicitly.

PHP does not let you define a class within another class as in C++/Java. In Java a non-static nested classes (“inner classes”) is associated with an instance of its enclosing class and has access to other members of the enclosing class, even if they are declared private. This only provides a method to partition an object into more pieces enhancing encapsulation. Cons: these pieces are not reusable outside the objects; inner classes are unnecessarily verbose; the more class you load the more memory you need and application startup increases.
Java needs inner classes mainly because the event-handling mechanism of the GUI uses them. PHP has callbacks, lambda functions and closures which are good substitutes for anonymous inner functions used as event handlers.

In Java too there is no 'friend' concept: you have to put your “friends” in the same package if you want a class to be able to access protected methods of another class without extending it. PHP does not provide visibility features for namespaces yet.



When you extend a class, the subclass inherits all of the public and protected methods from the parent class. Unless a class overrides those methods, they will retain their original functionality. This is useful for defining and abstracting functionality, and permits the implementation of additional functionality in similar objects without the need to reimplement all of the shared functionality.

PHP5 supports the concept of interfaces which you can use to accomplish the same thing you would with multiple inheritance, albeit in a more elegant and cleaner way. An interface imparts no behaviour, only a set of rules, while a base class could provide some default implementation.

Abstract classes act as a template for inheritance and cannot be instantiated in their own right. Difference with interfaces: an abstract class can have some non-abstract functions, while interfaces must only have method's signatures, you cannot define any implementation at the interface level.

Interfaces can only define methods and constants, no properties (this is an implementation details left to the classes that the implement that interface). A class cannot implement two interfaces that share function names, since it would cause ambiguity.

Polymorphism

```
class Base {
    function method() { echo "base\n"; }
}


class Child1 extends Base {
    function method() { echo "child1\n"; }
}


class Child2 extends Base {
    function method() { echo "child2\n"; }
}

$base = new Base();
$base->method(); # base

$child1 = new Child1();
$child1->method(); # child1

$child2 = new Child2();
$child2->method(); # child2
```

15

polymorphism translates from Greek as **many forms** (*poly - many morph - forms).*

Polymorphism in OOP means that a same method can behave differently on different instances (of different types) of an object. It allows us to organize similar objects in a logical way such that calling code does not have to worry about specific types; rather, we code to an expected interface or base class without regard to an object's concrete class. Our code is written to the lowest common denominator. In this way we increase the clarity of your code by encapsulating conditional behavior-–behavior based on an object's state-–within the object itself, rather than handling it with code that, for all intents and purposes, should not know enough about the object to make any real decisions on such matters.

Polymorphism is just overriding methods from a subclass.

Late Binding

```
interface Int {
    function behaviour();
}

class Base {
    function method() {
        $this->behaviour();
    }
}

class Sub extends Base implements Int {
    function behaviour() {
        echo 'Late binding works!';
    }
}

$sub = new Sub();
$sub->method(); # Late binding works!
```


Late (or dynamic) binding is referring to method and properties in the base class that do not exist yet, but are present in subclasses. In other words, calling unique methods in children from the parent.

Latest versions of PHP5 support late binding properly.

Fail-safe late binding

```
interface Int {
    function behaviour();
}

class Base {
    function method() {
        $this->behaviour();
    }

    function behaviour() { }
}

class Sub1 extends Base implements Int {
    function behaviour() { }
}

class Sub2 extends Base {

}

$sub2 = new Sub2();
$sub2->behaviour();
```


Fail-safe version: if you do not define a method behaviour() in child classes PHP won't throw a Fatal Error, because it will inherit the default behaviour from the base class

If we try to call behaviour from our Sub2 class, it will fail safely, and execute the default behaviour from the method in our Base class.

Overloading methods?

```
class Overloading {
   function overloaded() {
      return "overloaded called\n";
   }

   function overloaded($arg) {
      echo "overloaded($arg) called\n";
   }
}

$o = new Overloading();
echo $o->overloaded();
$o->overloaded(1);
```

Would not compile: Fatal error: Cannot redeclare Overloading::overloaded()

Overriding: same method names with same arguments and same return types associated in a class and its subclass, but PHP does not enforce the return types to be the same as in this traditional definition.

Overloading - same method name with different arguments may or may not be same return type written in the same class itself.

In polymorphism, you have methods that behaves differently depending on who is performing it, method overloading is having one method that behaves differently depending on the number of arguments passed, or type.

Trick to implement overloading

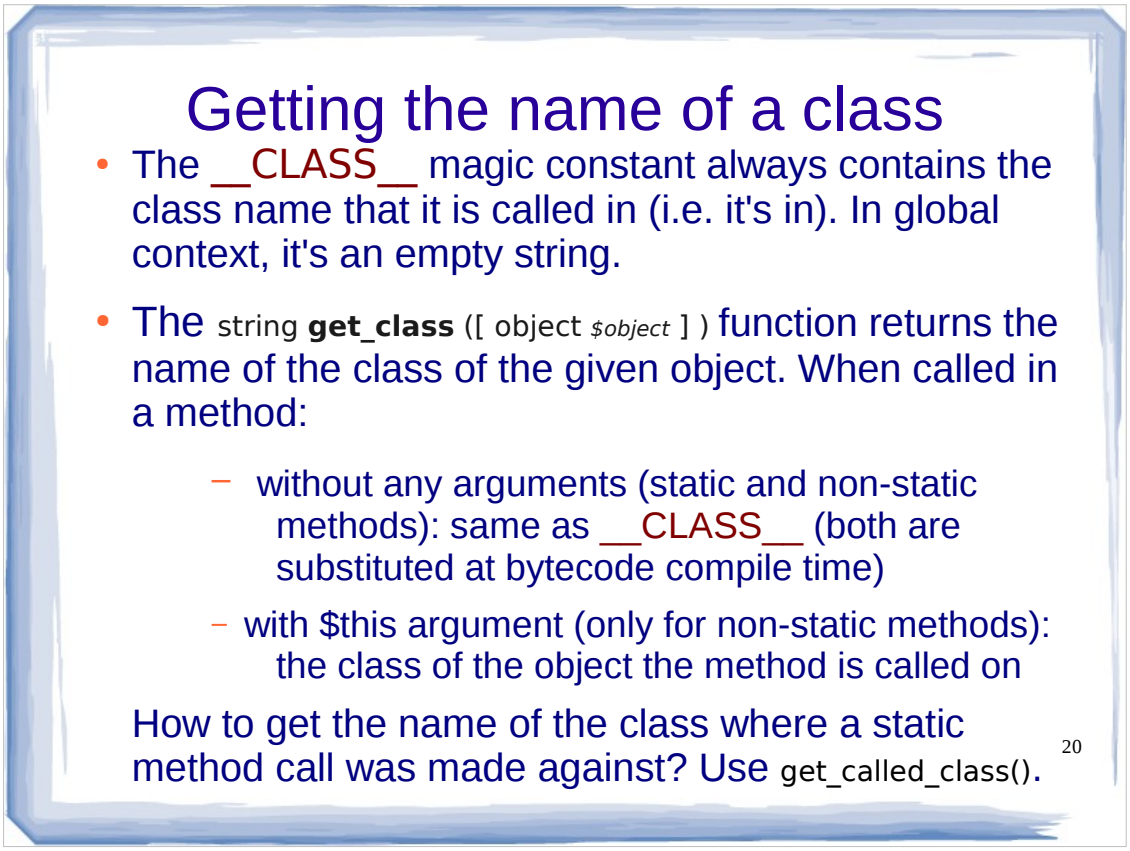
```
class Overloading {
   function __call($method, $args) {
      if ($method == 'overloaded') {
         if (count($args) == 0) {
            return $this->noArg();
         } elseif (count($args) == 1) {
            $this->oneArg($args[0]);
         } else {
            # Do nothing. You can also trigger an error here.
            return FALSE;
         }
      }
   }
   protected function noArg() { return "noArg called\n"; }
   protected function oneArg($arg) { echo "oneArg($arg) called\n"; }
}

$o = new Overloading();
echo $o->overloaded();
$o->overloaded(1);
```


To implement overload in PHP a trick using the _call magic method is required (see example): this method is triggered when invoking inaccessible methods in an object context.

When an overloaded method is called, since it does not exists, your __call function is called and provided with method name, and arguments. You can modify what you want to happen depending on what is passed and what is called. You can also check arguments types with is_array, is_numeric, etc.

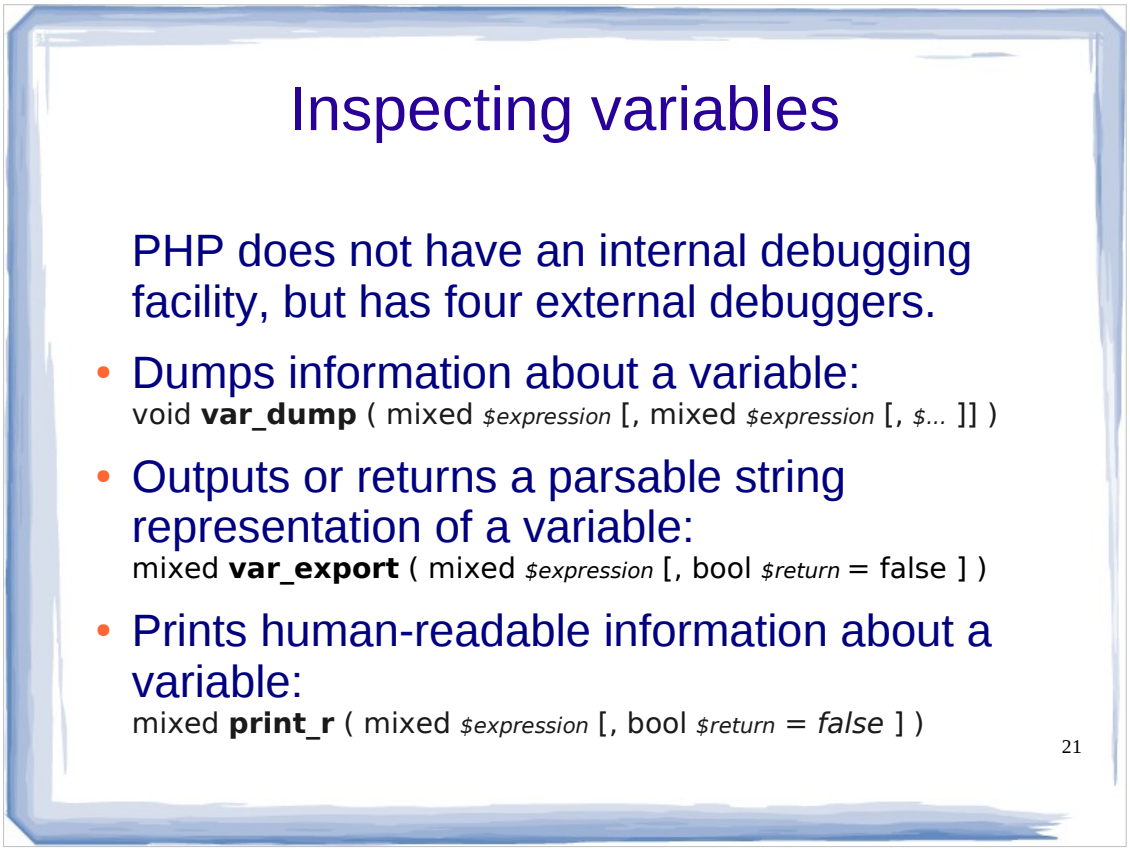
You can also do this with properties with magic methods __get an __set.



A bit of Runtyme Type Information (RTTI). In PHP get_class can only be applied to objects. E.g. a warning will be issued if you try to get the class name of an array or a string or call get_class without object from outside a class. Inside a non-static method get_class() can not be the same as get_class($this), if inheritance is involved.

get_class(obj) is the same as obj.getClass().getName() in Java. In Java getName is a method of java.lang.Class that returns the fully qualified name of the entity (class, interface, array class, primitive type, or void).

How to find the actual calling class of a static method? Don't use get_class(), use string **get_called_class** (void) which uses Late Static Binding to resolve the target class for a static method call at runtime rather than when it is defined.



var_dump: this function displays structured information about one or more expressions that includes its type and value. Arrays and objects are explored recursively with values indented to show structure.

var_export: similar to var_dump() with one exception: the returned representation is valid PHP code.

print_r(): displays information about a variable in a way that's readable by humans.

print_r(), var_dump() and var_export() will also show/return protected and private properties of objects in PHP 5, not only public ones. Static class members will not be shown.

Late Static Bindings

- TODO

The callback pseudo-type

- Example values:

```
'function_name', array($obj, 'methodName'),
array('ClassName', 'staticMethodName'),
array('ClassName::staticMethodName')
```

and closures...

- Make an array of strings all uppercase:

```
$array = array_map('strtoupper', $array);
```

- Sort an array by the length of its values:

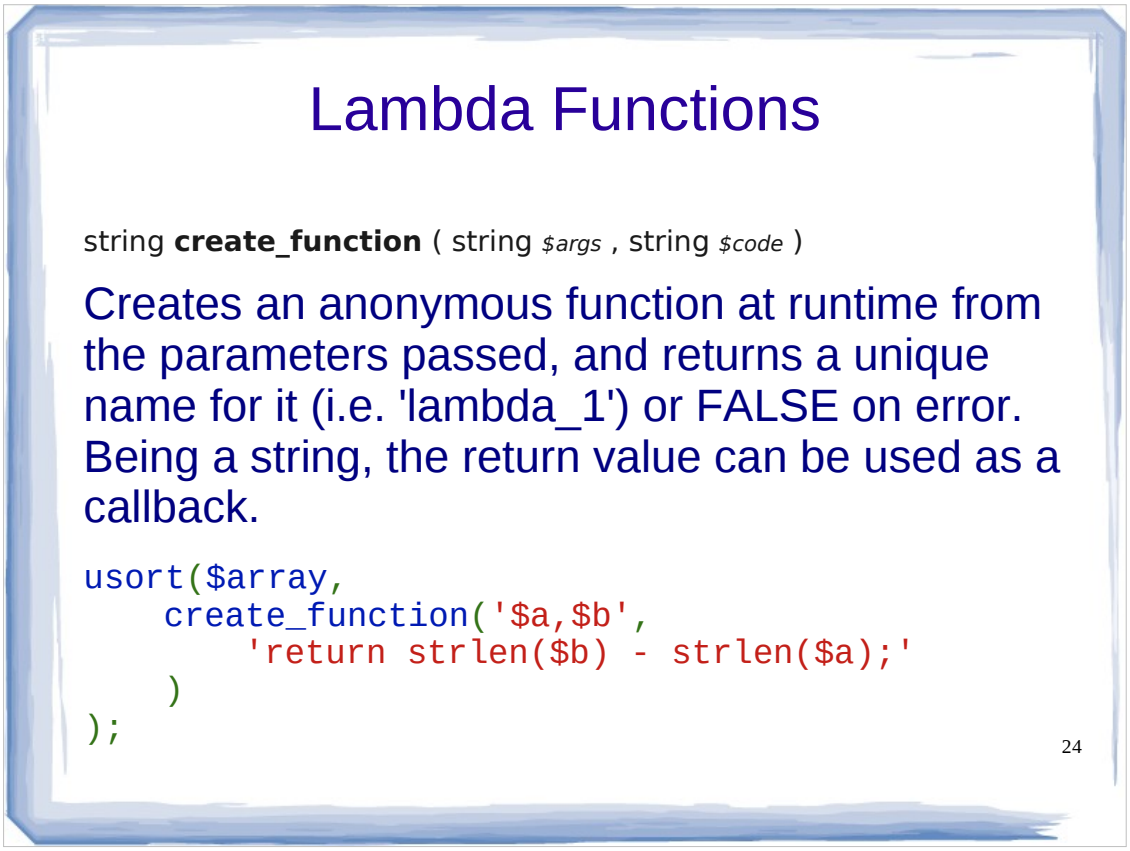
```
function cmp($a, $b){
    return strlen($b) - strlen($a);
}

usort($array, 'cmp');
```


callback is not a distinct type of the language: it is represented by other real types depending on the situation.

Some PHP library functions accept user-defined callback functions as a parameter. A PHP function (either built-in or user-defined) is passed by its name as a string. A method of an instantiated object is passed as an array containing an object at index 0 and the method name at index 1. Static class methods can also be passed without instantiating an object of that class by passing the class name instead of an object at index 0.

“I made an anagram machine and I have an array of positive matches. The trouble is they are all in a different order, I want to be able to sort the array so the longest array values appear first” – from stackoverflow.com. The solution shown uses a usort() (SORT with a User callback). As a side note usort() is an unstable sort. That means may change the order of the elements that compare equal.



Inspired from functional languages as LISP, PHP supports function types (aka first-class functions), but only for functions with no name: functions are objects that can be passed as a parameter, returned from other functions, or assigned into a variable. New functions can be defined at runtime.

Lambda-style (anonymous) functions allow the quick definition of throw-away functions that are not used elsewhere. They are most useful as callback functions. Advantages: increase locality of definition and thus readability Disadvantages: the function is compiled at run time and not at compile time so opcode caches can't cache the function.

Closures

function [**&**] ([formal parameters]) [use ($var1, $var2, &$refvar)] { ... }

```
usort($array, function($a, $b) {
    return strlen($b) - strlen($a);
});
```

TODO: example with “use”

create_function does not create closures. It is merely a convenience function that generates a unique name for a **regular** function

PHP also supports closures, aka anonymous functions. They make lambda functions even more useful: a closure is a lambda function that may also inherit variables from the parent scope. Any such variables must be declared in the function header. The parent scope of a closure is the function in which the closure was declared (not necessarily the function it was called from). By default, all imported variables are copied as values into the closure. This makes it impossible for a closure to modify the variable in the parent scope. By prepending an & in front of the variable name in the use declaration, the variable is imported as a reference instead. In that case, changes to the variable inside the closure will affect the outside scope.

It is possible to also pass a closure to a callback parameter.

PHP automatically converts closure expressions into instances of the **Closure** internal class which will be the type of the variable you assign a closure to.

Type Hinting

```
function test(MyClass $typehintexample = NULL) {}
function array_average(array $arr) { ... }
function average($val) {
        if (is_numeric($val)) return $val;
        /* a warning will be issued if we get
        here and $val isn't an array */
        return array_sum($val) / count($val);
}
```

- If the type does not match a PHP Catchable fatal error occurs (it's not an exception)

Unlike Java which is a strongly typed language, PHP is basically an untyped language as most scripting languages are. This typeless nature provides for the flexibility needed most of the times. Though some type checking is provided as an optional feature. So that you can make code easier to read and less error prone without too much runtime performance bottlenecks .

You can force function/method parameters to be objects (by specifying the name of the class in the function prototype) or arrays. However, if NULL is used as the default parameter value, it will be allowed as an argument for any later call.

Static type hints, e.g. with int and strings, aren't supported yet. They will probably be in PHP 6.

You can mix type hinted code with the non-type hinted one.

A catchable fatal error is not an exception: it can be handled traditionally with set_error_handler() or converted in an exception with the help of the ErrorException class.

PHP Arrays

- A PHP array is a ordered map optimized for several uses:
  numerical (indexed), associative or mixed array - multidimensional array - list (vector) - hash table – dictionary – collection – stack – queue – tree – coffe machine :-)
- Keys may only be integers or strings
- Values may be any value of any type, including other arrays
- No fixed number of values, no single type!
- Can be created/modified with square bracket syntax.
  A rich library of array functions is available.

27

A map is a type that associates values to keys.

PHP arrays are created by the array() language construct. It takes as parameters any number of comma-separated *key* => *value* pairs.

The indexed and associative array types are the same type in PHP, which can both contain integer and string indices.

It is common to use an array to hold one or more key =>value pairs to set attribute values for an object. a function, a method etc. This relieves the programmer from having to remember the order and default values of parameters. Associative arrays used as parameter lists are inspired from Tcl (Tool Command Language), the only good general-purpose language with built-in named parameters support.

Kludges for constant arrays

- static $CONSTARRAY = array(...);

  ... Class::$CONSTARRAY ...
- static function CONSTARRAY() { return array(...); }

  ... Class::CONSTARRAY() ...
- define('CONSTARRAY', serialize(array(...)));

  ... unserialize(CONSTARRAY) ...

  define('CONSTARRAY', 'return ' . var_export(array(...), 1) . ';');

  ... eval(CONSTARRAY) ...

28

Unfortunately you can't define an array as a class or global constant. Some sensible work-arounds are:

- define AND initialize a public static array variable instead; however, you have no guarantee that it will not be modified outside your class
- return it in a static method; client code will need to save it to a variable before indexing
- serialize the array and define it as a constant string;
- ditto plus you need to unserialize before use: var_export & eval can also be used. Definition has to be made outside the class using the old define() function, because the new keyword const does not yet supports expressions.

PHP Exception Handling

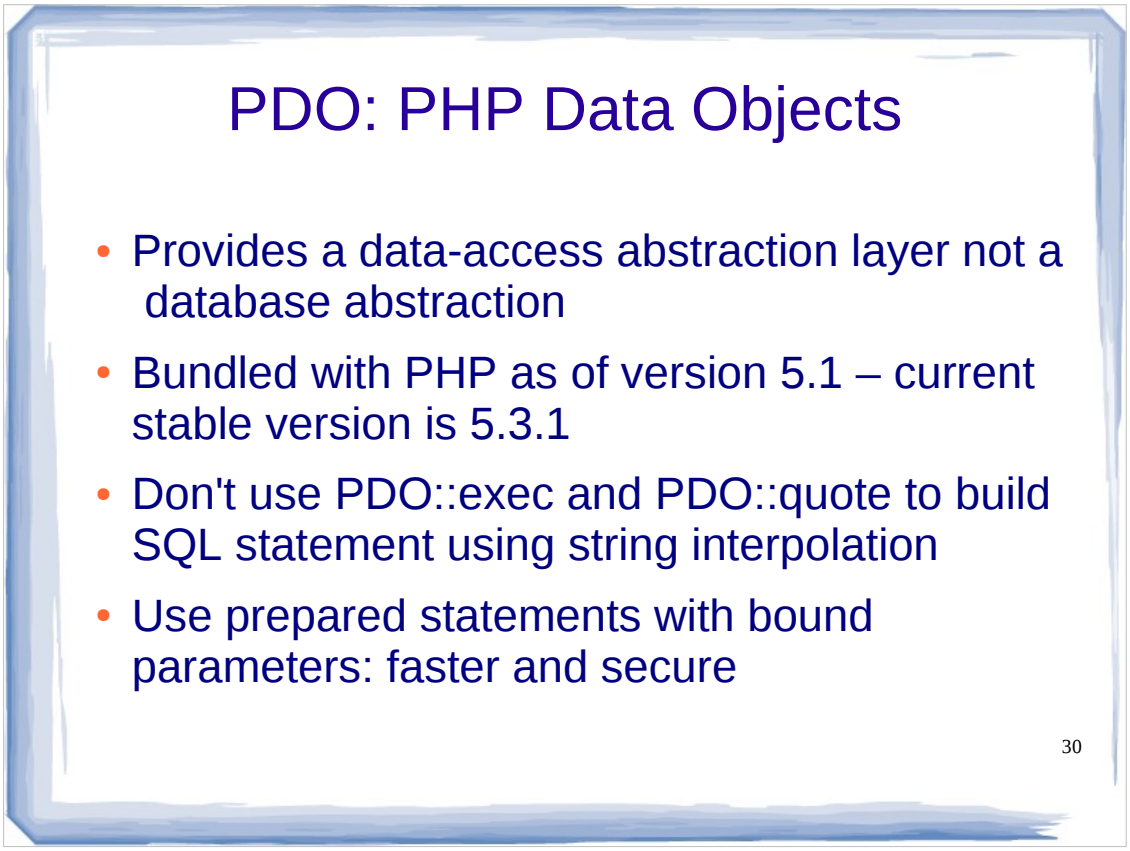
- Runtime exception handling in PHP resembles that of Java, but somewhat simplified
- Unhandled exceptions are always forwarded; catching and re-throwing them within a catch block is allowed (as in Java). Only needed if you want to change the exception object to forward up.
- There is no throws clause, that is there's only one exception type: unchecked
- Exception class (PHP) =Throwable class (Java)
- PHP has no finally block – this is a hole in the language

29

If an exception is not caught, a PHP Fatal Error will be issued with an "*Uncaught Exception ...*" message and a stack trace - that is the program terminates, as in Java, but if a user-defined top-level exception handler function has been defined with set_exception_handler() it will be triggered instead. .

Internal PHP functions mainly use the traditional error-management techniques (Error reporting), only modern Object oriented extensions use exceptions. However, errors can be simply translated to exceptions with the help of an Exception subclass called ErrorException.

In PHP there is no “catch or specify requirement” as is enforced in Java for checked exceptions: simply any thrown exception will propagate up your stack until it is either caught or runs out of stack. This is like declaring all classes (or methods) in Java as "class ClassName throws Exception". That is PHP will never force you to catch exceptions by compile-time checks.



That means PDO does not rewrite SQL or emulate missing features in a certain RDBMS implementation. It is not a full abstraction layer: regardless of which database you're using, you use the same functions to issue queries and fetch data. Each database driver that implements the PDO interface can expose database-specific features as regular extension functions.

Remember during PHP configuration we enabled both the PDO extension and the specific PDO driver for MySQL, which we will use here.

Prepared statements are more portable: PDO::quote is not available for all drivers (notably PDO_ODBC), immune to SQL injection, easier to use – you don't need to escape/quote/format-check any data and often much faster than interpolated queries, as both the server and client side can cache a compiled form of the query.

PDO: prepared statements

```
$dbh = new PDO('mysql:dbname=testdb;host=localhost', 'user', 'pwd');
$sth = $dbh->prepare('... ? … ? …');
        or - you cannot mix ? and :name in the same query:
$sth = $dbh->prepare('... :name … :othername …');
$sth->bindParam/Value(1, ...); $sth->bindParam(2, ...); or:
$sth->bindParam/Value(':name', ...); opt. add param type
$sth->execute();
shortcut for input only and all-strings parameters:
$sth->execute(array(..., ...)); or:
$sth->execute(array(':name' => ..., ':othername' => ...));
```


They can be thought of as a kind of compiled template for the SQL that an application wants to run, that can be customized using variable parameters.

named (:name) or question mark (?) parameter markers will be substituted for real values when the statement is executed

Any user input must be bound using parameters and must not be included directly in the query.

You cannot bind multiple values to a single named parameter in, for example, the IN() clause of an SQL statement. Workaround: build a list of ? as IN(?,?,?,?) using str_repeat().

If the DB server supports prepared statement, PDO::prepare() will check the statement and returns FALSE or emits and exception in case of syntax errors.

PDO: some param/value data types

bool PDOStatement::bindValue(mixed $parameter, mixed $value [, int $data_type = PDO::PARAM_STR])

bool PDOStatement::bindParam(mixed $parameter, mixed $value [, int $data_type = PDO::PARAM_STR [, int $length]])

PDO::PARAM_BOOL	boolean
PDO::PARAM_NULL	NULL
PDO::PARAM_INT	integer
PDO::PARAM_STR	string (default)

PDO::PARAM_INPUT_OUTPUT OR-bitwise with one of the above to bind and INOUT parameter (only useful with stored procedures). A length must follow.

include/require quirks

- These statemens includes and evaluates the specified file. Slower _once versions are available.
- Use require if you want a missing file to halt processing of the page; include if you want your script to continue regardless
- Path defined: absolute, relative: if an included script contains another include with a relative path, note this is only based off of the first script current working directory, and not that of the included script
- No path is given: include_path will be used

33

The two constructs are identical in every way except how they handle failure. Include produces a Warning while require results in a Fatal Error.

An absolute pathname starts with a "/" on unix, and with a drive letter and colon on Windows.

The PHP include_path configuration directive contains a list of directories that PHP searches for files to include for which only the file name, without any path, is specified . The format is like the system's PATH environment variable: entries are column-separated in Unix and semicolon-separated in Windows.

Using a . in the include path allows for relative includes as it means the current directory. However, it is more efficient to explicitly use *include './file'* than having PHP always check the current directory for every include. Default value example: .:/usr/share/pear

Autoloading Classes

```
function __autoload($class_name) {
    require_once $class_name . '.php';
}
```

- The autoload function must be defined in main scope in "\" namespace.
- To keep things simple, it is advisable to keep source files in the filesystem in a directory tree correspondent to that of namespaces.

Every time you want to use a new class in your PHP project, first you need to include this class (using include or require language construct, that’s right this are not functions). However if you have __autoload function defined, inclusion will handle itself: the __autoload function is automatically called in case you are trying to use a class/interface which hasn't been defined yet. By calling this function the scripting engine is given a last chance to load the class before PHP fails with an error.

It is a good practice to create one PHP source file per-class definition.

PHP Namespaces

- Declaration: namespace MyWebapp; (that is \MyWebapp)
- Namespaces can nest into others:

  namespace CompanyName\ProjectName\Library\Database;
- Before, to avoid name clashes, long prefixes were used: CompanyName_ProjectName_Library_Database_ClassName
- Namespace aliasing allows to shorten long qualified names:

  use CompanyName\ProjectName\Library\Database [as Database];

  $obj = new Database\ClassName();
- Single classes can also be aliased (not constants or functions):

  use CompanyName\ProjectName\Library\Database\ClassName as C;

  $obj = new C();

35

The NS definition must be the first command at the top of the PHP file with no HTML or white space preceding it. If it lacks, by default, all constants, class, and function names are placed in the global or root NS (\).

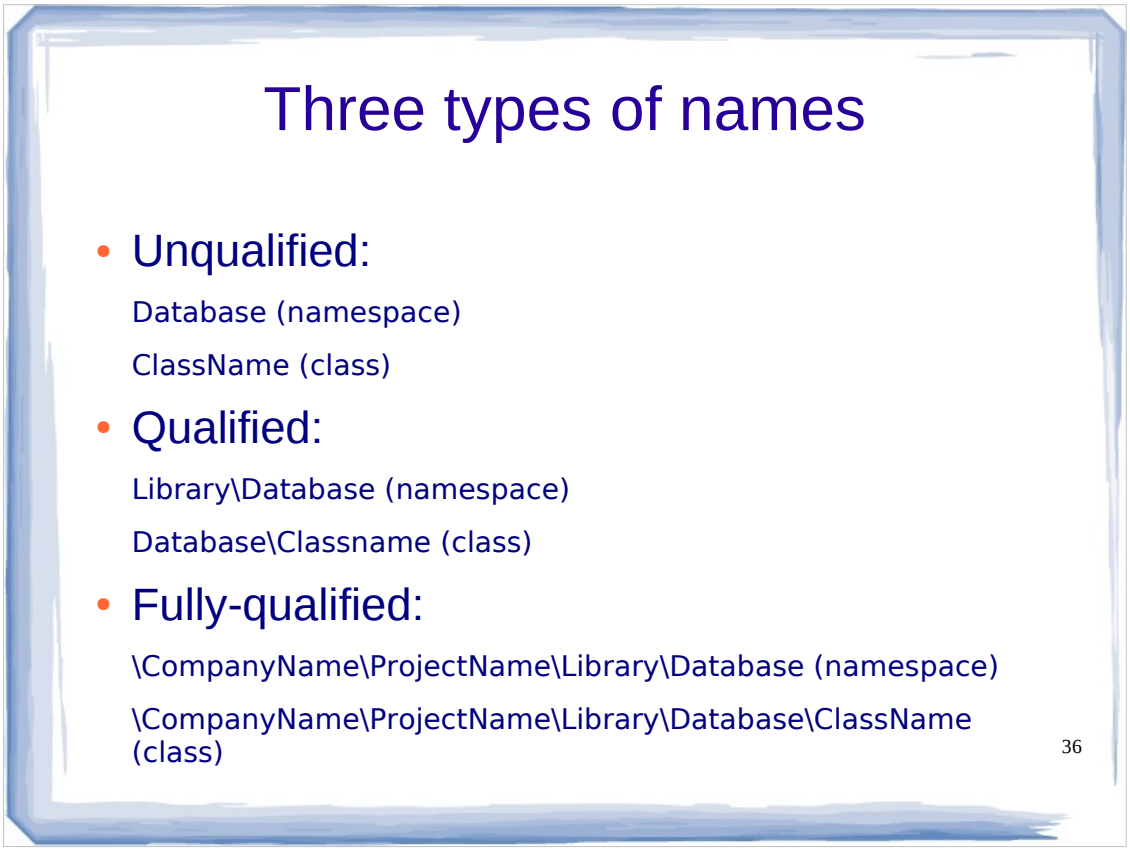
PHP allows you to define a hierarchy of NSs so libraries can be sub-divided - just like directories in a file system group related files and allow different files to have the same name, if they're not in the same directory.

NSs avoids name collitions between your and third-party or internal PHP code.

NS aliasing (also called importing) is akin to symbolic links in filesystems.

The same NS can be defined in multiple files; a single file may define multiple NSs in sequence (only useful if you want combine multiple PHP scripts into one) – NS declarations cannot be nested. A code block can only belong to one NS.

Declaration and Import names must be fully qualified, they are not processed relative to the current namespace. The leading \ is unnecessary.



Namespace terminology (\ = namespace separator):

Unqualified: identifier without a \. They are resolved first in the current namespace. If not found and the code is not global they are searched in the global namespace, but only for constants and functions, not class names (use \ClassName if you want to refer to a global class). This rule exists to make it easier to use in namespaced context the many constants and functions from the traditional procedural PHP standard library that reside in the global namespace, without the need to prefix them with a \. This way existing non-namespaced code can be namespaced with little modifications: only internal or non-namespaced user classes must be fully qualified. If a namespace defines a constant or function with the same name as a standard PHP one, the \ is mandatory.

Qualified: identifier with at least one \. These resolves starting from the current namespace (which will be the global \ namespace if the code is global).

Fully-qualified: identifier starting with the \ (except after operators `namespace` and `use`). Unambiguos and thus resolved at compile time. Often very long and only practical for one-off function calls or object initialization. When you are making more than one call, use alias.

Data Filtering

- Validation: check if the data meets certain qualifications: BOOLEAN, EMAIL, FLOAT, INT, IP, REGEXP, URL
- Sanitization: alter the data, e.g. by removing undesired characters, applying URL-encoding, adding slashes, stripping tags, HTML-escaping

mixed **filter_input** (int *$type* , string *$variable_name* [, int *$filter* = *FILTER_DEFAULT* [, mixed *$options*]])

mixed **filter_input_array** (int *$type* [, mixed *$definition*])

37

Data coming from untrusted external sources, like user supplied input coming from an HTML form, should be filtered before use for both security and robustness. There are two main types of filtering (validation and sanitization) and they can be applied both to the same data with many flags and options available. In the few cases where they are not enough, user-defined function to filter data can be defined.

$type: one of **INPUT_GET, INPUT_POST, INPUT_COOKIE, INPUT_SERVER, INPUT_ENV...**

$filter: filter to apply (a constant)

$options: associative array of options or bitwise disjunction of flag.

$definition: $filter or array defining multiple values to filter; key=$variable_name, value=$filter or $options