PHP

For Programmers

Andrew Beak

FOR DUCKEMS, BEASTLY, AND BRATFACE
FOR WHOM I WOULD CHANGE THE WORLD
IF ONLY I HAD THE SOURCE CODE

Contents

Introduction

Welcome to what I hope is an accessible reference that helps you quickly find and learn relevant facts about PHP programming. I'm writing it with the following readers in mind:

1. Intermediate PHP programmers with two or three years of experience who are hoping to sit the Zend certification exams.
2. Programmers who are proficient in another language but want a quick reference book to dive into PHP.

This book is specifically not an introduction to programming and no attempt is made to introduce basic topics. It is purely a reference to learn the idiosyncrasies of PHP. Additionally it does not intend to replace the PHP manual but rather to focus the reader's attention on aspects of PHP relevant to the Zend certification exams.

It also won't make you a better programmer as I won't be discussing design patterns or programming concepts. This book just focuses on referencing the language of PHP.

This document tries to avoid having an opinion about coding standards such as PSR2 but it is strongly recommended that a practicing developer learn these in parallel.

Throughout this manual if a feature belongs only to a particular version of PHP I'll use mark it in brackets. As an example this (PHP 5.6+) will mean that the feature is available only in PHP version 5.6 and above.

Liberal use of footnotes linking to manual pages and other reference pages is made. This guide cannot possibly cover the full depth of the PHP manual and you cannot consider yourself prepared until you have worked through the manual pages. It is essential to follow the footnote links, read the page, and even follow some links through the PHP manual related to the topic.

PHP Basics

Syntax

PHP has its roots in C and was created by Rasmus Lerdorf to be a more accessible language to make webpages in than C. Other languages like Perl and Java have influenced the language as it developed, but the idea that PHP should have a simple syntax has remained dominant.

All statements in PHP must be terminated with a semi-colon. An exception to this is if the statement happens to be the last statement before a closing tag. Coding standards would normally require you to properly terminate every statement.

Whitespace has no semantic meaning in PHP. There is no need to line code up, but most coding standards enforce this to improve code readability. Whitespace may not appear in the middle of function names, variable names, or keywords. Multiple statements are allowed on a single line.

Code blocks are denoted by the brace symbols { }.

Inserting PHP into web pages

Although PHP can be used for other purposes it is really intended to be a text processor to support web pages. Therefore the most common usage of PHP is within HTML pages.

The server must be configured to recognize the PHP tags and pass the code inside them to a PHP processor. There are five ways to indicate to the server that it must include PHP.

	Open	Close	Note
Standard	`<?php`	`?>`	
Echo	`<?=`	`?>`	
Short	`<?`	`?>`	Deprecated
Script	`<script language="php">`	`</script>`	Discouraged
ASP	`<%`	`%>`	Discouraged

The last three methods of including PHP onto web-pages should not be used.

The echo tag allows you to easily echo a PHP variable and the shortened tag makes your HTML document easier to read. Its usage is easiest to understand when it's shown along with the equivalent in standard opening codes. The following two tags are identical:

```
<?= $variable ?>
<?php echo $variable ?>
```

It is quite common in PHP programs to omit the closing tag `?>` in a file. This is acceptable to the parser and is a useful way to prevent problems with new line characters appearing after the closing tag. These new line characters would be sent as output by the PHP interpreter and could interfere with the HTTP headers or cause other unintended side effects. By not closing the script in a PHP file you prevent the chance of new line characters being sent.

Language Constructs

Language constructs are different from functions in that they are baked right into the language. Language constructs can be understood directly by the parser and do not need to be broken down. Functions, on the other hand, are mapped and simplified to a set of language constructs before they are parsed.

Language constructs are not functions, and so cannot be used as a callback function.

They follow rules that are different from functions when it comes to parameters and the use of parentheses.

The PHP manual has a complete list[1], but here are some of the constructs that you should be familiar with:

Construct	Used for
echo	Outputting a value
print	Outputting a string
exit	Outputting a message and terminating the program
die	This is an alias for exit
return	Terminates a function and returns control to the calling scope, or if called in the global scope terminates the program
include	Includes a file and evaluates it. A warning is generated if the file can't be read.
require	Includes a file and evaluates it. If the file can't be read then a fatal error occurs.
include_once	Include a file and evaluates it. Subsequent calls will not result in the file being included and evaluated multiple times.
require_once	As for include_once, but a fatal error instead of a warning is generated if the file can't be read
eval	The argument is evaluated as PHP and affects the calling scope
empty	Returns a Boolean value depending on whether the variable is empty or not. Empty variables include null variables, empty strings, arrays with no elements, numeric values of 0, a string value of '0', and Boolean values of false
isset	Returns true if the variable has been set and false otherwise.
unset	Clears a variable
list	Assign multiple variables at one time from an array

Comments

There are three styles to mark comments.

	Single line	Multiline
Perl style	#	None
C style	//	/* Multi-line comment blocks */
API style	None	/** * API documentation */

[1] https://secure.php.net/manual/en/reserved.keywords.php

API documentation can additionally conform to external standards such as those used by the PHPDocumentor (http://www.phpdoc.org/) project. This tool examines your API style comments and automatically creates documentation for you.

Variables

Variable Types

PHP is a loosely typed language. It is important not to think that PHP variables don't have a type. They most definitely do, it's just that they may change type during runtime and don't need their type to be declared explicitly when initialized.

PHP will implicitly cast the variable to the data type required for an operation. For example, if an operation requires a number, such as the addition (+) operation, then PHP will convert the operands into a numeric format.

You'll be introduced to type juggling in the "Casting variables" section and you'll need to know the rules PHP follows when changing a variable type. For now, you just need to know that PHP variables have a type, that type can change, and although you can explicitly change the type PHP does this implicitly for you.

PHP has three categories of variable – scalars, composite, and resources. A scalar variable is one which can only hold one value at a time. Composite variables can contain several values at a time.

A resource variable points to something not native to PHP like a handle provided by the OS to a file or a database connection. These variables cannot be cast.

Finally PHP has the null type which is used for variables that have not had a value set to them. You can also assign the null value to a variable.

Scalar types

There are four scalar types:

Type	Contains
Boolean	True or False
Integer	A signed numeric integer
Float	A signed numeric double or float value
String	An ordered collection of binary data

Note that strings are not simply a list of characters and may contain binary information such as an image file that has been read from disk.

Composite types

> In PHP 5.6 these are the only variable types that can be type hinted as parameters to a function. In PHP7 support for type hinting scalar types is added.

There are two composite types: arrays and objects. Each of these has its own section in this reference.

Casting variables

This is a very important section of understanding PHP and even very experienced developers may not be aware of some of the rules that PHP uses to cast variables.

In this section I aim to highlight some of the common pitfalls and link to the manual pages where the complete rules can be found.

PHP implicitly casts variables to the type required in order to perform an operation.

It is also possible to explicitly cast variables using one of two options:

1. Use a casting operator
2. Use a PHP function

Casting operators are used by putting the name of the data type you want to cast into in brackets before the variable name. For example:

```
$a = '123';        // $a is a string
$a = (int)$a;      // $a is now an integer
$a = (bool)$a;     // $a is now Boolean and is true
```

There are also PHP functions which will convert a variable to a datatype. These are named in way that is self-documenting: `floatval`, `intval`, `strval`, `boolval`.

You can also call the `settype()` function on a variable which takes the desired data type as a second argument.

> The `boolval()` function was added in PHP 5.5

There are some rules that need to be remembered regarding how variables are cast in PHP. You should read through the manual[2] carefully because there are many trips and traps in type juggling. Also make sure that you read the pages linked to from the type juggling page.

Instead of exhaustively listing all of the rules I'll focus on some of the rules that may be counter-intuitive or are commonly mistaken.

Casting from float to integer does not round the value, but rather *truncates* the decimal portion:

```
$a = 1234.56;
echo (int)$a;    // 1234 (not 1235)
```

Some general rules for casting to Boolean are that:

- Empty arrays and strings cast to false.
- Strings containing numbers evaluate to true as long as the number is not zero. Recall that such strings return false when the `empty()` function is called on them.
- Any integer (or float) that is non-zero is true, so negative numbers are true.

Objects can have the magic method `__toString()` defined on them. This can be overloaded if you want to have a custom way to cast your object to string. We look at this in the section on "Casting Objects to String".

[2] https://secure.php.net/manual/en/language.types.type-juggling.php

Converting a string to a number results in 0 unless the string *begins* with valid numeric data[3]. By default the variable type of the cast number will be integer, unless an exponent or decimal point is encountered, in which case it will be a float.

Here is an example script that shows some string conversions:

```
<?php
$examples = [
    "12 o clock",
    "Half past 12",
    "12.30",
    "7.2e2 minutes after midnight"
];

foreach ($examples as $example) {
    $result = 0 + $example;
    var_dump($result);
}
```

This outputs:

```
int(12)
int(0)
double(12.3)
double(720)
```

Representing numbers

There are four ways in which a number may be expressed in a PHP script:

Notation	Example	Note
Decimal	1234	
Binary	0b10011010010	Identified by leading 0b or 0B
Octal	02322	Identified by leading 0
Hexadecimal	0x4D2	Identified by leading 0x or 0X

Floating point numbers (called doubles in some other languages) can be expressed either in standard decimal format or in exponential format.

Form	Example
Decimal	123.456
Exponential	0.123456e3

The letter “e” in the exponential form is case-insensitive, as are the letters used in the integer formats above.

[3] https://secure.php.net/manual/en/language.types.string.php#language.types.string.conversion

Naming Variables

PHP variables begin with the dollar symbol $ and PHP variable names adhere to the following rules:

- Names are case sensitive
- Names may contain letters, numbers, and the underscore character
- Names may not begin with a number

Coding conventions differ on the use of camelCase, StudlyCase, or snake_case but all of these formats are valid PHP variable name formats.

PHP also allows for variable variable names (PHP7-). This is best illustrated by example:

```
<?php
$a = 'foo';
$$a = 'bar'; // $a is 'foo', so variable $foo is set
echo $foo;   // bar
```

There are several caveats to using variable variable names, including security risks, and the fact that they are deprecated in PHP 7. They can also make your code a little murky to read.

Checking if a variable has been set

The command `isset()` will return true if a variable has been set and false otherwise. It is preferable to use this function instead of checking if the variable is null because it won't cause PHP to generate a warning.

Variables become unset when they become out of scope.

You can use the command `unset()` to manually un set a variable.

> The unset command is not a guaranteed way to free memory even if you manually trigger garbage collection. Although it's beyond the scope of this document it's worth reading up on how Apache with PHP installed as a module handles memory allocation compared to Nginx using php-fpm.

Constants

Constants are similar to variables but are immutable. They have the same naming rules as variables, but by convention will have uppercase names.

They are defined using the `define()` function as shown:

```
<?php
define('PI', 3.142);
echo PI;      // 3.142
```

As of PHP 5.3 you can use the `const` keyword to define constants:

```
const MILES_CONVERSION = 1.6;
echo "5km in miles is " . 5 * MILES_CONVERSION;
```

The `const` keyword must be used to create a namespaced constant.

As of PHP 5.6 you can use static scalar values to define a constant:

```
const STORAGE_PATH = __DIR__ . '/storage';
```

Note the use of the "magic" constant `__DIR__` that is set by PHP at runtime and contains the path that the script resides in on the file system. These constants are discussed in the section "Magic Constants".

> Constants may only contain scalar variable types.

Superglobals

PHP has a number of superglobals that are available automatically to the script. Superglobals are available in every scope. You are able to alter the values of superglobals but it's generally suggested to rather assign a locally scoped variable to the superglobal and modify that.

You need to know what each of the superglobals stores.

Superglobal	Stores
$GLOBALS	an array of variables that exist in the global scope
$_SERVER	an array of information about paths, headers, and other information relevant to the server environment
$_REQUEST	POST and GET request variables
$_POST	Variables sent in a POST request
$_GET	Variables sent in a GET request
$_FILES	An associative array of files that were uploaded as part of a POST request
$_ENV	An associative array of variables passed to the current script via the environment method.
$_COOKIE	An associative array of variables passed to the current script via HTTP Cookies.
$_SESSION	An associative array containing session variables available to the current script.

The `$_SERVER` superglobal has many keys, and you should be familiar with them. The PHP manual has a list of them[4] and you should make sure that you've read through the manual and understood all of the keys.

Note that the `$_SERVER['argv']` contains arguments sent to the script, which is distinct from the `$_ENV`. Knowledge of this level of detail is required for the certification exam.

Magic Constants

Magic constants are those which PHP provides automatically to every running script. There are quite a lot of them[5] and you will need to know the error constants, as well as the commonly used constants[6].

[4] https://secure.php.net/manual/en/reserved.variables.server.php
[5] https://secure.php.net/manual/en/reserved.constants.php
[6] https://secure.php.net/manual/en/language.constants.predefined.php

Operators

Types of operators

Arithmetic

	Example	Description
Addition	1 + 2.3	Adds 2.3 to 1
Subtraction	4 – 5	Subtracts 5 from 4
Division	6 / 7	Divide 6 by 7
Multiplication	8 * 9	Multiplies 8 by 9
Modulus	10 % 11	Gives the remainder of dividing 10 by 11
Power	12 ** 13	Raises 12 to the power of 13 (PHP 5.6+)

The above arithmetic operators take two arguments and so are called binary.

The unary operators following take only one argument and their placement before or after the variable changes how they work. There are two unary operators in PHP, namely prefix and postfix. They are named for whether the operator appears before or after the variable that it affects.

- If the operator appears before the variable (*prefix*) then the interpreter will first evaluate it and then return the changed variable.
- If the operator appears after the variable (*postfix*) then the interpreter will return the variable as it was before the statement executed and then increment the variable.

Lets show their effects on a variable $a that we initialize to 1 and then operate on:

Command	Output	Value of $a afterwards	Description
`$a = 1;`		1	
`echo $a++;`	1	2	Postfix
`echo ++$a;`	3	3	Prefix
`echo $a--;`	3	2	Postfix
`echo --$a;`	1	1	Prefix

Logic operators

PHP uses both symbol and word form logic operators. The symbol form are C based.

Operator	Example	True when
and	$a and $b	Both $a and $b evaluate true
and	$a && $b	
or	$a or $b	Either $a or $b evaluate true
or	$a || $b	
xor	$a xor $b	One of (not both) $a or $b is true
not	! $a	$a is not true (false)

It is best practice not to mix the word form (e.g.: “and”) and the symbol (e.g.: “&&”) in the same comparison as the operators have different precedence.

Ternary operator

PHP implements the ternary operator as illustrated in this example:

```
$a = 'foo';
$b = (isset($a)) ? $a : 'bar';
echo $b;    // foo
```

If the true value is omitted in the ternary operator then the statement is evaluated as the expression, as follows:

```
$a = true;
$b = $a ?: 'foo';
echo $b;   // 1
```

This shortened version of the ternary operator is not suitable for testing if a variable exists as the interpreter will throw a warning in this case.

Bitwise

Bitwise operators work on the bits of integers represented in binary form. Using them on a different variable type will cause PHP to cast the variable to integer before operating on it.

There are three standard logical bitwise operators:

Operator	Description
&	Bitwise AND – The result will have a bit set if both of the operands bits were set
|	Bitwise OR – If one or both of the operands have a bit set then the result will have that bit set
^	Bitwise XOR – If one and only one of the operands (not both) has the bit set then the result will have the bit set.

The result of a bitwise operator will be the value that has its bits set according to the rules above.

PHP also has operators to shift bits left and right. The effect of these operators is to shift the bit pattern of the value either left or right while inserting 0 bits in the newly created empty spaces.

It's important to be cautious when using bitwise operations to perform calculations as the integer overflow size may vary between the different environments that PHP is deployed on.

For example on a 32 bit integer system the following statements will not echo out the same result:

```
$x = 1;
echo $x << 32;
echo $x * pow (2, 32);
```

The first line will echo 0 as shifting left 32 bits will fill the 32 bit integer with 0 bits. The second line will use the maths library and output the correct value of 2 raised to the power of 32.

Assignment operators

PHP uses the = symbol as an assignment operator. The following line sets the value of $a to 123.

```
$a = 123;
```

The assignment operator can be combined with just about all of the binary and arithmetic operators. This syntax serves as a shortcut that is best shown by providing an example of equivalent statements:

```
$a += 345;    // equivalent to $a = $a + 345;
$a .= 'foo';  // equivalent to $a = $a . 'foo';
```

Reference operator

By default PHP assigns all variables other than objects by value and not by reference. Objects are always assigned by reference.

PHP has optimizations to make assignment by value faster than assigning by reference (see the section on "Memory management"), but if you want to assign by reference you can use the & operator as follows:

```
$a = 1;
$b = &$a;  // assign by reference
$b += 5;
echo $a;  // 6
```

> Creating an object by reference is deprecated, so this code will generate a warning:
> $a = &new myClass;

Comparison operators

PHP uses the following comparison operators:

Operator	Description
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
<>	Not equal
==	Equivalence – values are equivalent if cast to the same variable type.
===	Identity – values must be of the same data type and have the same value.
!=	Not equivalent
!==	Not identical

It is important to understand the difference between an equivalent comparison and an identity comparison:

- Operands are equivalent if they can be cast to a common data type and have the same value.
- Operands are identical if they share the same data type and have the same value.

Be careful when using comparison operators on strings, or when using them on mismatching variable types. See the section on "Casting variables" for more detail.

Two more operators

PHP provides an operator to suppress error messages. This will only work if the library that the function is based on uses PHP standard error reporting.

```
$dbConnection = @mysqli_connect(…);
```

It's good practice to suppress errors using PHP settings in production rather than using the @ error suppression operator.

The last operator we will discuss is the backtick operator. It is not commonly used and is equivalent to calling the `shell_exec()` command. In the following example the variable $a will contain the name of the user running the PHP interpreter.

In a web environment this will probably be `www-data`. This is the default for Nginx and Apache, but from the command line will be the name of the user logged in.

```
$a = `whoami`;
```

Operator Precedence

The PHP manual is the best place to read about operator precedence (https://secure.php.net/manual/en/language.operators.precedence.php).

Precedence can be forced by using parenthesis.

Note that the logical operators in the form of words and symbols have different precedence. In other words the "or" operator has a different precedence from the "||" operator. This means that you should use either word form or symbol form in a comparison statement, but not both.

Control Structures

Conditional Structures

PHP supports `if`, `else`, `elseif`, `switch`, and ternary conditional structures.

"If" structures look like this:

```
if (condition) {
    // statements to execute
} elseif (second condition) {
    // statements to execute
} else {
    // statements to execute
}
```

Note that the space between `else` and `if` in "`elseif`" is optional.

The ternary operator has been discussed in the operators section of this reference.

"If" statements may be nested.

The switch statement looks like this:

```
switch ($value) {
    case ‘10’ :  // statements to execute
        break;
    case ‘20’  : // statements to execute
        break;
    case ‘30’  : // statements to execute
        break;
    default: // statements to execute
        break;
}
```

Once a case matches the value the statements in the code block will be executed until it reaches a `break` command.

If you omit the `break` command then all of the following statements in the switch will be executed until a break is hit even if the case does not match the value. This can be useful in some circumstances but can also produce unintended outcomes if you forget to use the `break` statement.

To illustrate, consider this example:

```
$value = 10;
switch ($value) {
    case ‘10’ :  echo "Value is 10";
                 // no break statement
    case ‘20’  : echo "Value is 20";
                 break;
    case ‘30’  : echo "Value is 30";
                 break;
    default: echo "Value is not 10,20, or 30";
             break;
}
// Value is 10Value is 20
```

Loops

PHP’s most basic loop is the “`while`” loop. It has two forms, as shown:

```
while (expression) {
    // statements to execute
}

do {
    // statements to execute
} while (expression)
```

The difference between them is that in the first form the expression is evaluated at the beginning of the loop and in the second form it's evaluated at the end.

This means that if the expression is false the while loop will not run at all in the first case but it will run at least once in the second case.

The for loop syntax shows the C roots of PHP and looks like this:

```
for ($i = 0; $i < 10; $i++) {
  // do something
}
```

As with C the first statement is executed to initialize the loop, the second condition is evaluated at the beginning of each loop, and the last statement is executed at the end of each loop. The loop will continue to run until the condition evaluates as false.

To iterate over an array you can use `foreach`, as follows:

```
$arr = [ 'a' => 'one',
    'b' => 'two',
    'c' => 'three'];

foreach ($arr as $value) {
    echo $value;   // one, two, three
}

foreach ($arr as $key => $value) {
    echo $key;    // a, b, c
    echo $value;  // one, two, three
}
```

Breaking out of loops

There are two ways to stop an iteration of a loop in PHP – `break` and `continue`.

Using "`continue`" has the effect of stopping the current iteration and allowing the loop to process the next evaluation condition. This allows you to let any further iterations which would run to occur.

Using "`break`" has the effect of stopping the entire loop and no further iterations will occur.

The break statement takes an optional integer value that can be used to break out of multiple levels of a nested loop. If no value is specified it defaults to 1.

Namespaces

Namespaces help to avoid naming collisions between libraries or other shared code. A namespace will encapsulate the items inside it so that they don't conflict with items declared elsewhere.

They can be used to avoid having to use overly descriptive names for classes, sub-divide a library into sections, or limit the applicability of constants to one section of code.

The namespace declaration must occur straight after the opening `<?php` tag and no other statements may precede it.

Namespaces affect constants, but you must declare them with the `const` keyword and not with define.

It is possible to have two namespaces in a file but most coding standards will strongly discourage this. In order to accomplish this you would wrap the code for the namespace in braces, as in this example:

```
<?php
namespace A {
   // this is in namespace A
}
namespace B {
   // this is in namespace B
}
namespace {
   // this is in the global namespace
}
```

This usage is far from standard and in most cases a namespace declaration does not include the braces and all the statements in a file exist in only one namespace.

Fully qualified namespace names

If you are working in a namespace then the interpreter will assume that names are relative to the current namespace.

Consider this class as a basis for our following examples:

```php
<?php namespace MyApp\Helpers;

class Formatters {
    public static function asCurrency($val) {
      // statement
    }
}
```

If we want to use this class from another class we need to provide a fully qualified namespace, as in this example:

```php
<?php namespace MyApp\Lib;
echo MyApp\Helpers\Formatters::asCurrency(10);
```

Alternatively you may use the "`use`" statement to import a namespace so that you don't have to use the long format all the time:

```php
<?php namespace MyApp\Lib;
use MyApp\Helpers\Formatters;
echo Formatters::asCurrency(10);
```

You may precede a name with a backslash to indicate that you intend to use the global namespace, as in this example:

```php
<?php namespace MyApp;
throw new \Exception('Global namespace');
```

In this example if we had not indicated the global scope with the backslash the interpreter would look for a class called Exception within the MyApp namespace.

Errors

PHP has a number of magic constants that are used in relation to errors. These constants are used when configuring PHP to hide or display errors of certain classes.

Here are some of the more commonly seen error codes:

Code	Description
E_DEPRECATED	The interpreter will generate warnings of this type if you use a language feature that is deprecated. Your script continues to run.
E_STRICT	Similar to E_DEPRECATED, this indicates that you are using a language feature that is not currently standard and might not work in the future. Your script continues to run.
E_PARSE	Your syntax could not be parsed and so your script won't start
E_NOTICE	An informational message. The script will continue to run.
E_WARNING	These are non-fatal warnings. Your script will continue to run.
E_ERROR	The script cannot continue to run and is being terminated

Displaying or Suppressing error messages

Generally you want to hide all error messages while in production and your code should run without generating warnings or messages.

This means that in your development environment you want all errors to be displayed so that you can fix all the issues that they relate to, but while in production you want to suppress any messages being sent to the user.

To accomplish this you need to configure PHP using the following settings in your php.ini file:

- display_errors can be set to false to suppress messages
- log_errors can be used to store error messages in log files
- error_reporting can be set to configure what errors trigger a report

Gracefully responding to errors

Instead of your script falling over with the white screen of death you want to be able to manage error situations and provide some form of meaningful feedback to your user.

PHP makes this easy with the `set_error_handler()` function. It takes as a string argument the name of the function that you want to use to catch errors.

Exceptions

Exceptions are a core part of object orientated programming and were introduced in PHP 5.0.

Exceptions differ from errors in that:

- They are objects, created from a base Exception class or one of its children
- They bubble up through the call stack if they are not caught
- Uncaught exceptions are always fatal

Extending the base Exception class

The Exception class is like any other and can be extended. This allows you to create flexible error hierarchies and to tailor your exception handling.

As an example lets create an exception class that we can use to signal that there has been a form validation problem:

```php
<?php
class ValidationException extends Exception { }

function myValidation() {
    if (empty($_POST)) {
        throw new ValidationException('No form fields entered');
    }
}
```

Catching Exceptions

Lets continue from the previous example and imagine that we are calling the myValidation function and want to catch exceptions. The syntax for this is as follows:

```php
<?php
try {
    myValidation();
} catch (ValidationException $e) {
    echo "Validation exception caught ";
    echo $e->getMessage();
} catch (Exception $e) {
    echo "General exception type caught";
}
```

Note that there are two catch clauses. Exceptions will be matched against the clauses from top to bottom until the type of the exception matches the catch clause. Since myValidation throws a ValidationException we would expect it to be caught in the first block, but if any other type of exception is thrown in the function then it will be caught in the second catch block.

Note also the method `getMessage()` is being called on the exception object. Other methods in the basic Exception class will give error codes, stack traces, and other information. The PHP manual is the best reference for the prototype for the exception object - https://secure.php.net/manual/en/class.exception.php.

The finally block

PHP 5.5 introduced the `finally` clause to exceptions. This block of code will always be executed, whether an exception is thrown or not. It is executed either after the try block completes or after the exception block has completed.

One common use for the finally block is to close a file handle, but finally can be used wherever you want code to always be executed.

```php
try {
   // perform some functions
} catch (Exception $e) {
   // handle the error
} finally {
   // always run these statements
}
```

Configuration

I can highly recommend that you do some practical work to configure PHP. You can setup a virtual machine[7] on your computer and install Linux[8] on it, which will give you hands on experience.

There are several Windows and Mac packages that offer an all-in-one configuration for PHP but you should make sure that you find the config files and go through them.

Where settings may be set or changed

PHP offers a flexible configuration strategy whereby base configuration settings may be overridden by user configuration files and even at runtime by PHP itself.
It's best to refer to the manual for this and duplicating it here will only result in stale information.

Refer to the following links:

- https://secure.php.net/manual/en/configuration.changes.modes.php
- https://secure.php.net/manual/en/ini.list.php

Php.ini

The PHP ini file defines the configuration for each php environment. An environment here refers to how PHP is run – for example by command shell, as an FPM process, or within Apache2 as a module.

Each environment will have a directory off the main configuration directory which is `/etc/php5` by default on Ubuntu.

The config file is read whenever the server (apache) or process (fpm/cli) starts. This means that if you make a change to the PHP configuration you will need to reload your apache2 server or restart the fpm service. In contrast, changes to the CLI configuration will take effect the next time you run PHP from the shell.

User ini files

These files are checked by PHP when it is operating in fastcgi mode (PHP 5.3+). This is the case when you're using the fpm module, but not in CLI or apache2.

PHP will first check for these files in the directory that the script is running in and work backwards up to the document root. The document root is configured in your host file and is reflected in the `$_SERVER['DOCUMENT_ROOT']` variable.

Nginx is a very popular server choice for running PHP on. It uses a non-blocking threading model that is optimized for concurrent visitors. You can read more details on their blog at https://goo.gl/xazbZX

[7] http://www.oracle.com/technetwork/server-storage/virtualbox/downloads/index.html
[8] http://www.ubuntu.com/download/server

These ini files will override the settings in php.ini, but will only affect settings that are flagged as `PHP_INI_PERDIR` or `PHP_INI_USER`. Refer to the link above for a list of settings and where they may be changed.

The main configuration file has two directives which pertain to user ini files. The first, `user_ini_filename`, governs the name of the file that PHP looks for.

The second, `user_cache_ttl`, governs how often the user file is read from disk.

Apache version of ini files

If you are using Apache then you can use .htaccess to manage user ini settings. They are searched for in exactly the same method as the fastcgi files are.

You must set the `AllowOverride` setting in your vhost config to true in any directories that you want the .htaccess file to be read.

Performance

A great deal of PHP performance issues relate to the deployment environment which is beyond the scope of this reference. We can't possibly discuss the multitude of strategies that exist to improve website performance.

One potential deployment issue with performance worth mentioning in the context of the Zend examination is using the xdebug extension in production. As the name suggests this extension is for debugging and should not be installed in production[9].

Another deployment concern is in keeping your PHP version up to date. PHP is constantly improving its performance and it's a good idea to migrate your code to keep up with new PHP versions.

When considering performance for the Zend examination we focus on memory management, and the opcode cache.

Memory management

Optimizing memory performance in PHP requires some understanding of how the language works.

PHP uses a container called a "zval" to store variables. The zval container contains four pieces of information:

Piece	Description
Value	The value the variable is set to
Type	The data type that this variable is
Is_ref	A Boolean value indicating whether this variable is part of a reference set. Remember that variables can be assigned by reference. This Boolean value helps PHP decide if a particular variable is a normal variable or if it is a reference to another variable.
Refcount	This is a counter that tracks how many variable names point to this particular zval container. This refcount is incremented when we declare a new variable to reference this one.

[9] http://goo.gl/tEuBNy

Variable names are referred to as symbols and are stored in a symbol table that is unique to the scope that the variables occur in.

Symbols are pointed to zval containers.

```
$a += 345;    // equivalent to $a = $a + 345;
$a .= ‘foo’;  // equivalent to $a = $a . ‘foo’;
```

Reference operator” I mentioned that PHP has optimizations for assigning by value.

This is accomplished by PHP by only copying the value to a new zval when it changes, and initially pointing the new symbol to the same zval container.

Here is an example to illustrate:

```
<?php
$a = "new string";
$b = $a;
// the variable b points to the variable a
xdebug_debug_zval( 'a' );
xdebug_debug_zval( 'b' );
// change the string and see that the refcount is reset
$b = 'changed string';
xdebug_debug_zval( 'a' );
xdebug_debug_zval( 'b' );
```

The output of this script is:

```
a: (refcount=2, is_ref=0)='new string'
b: (refcount=2, is_ref=0)='new string'
a: (refcount=1, is_ref=0)='new string'
b: (refcount=1, is_ref=0)='changed string'
```

We can see that until we change the value of $b it is referring to the same zval container as $a.

Arrays and Objects

Arrays and objects use their own symbol table, separate from the scalar variables. They also use zval containers, but creating an array or object will result in several containers being created.

Consider this example:

```
<?php
$arr = ['name' => 'Bob', 'age' => 23 ];
xdebug_debug_zval( 'arr' );
```

The output from this script looks like this:

```
arr: (refcount=1, is_ref=0)=array (
    'name' => (refcount=1, is_ref=0)='Bob',
    'age' => (refcount=1, is_ref=0)=23)
```

We can see that three zval containers are created, one for the array variable itself and one for each of its two values.

Just as for scalar variables if we had a third member of the array with the same value as another member then instead of creating a new zval container PHP will increase the refcount and point the duplicate symbol to the same zval.

Memory leaks in arrays and objects

Memory leaks can occur when a composite object includes a reference to itself as a member. This is more likely to occur in use cases with objects because they are always assigned by reference, possibly for example in parent-child relationships such as might be found in an ORM model.

The PHP manual has a series of diagrams explaining this[10]. The problem occurs when you unset a composite object that has a reference to itself.

In this event the symbol table is cleared of a reference to the zval structure that was used to contain the variable. PHP does not iterate through the composite object because this would result in recursion as it follows links to itself. This means that the member in the variable that is a reference to itself is not unset, and the zval container is not marked as free. There is no symbol pointing to this zval container and so the user cannot free the memory herself.

PHP will clear up these references at the end of the request. Remember that PHP is not intended to be a long-running language and is designed to be a text processor built for serving specific requests within the context of a web application.

Garbage collection

The garbage collector clears circular references which are those where a complex object contains a member that refers to itself.

PHP will initiate garbage collection when the root buffer is full or when the function `gc_collect_cycles()` is called.

The garbage collector will only cause a slowdown when it actually needs to do something. In smaller scripts where there is no leakage it won't cause a performance drop.

Garbage collection is likely to be of benefit in long-running scripts or those where a memory leak is repeatedly created, such as processing a very large amount of database records using ORM models that leak.

The opcode cache

PHP is an interpreted language[11] that is converted into a sequence of instructions that are executed in order. These instructions are called opcodes or bytecode and this process occurs every time the script is run.

An opcode cache stores the converted instructions for a script. Subsequent calls to the script do not require the script to be interpreted prior to being run.

[10] https://secure.php.net/manual/en/features.gc.refcounting-basics.php
[11] https://en.wikipedia.org/wiki/Interpreted_language

In 2013 Zend contributed their optimization engine to PHP. Known as *Opcache* it is baked into distributions of PHP as of version 5.5 and is probably either the most commonly used cache or will become so.

In addition to the cache built into PHP there are a number of third party opcode caches available[12].

Using the opcode cache results in significant performance increases.

Extensions

PHP extensions extend upon the functionality offered by the core language. A number of them are enabled by default into standard repository distributions of PHP.

Installing an Extension

Extensions are enabled through the php.ini file using the "extension" setting to specify the filename of the extension, like this example for mcrypt:

```
extension=mcrypt.so;
```

You can set the extension directory with a setting in your php.ini file like this:

```
extension_dir = "/usr/lib/php5/20121212/mcrypt.so"
```

Different systems may provide convenient ways of installing and enabling extensions.

Checking for Installed Extensions

The extensions installed will display if you call `phpinfo()` or if you use the more specific command `get_loaded_extensions()`[13].

Running "php –m" from the shell will show a list of extensions installed.

You can check if an extension is loaded by `calling extension_loaded()`. This is recommended if you're using a function in an extension that is not loaded by default. Here is an example from the PHP manual:

```
<?php
if (!extension_loaded('gd')) {
    if (!dl('gd.so')) {
        exit;
    }
}
```

[12] https://en.wikipedia.org/wiki/List_of_PHP_accelerators
[13] https://secure.php.net/manual/en/function.get-loaded-extensions.php

Functions

Functions are packages of code that can be used to execute a sequence of instructions. Any valid code can be used inside a function, including calls to other functions and classes.

In PHP function names are *case insensitive* and can be referenced before they are defined, unless they are defined in a conditional code block.

Functions can be built-in, provided by an extension, or user-defined.

Functions are distinct from language constructs.

Arguments

Arguments to a function, also known as parameters, allow you to pass values into the function scope. Arguments are passed as a comma separated list and are evaluated left to right.

Type Declarations

As of PHP 5.0.0 it has been possible to specify that an argument must be an instance of a particular class type. For methods in a class you could also specify that a parameter must be an instance of the same class the method is defined on.

PHP 5.1.0 introduced the ability to specify that an argument must be an array.

PHP 5.4.0 introduced the concept of a callable (more on this later).

PHP 7.0.0 introduced the ability to specify arguments must be of the scalar types, bool, float, int, and string.

Optional Parameters

PHP will create a recoverable fatal error if you do not supply enough parameters to a function. You can specify a default value for a parameter which has the effect of making it optional.

In the following example if the user does not supply a message the function assumes it will be “world”.

```
<?php
function sayHi($message = 'world') {
    echo "Hello $message";
}
sayHi();
```

Overloading functions

In other programming languages overloading usually refers to declaring multiple functions with the same name but with differing quantities and types of arguments. PHP views overloading as providing the means to dynamically "create" properties and methods[14].

[14] https://secure.php.net/manual/en/language.oop5.overloading.php

PHP will not let you redeclare the same function name. However PHP does let you call a function with different arguments and offers you some functions to be able to access the arguments that a function was called with.

Here are three of these functions:

Function	Returns
func_num_args()	How many arguments were passed to the function
func_get_arg($num)	Parameter number $num (zero based)
func_get_args()	All parameters passed to the function as an array

Here is an example showing how a function can accept any number of parameters of any sort, and how you can access them:

```
<?php
function myFunc() {
    foreach(func_get_args() as $arg => $value) {
        echo "$arg is $value" . PHP_EOL;
    }
}
myFunc('variable', 3, 'parameters');
```

This outputs:

```
0 is variable
1 is 3
2 is parameters
```

Variadics

PHP 5.6 introduced *variadics* which explicitly accept a variable number of parameters. By using the ... token you specify that the function will accept a variable number of parameters.

The variadic parameters are made available in your function as an array.

Note that if you are mixing normal fixed parameters with a variadic syntax then the variadic parameter must be the last parameter in the list of parameters.

The PHP manual has a very clear example[15] which shows the interaction between compulsory, optional, and variadic parameters:

[15] https://secure.php.net/manual/en/migration56.new-features.php

```
<?php
function f($req, $opt = null, ...$params) {
    printf('$req: %d; $opt: %d; number of params: %d'."\n",
        $req, $opt, count($params));
}

f(1);
f(1, 2);
f(1, 2, 3);
f(1, 2, 3, 4);
f(1, 2, 3, 4, 5);
```

This outputs:

```
$req: 1; $opt: 0; number of params: 0
$req: 1; $opt: 2; number of params: 0
$req: 1; $opt: 2; number of params: 1
$req: 1; $opt: 2; number of params: 2
$req: 1; $opt: 2; number of params: 3
```

Note that the variadic parameter is made available as an ordinary array `$params`.

References

By default PHP passes arguments to functions by value, but it is possible to pass them by reference. You can do this by declaring the argument as pass by reference, as in this example:

```
<?php
function addOne(&$arg) {
    $arg++;
}
$a = 0;
addOne($a);
echo $a; // 1
```

The `&` operator marks the parameter as being passed by reference. Changes to this parameter in the function will change the variable passed to it.

The ability to pass a variable by reference at call time was removed in PHP 5.4.0. This code will generate a fatal error:

```php
<?php
function addOne($arg) {
    $arg++;
}
$a = 0;
addOne(&$a); // fatal error
echo $a;
```

Variable Functions

Variable functions[16] are similar in concept to variable variable names. They're easiest to explain with a syntax example:

```php
<?php
function foo() {
    echo 'Foo';
}
$var = 'foo';
$var(); // calls foo()
```

As of PHP 5.4.0 you can call any callable as a variable function. We'll discuss callables a little later in the "Callables, Lambdas, and closures" section

Returns

PHP will return NULL if you do not specify a return value for your function using the `return` keyword. Using the return statement will prevent further code from executing in your function.

Return by reference

It is possible to declare a function so that it returns a reference to a variable, rather than a copy of the variable. The PHP manual[17] notes that you should not do this as a performance optimization, but rather only when you have a valid technical reason to do so.

To declare a function as return by reference you place an `&` operator in front of its name:

```php
function &getValue() {...}
```

Then, when calling the function you also place the `&` operator in front of the call:

```php
$myValue = &getValue();
```

After this call the `$myValue` variable will contain a reference to the variable that the `getValue()` function returns.

The function itself must return a variable. If you try to return, for example, a numeric literal like 1 a run-time error will be generated.

[16] https://secure.php.net/manual/en/functions.variable-functions.php
[17] https://secure.php.net/manual/en/language.references.return.php

Two use cases for this are the Factory pattern and for obtaining a resource like a file handle or database connection.

Variable Scope

As in other languages the scope of a PHP variable is the context in which it was defined. PHP has three levels of scope – global, function, and class. Every time a function is called a new function scope is created.

You can include global scope variables into your function in one of two ways:

```php
<?php
$glob = "Global variable";
function myFunction() {
    global $glob; // first method
    $glob = $GLOBALS['glob']; // second method

    $glob = "Changed";
}
myFunction();
echo $glob;  // Changed
```

Note that the two methods have an identical effect of allowing you to use the `$glob` variable in `myFunction()` and have it refer to the `$glob` variable declared in the global scope.

The use of global variables is usually quite strongly discouraged by coding standards as they introduce problems when writing tests and make debugging more difficult.

Callables, Lambdas, and closures

Lambda and Closure

A lambda in PHP is an anonymous function that can be stored as a variable.

```php
<?php
$lambda = function($a, $b) {
    echo $a + $b;
};
```

This variable can be used in functions that accept a callable

A closure in PHP is an anonymous function that encapsulates variables so they can be used once their original references are out of scope. Another way of putting this is to say that the anonymous function "closes over" variables that are in the scope it was defined in.

In practical syntax in PHP we define a closure like this:

```php
<?php
$string = "Hello World!";
$closure = function() use ($string) {
    echo $string;
};

$closure();
```

That looks nearly identical to a lambda but notice the `use ($string)` syntax that occurs just before the code block begins.

The effect of this is to take the `$string` variable which exists in the same scope of the closure and make it available *within* the closure.

Notice that we can call lambdas and closures using the syntax we use for variable functions.

In our lambda example above the function only had access to the parameters it was passed, and nothing from the containing scope would be passed in. Calling `echo $string` would result in a warning because the variable doesn't exist.

Callables

Callables were introduced as a type hint for functions in PHP 5.4.0.

They are callbacks that some functions, for example `usort()`, accept.

A callable for a function such as `usort()` can be one of the following:

- An inline anonymous function
- A lambda or closure variable
- A string denoting a PHP function (but not language constructs)
- A string denoting a user defined function
- An array containing an instance of an object in the first element, and the string name of the function to call in the second element
- A string containing the name of a static method in a class (PHP 5.2.3+)

There are examples of all of these in the PHP manual[18].

[18] https://secure.php.net/manual/en/language.types.callable.php

Strings and Patterns

Declaring Strings

In PHP strings may be declared either as simple type or complex type. The difference is that complex strings will be evaluated with respect to control characters and variables.

Simple strings are declared in 'single quote marks' while complex strings are declared in "double quote marks".

Example:

```
$name = 'Bob';
$a = 'Hello $name\n";
$b = "Hello $name\n";
echo $a;       // Hello $name\n
echo $b;       // Hello Bob
```

In this example the new line character is output after Hello Bob, but in the simple string the literal characters are output.

Notice also that the variable $name is evaluated as the string "Bob" and is inserted into the complex variable $b when it is output.

In order to help the parser interpolate a complex string you may use braces to indicate a variable name. This is necessary, for example, when outputting an element from an array where it might not be immediately clear that the square brackets are intended as punctuation in the string or as syntax to reference an element in the array:

```
$arr = ['one', 'two', 'three'];
echo "Array position 1 is {$arr[1]}";
```

As with other languages the backslash character can be used to escape control characters and quote marks:

```
echo "Hello \'World\'";  // Hello 'World'
echo 'Hello \"World\"';  // Hello "World"
echo "Escaped \\ backslash";  // Escaped \ backslash
```

Control characters

The PHP manual[19] has a list of control characters that may be used, but here they are in table form:

[19] https://secure.php.net/manual/en/language.types.string.php

Sequence	Meaning
\n	Linefeed
\r	Carriage return
\t	Tab
\v	Vertical tab
\e	Escape
\f	Form feed
\\	Backslash
\$	Dollar
\[0-7]{1,3}	Sequences matching this regular expression are in octal notation
\x[0-9A-Fa-f]{1,2}	Similarly, matching sequences are in hexadecimal notation
\u{[0-9a-f]{1,6}}	Matching sequences are a Unicode codepoint which will be output to the string as as that codepoints UTF-8 representation

Heredoc and Nowdoc

A heredoc in PHP is a convenient way to declare a string that spans multiple lines. Instead of having to add in multiple new line characters you can declare the string in one easy format.

Heredoc strings are evaluated for control characters and variables, just like double quoted strings are.

Common uses for heredoc include creating SQL queries, or for creating formatted snippets of HTML for emails or web pages. You can also use them to initialize variables, or anywhere else that you want to use a string that spans multiple lines.

Nowdoc was introduced in PHP 5.3.0 and is to heredoc what single quoted strings are to double quoted strings. In other words Nowdocs are *not* evaluated for special characters and variables.

Heredocs use the syntax like this:

```
<?php
echo <<<HEREDOC
This is a heredoc string, note:
  1) the capitalization of the tag
  2) the tag name follows variable naming rules
  3) where the closing tag is
HEREDOC;
```

The closing tag *must* start on the first character of a new line.

You specify that a string is a Nowdoc and not a Heredoc by wrapping the label in single quotes.

```
<?php
echo <<<'NOWDOC'
This is a nowdoc string, note:
    1) Single quotes around the label
    2) Variables will not be evaluated
    3) Control characters will not be evaluated
NOWDOC;
```

PHP and multibyte strings

A variable-width encoding scheme uses codes of differing lengths to encode a character set. Multibyte encodings use varying number of bytes to encode characters.

Multibyte encoding allows a larger number of characters to be encoded and so represented on a computer. One of the encoding schemes that you will commonly encounter in PHP is UTF-8.

PHP implements strings as an array of bytes with an integer indicating the length of the buffer (not null terminated). It does not dictate a specific encoding for strings and just stores the bytes.

Consequently we can see that PHP does not natively support multibyte encoding. Instead it uses the *mbstring* extension to handle multi-byte strings.

The native string functions in PHP assume strings are an array of single bytes, so functions like `substr()`, `strpos()`, `strlen()`, and `strcmp()` will not work on multibyte strings. You should use the *multibyte equivalents* of those functions, such as `mb_substr()` for example.

Unicode

Unicode was an attempt to unify all the code sets that represented characters. Unicode defines *code points* that are abstract concepts of a character. A Unicode code point represents a character and is written like this: U+0041[20]. That number is assigned to capital "A".

There is no limit on the characters that Unicode can store. There was some confusion originally about Unicode being two bytes, but that related to the encoding scheme and not to Unicode itself.

Unicode is not an encoding system. Encoding is the way in which a Unicode character is represented.

UTF-8 stores all the code points from 0-127 in a single byte. This covers the entire range of the English alphabet, numbers, and some symbols. Code points above 127 are stored in multiple bytes (up to 6 bytes).

Because the Unicode code points from 0-127 match the ASCII table from 0-127 English text encoded in UTF-8 looks exactly the same as if it were encoded in ASCII. Only people who wrote characters with accents would ever end up with a file that was encoded differently from ASCII.
There are hundreds of encoding schemes that are able to store *some* of the Unicode code points, but not all. If you use one of these encodings and encounter a Unicode character that cannot be represented you'll be presented with a question mark or empty box.

[20] http://unicode-table.com/

For example, if your encoding scheme is geared towards storing Hebrew characters and you try to store Russian characters in it then you'll get a bunch of question marks instead of your Russian characters because the encoding scheme doesn't support them.

Telling clients how a string is encoded

You can't always detect how a string is encoded. Unless you know how a string is encoded you won't be able to display it with confidence. It's your job as a PHP programmer to tell the clients reading your HTML output how it is encoded.

You should specify the character encoding scheme being used in the `Content-Type` HTTP header. This lets the client know how your output is encoded and therefore how to display it correctly.

Putting the content type in the HTML as a meta tag is slightly less satisfactory because unless the client knows the encoding type it won't be able to read the HTML to determine the encoding. You can get away with doing it this way, but it's better not to.

Changing between encoding schemes

The mbstring extension provides a number of functions that can be used to help detect and convert between encoding schemes.

The `mb_detect_encoding()` function will go through a list of possible encodings and attempt to determine how the string is encoded. You can change the order of the detection with the `mb_detect_order()` function.

You can use `mb_convert_encoding()` to convert a string between encoding formats.

Matching Strings

Comparing strings in PHP should be done with an appropriate level of care when you're trying to match different variable types. In the section Casting variables we read through the manual pages relating to casting, make sure that you're familiar with how PHP casts various variable types to string.

Using comparison operators like > and < might not always work as expected. It's common to refer to PHP using alphabetical comparison to evaluate strings against these operators.

However, it uses the ASCII value of the character to make the comparison. Lower case letters have a higher ASCII value than capitals so you can have the situation where lower case letters are placed after capitals, like this:

```php
<?php
$a = "PHP";
$b = "developer";
if ($a > $b) {
    echo "$a > $b";
} else {
    echo "$a < $b";
}
// developer comes before PHP in the alphabet
// but this script outputs
// PHP < developer
```

Recall the rules for converting strings to integers that we discussed in the section on "Casting variables". In the below example the string is cast to an integer value of 12, which equals the float value of 12.00 and so the message is echoed.

```php
<?php
$a = "12 o'clock";
$b = 12.00;
if ($a == $b) {
    echo "The mouse ran up the clock";
}
```

Unless you are confident about the strings you're comparing you should consider using the identity operator `===` to make this sort of comparison.

In addition to using operators PHP also provides a number of string comparison functions.

`strcmp()` is a function to perform binary safe string comparisons. It takes two strings as arguments and returns < 0 if str1 is less than str2; > 0 if str1 is greater than str2, and 0 if they are equal.

It has a case insensitive version named `strcasecmp()` that first converts strings to lowercase and then compares them.

This example shows the difference:

```php
<?php
$a = "PHP";
$b = "developer";
$comparison = strcmp($a, $b);
echo $comparison . PHP_EOL; // -20
$caseInsensitive = strcasecmp($a, $b);
echo $caseInsensitive . PHP_EOL; // 12
```

The functions `strncmp()` and `strcasencmp()` can be used to only compare the first "n" characters of two strings.

PHP has a very powerful function called `similar_text()` that will calculate the similarity between two strings. This can be a very computationally expensive procedure for long sections of text. The order that you pass the arguments in is significant, so `similar_text($a, $b) != similar_text($b, $a)`.

Another function, `levenshtein()`, can be used to calculate the Levenshtein distance between two strings. The Levenshtein distance is defined as the minimal number of characters you have to replace, insert or delete to transform str1 into str2.

To compare substrings you can use the binary safe `substr_compare()` function.

PHP 5.6.0 introduced the `hash_equals()` function which is a timing attack safe way of comparing strings. We discuss this in the Security section.

Extracting strings

An individual position in a string can be referenced with the same syntax as an array element. All positions in the string are always zero based – the first character in the string is position 0.

```
<?php
$string = 'abcdef';
echo $string[0];    // a
```

You can use the `substr()` function to return a portion, or slice, of a string. The PHP Manual[21] shows the syntax for the command like this:

```
string substr ( string $string , int $start [, int $length ] )
```

We can see that it takes two compulsory parameters and one optional parameter.

Both the start and the length parameters can be positive or negative.

If the start value is greater than the length of the string `substr()` will return false.

If the start value is positive (or 0) the slice of the string returned starts at the start'th position of the string counting from the beginning.

Otherwise, if it is negative the slice starts at the start'th position from the *ending* of the string.

```
<?php
echo substr("abcdef", 2) . PHP_EOL;    // cdef
echo substr("abcdef", -2) . PHP_EOL;    // ef
```

If length is omitted, as in the above example, then the slice will continue from the slice starting point to the end of the string.

If length is given as a positive number then at most length characters will be returned.

If length is given as a negative number then that many characters will be omitted from the end of the string:

[21] https://secure.php.net/manual/en/function.substr.php

```php
<?php
echo substr("abcdef", 0, 2) . PHP_EOL;     // ab
echo substr("abcdef", 0, -2) . PHP_EOL;     // abcd
```

If length is given and is 0, FALSE, or NULL then an empty string is returned

The PHP Manual gives some more examples:

```php
<?php
echo substr('abcdef', 1);     // bcdef
echo substr('abcdef', 1, 3);  // bcd
echo substr('abcdef', 0, 4);  // abcd
echo substr('abcdef', 0, 8);  // abcdef
echo substr('abcdef', -1, 1); // f
```

Searching Strings

Useful tips

A common complaint about PHP is that it is difficult to tell the order of parameters for searching strings and arrays.

PHP search parameters have a $haystack that we are searching for a $needle.

Compare the order of parameters used for `strpos()` and `array_search()`:

```php
<?php
$arr = ['a', 'b', 'c', 'd', 'e', 'f' ];
$str = 'abcdef';
echo strpos($str, 'c') . PHP_EOL;
echo array_search('c', $arr) . PHP_EOL;
```

It seems at first that sometimes the $needle parameter comes first and sometimes the $haystack parameter comes first.

However, it's a lot more simple when you remember that PHP is using underlying C libraries and the consistent rule is:

- For strings it is always $haystack then $needle
- For arrays it is always $needle then $haystack

The next useful tip is to remember the difference between 0 and false. Although Boolean false evaluates to 0 if you cast it to integer, the number 0 is not identical to Boolean false. Here's an example:

```php
<?php
$string = 'abcdef';
if (strpos($string, 'a') == false) {
    echo "False negative!" . PHP_EOL;
}
```

Remember that strings are zero based, so the first position is position 0. `strpos()` is returning the integer 0 because it found "a" in the first position.

We are using the equality operator `==` to check the result of `strpos()` and so we are falsely reporting that the letter "a" does not appear in the string above.

To handle the case where the substring is genuinely not found you should use the identity `===` operator.

Quick overview of search functions

PHP has a number of functions used to search strings.

`substr_count()` will return the number of substring occurances in a string.

`strstr()` searches for a substring in a string and returns the portion of the haystack that occurs after the first found occurrence. It returns false if no occurrence is found. Note that using `strpos()` is preferable because it is faster.

`stristr()` is a case-insensitive version of `strstr()`. As a general rule the case insensitive functions have an "i" after the prefix.

`strchr()` returns the portion of the string before the first occurrence of the needle.

`strpos()` returns the position of the first occurrence of the needle.

`stripos()` is a case-insensitive version of `strpos()`.

`strspn()` finds the length of the initial segment of a string consisting entirely of characters contained within a given mask.

`strcspn()` returns the length of the initial segment of subject which does not contain any of the characters in mask. In other words it searches for the first occurrence of any of the mask letters in the string and returns the number of characters that exist before it is found.

Replacing strings

PHP has three simple functions for replacing strings. `str_replace()` and it's case-insensitive version `str_ireplace()` can be used for basic replacements.

```php
<?php
echo str_replace('foo', 'bar', 'Delicious food');
```

They both take three parameters, the search string, replacement string, and the string to operate on.

If you pass a fourth variable by reference it will be set to the number of replacements that PHP performed.

Both the search and replacement parameters can be arrays. This lets you replace multiple values in one call, as in this example:

```
<?php
$string = "I like black hot coffee";
$search = ['black', 'coffee'];
$replace = ['green', 'tea'];
echo str_replace($search, $replace, $string);
```

The example above will replace black with green and coffee with tea.

You can use the `substr_replace()` function to replace substrings. `substr_replace()` replaces a copy of the string delimited by the start and (optionally) length parameters with the string given in replacement.

`strtr()` is another function to replace substrings and characters. If only two parameters are supplied the second parameter should be an array of replacement pairs. It otherwise takes three parameters, as in this example from the PHP manual it is being used to convert characters with accent marks to English format characters:

```
<?php
$addr = strtr($addr, "äåö", "aao");
```

The most flexible and powerful way to replace strings is by using the `preg_match()` function which allows you to use regular expressions to find slices of the string to replace. We'll learn more about regular expressions in the "String Patterns – Regular Expressions" section.

Formatting Strings

The `printf()` function is used to output a formatted string. You should read carefully though the PHP manual[22] and make sure that you've practiced using it. The general usage is to specify a formatting string and the values that need to be placed into it.

```
<?php
$minutes = 60;
$timeUnit = "an hour";
printf("There are %u minutes in %s.", $minutes, $timeUnit);
```

In the example you'll notice that the first parameter to `printf()` has two placeholders marked by percentage symbols in it. The following parameters are values that must be typecast and inserted into those placeholders.

There are a number of symbols that can be used to format parameters:

[22] https://secure.php.net/manual/en/function.printf.php

Symbol	Format
%%	a literal percent character. No argument is required.
%b	the argument is treated as an integer, and presented as a binary number.
%c	the argument is treated as an integer, and presented as the character with that ASCII value.
%d	the argument is treated as an integer, and presented as a (signed) decimal number.
%e	the argument is treated as scientific notation (e.g. 1.2e+2). The precision specifier stands for the number of digits after the decimal point since PHP 5.2.1. In earlier versions, it was taken as number of significant digits (one less).
%E	like %e but uses uppercase letter (e.g. 1.2E+2).
%f	the argument is treated as a float, and presented as a floating-point number (locale aware).
%F	the argument is treated as a float, and presented as a floating-point number (non-locale aware). Available since PHP 4.3.10 and PHP 5.0.3.
%g	shorter of %e and %f.
%G	shorter of %E and %f.
%o	the argument is treated as an integer, and presented as an octal number.
%s	the argument is treated as and presented as a string.
%u	the argument is treated as an integer, and presented as an unsigned decimal number.
%x	the argument is treated as an integer and presented as a hexadecimal number (with lowercase letters).
%X	the argument is treated as an integer and presented as a hexadecimal number (with uppercase letters).

PHP formats are locale aware[23] which affects how they represent numbers and dates. For example if you set the locale to Dutch then the date would be output in Dutch. This is shown in an example on the PHP manual:

```php
<?php
/* Set locale to Dutch */
setlocale(LC_ALL, 'nl_NL');
/* Output: vrijdag 22 december 1978 */
echo strftime("%A %e %B %Y", mktime(0, 0, 0, 12, 22, 1978));
```

The PHP Manual warns that the locale information is maintained *per process*, not per thread.

If you are running PHP on a multithreaded server API like IIS, HHVM or Apache on Windows, you may experience sudden changes in locale settings while a script is running, though the script itself never called `setlocale()`.

This happens due to other scripts running in different threads of the same process at the same time, changing the process-wide locale using `setlocale()`.

[23] https://secure.php.net/manual/en/function.setlocale.php

On a POSIX system you can use the shell command "`locale -a`" to list all of the locales it supports. On Windows machines there are pages on the MSDN[24] listing the regions and you can see them in your control panel.

Formatting Numbers

The `number_format()` function is a basic way to format numbers.

`number_format()` is not locale aware and so won't automatically choose the separator characters for you. By default thousands separator is a comma and no decimal places are displayed.

The function takes parameters for the number to be formatted, how many decimal places to display, the character for the decimal point, and the thousands separator character.

You can pass either 1, 2, or 4 parameters to the function. Here is an example:

```
<?php
$number = 1234.5678;
echo number_format($number) . PHP_EOL;
echo number_format($number, 3) . PHP_EOL;
echo number_format($number, 2, ',', '.') . PHP_EOL;
```

This outputs:

```
1,235
1,234.568
1.234,57
```

To format currency you can use the `money_format()` function. It is locale aware and uses the information set by the host system.

In the following example you can see an example of this:

```
<?php
// Locale is British English
setlocale(LC_MONETARY, 'en_GB');
echo money_format('%.2n', "5000000.123");
// Locale is Denmark
setlocale(LC_MONETARY, 'da_DK');
echo money_format('%.2n', "5000000.123");
```

The output looks like this:

[24] https://msdn.microsoft.com/en-us/library/cdax410z%28v=vs.90%29.aspx

```
£5,000,000.12
kr 5.000.000,12
```

String Patterns – Regular Expressions

PHP uses Perl Compatible Regular Expressions (PCRE). Up until PHP 5.3 it also supported POSIX regular expressions, but these are now deprecated.

Regular expressions are a set of rules for matching strings against. The rules are written as a string using a format that describes the pattern you are searching for.

When learning regular expressions you should find an online regex tester that you like. There are several to choose[25] from and they make it a lot quicker to play with expressions and see how they match strings.

Delimiters

Regular expressions are delimited by characters that appear at the beginning and end of each pattern in your expression.

Usually the forward slash is used, but # and ! are also common.

Any character can be used, but the delimiter will need to be escaped inside your expressions so it is standard to choose a delimiter that is not likely to occur in your search expression.

For example if you're going to be searching directories to find those which match a pattern then the forward slash character might not be the best choice of delimiter.

Meta-Characters

Meta-characters are interpreted to have a meaning in the search pattern. They need to be escaped if you intend to have them as a literal part of the expression.

Character	Meaning
\	General escape character
^	Start of subject, or line
$	End of subject, or line
.	Match any character except new line
[	Start defining a character class
]	End defining a character class
|	Start of an alternate branch (like an "or")
(	Start of a sub-pattern
)	End of a sub-pattern
?	0 or 1 quantifier
*	0 or more quantifier
+	1 or more quantifier
{	Start min/max quantifier
}	End min/max quantifier

[25] Here's an example : https://regex101.com/

We'll be building on this as we work through this section, but for now just be aware that these symbols convey a certain meaning in a regular expression or pattern. You will need to be familiar with them before sitting your exam.

Generic Character Types

Regex offers a way for you to specify that a character in your search string may be any of a particular type.

You specify them using the backslash (escape) metacharacter and then providing the letter for the type.

The following table lists the character types that are available in PCRE:

Symbol	Character type
\d	Any decimal digit
\h	Any horizontal whitespace character
\s	Any white space character
\v	Any vertical white space character
\w	Any "word" character
\D	Any character that is not a decimal digit
\H	Any character that is not horizontal whitespace
\S	Any character that is not whitespace
\V	Any character that is not a vertical white space character
\W	Any "non-word" character

You should immediately spot that the capital symbol is the inverse of the lower case symbol.

A "word" character is any letter, digit, or the underscore character. The actual characters that are included in this are locale aware.

Boundaries

A word boundary is a position in the string where the current character and previous character do not both match \w or \W.

In other words it is a position in the string where a word starts or finishes, or a position where one of the characters matches \w and the other matches \W.

Symbol	Boundary
\b	Word boundary
\B	Not a word boundary
\A	Start of a subject
\Z	End of subject or newline at end
\z	End of subject
\G	First matching position in subject

Character Classes

Character classes are very flexible ways to define what set of characters in your search string can be matched.

By specifying a small sequence of characters in your pattern you are able to match a much larger set of characters in your search string.

We saw in the meta-characters table that we create a character class by putting it inside square brackets. An example of a character class is [A-Z] which stands for all of the letters in the upper case alphabet.

We can also use all of the generic types in character classes, so [A-Z\d] would match all of the uppercase letters as well as digits.

Matching more than once

The expression `/[A-Z\d]/` applied against the string "abc123ABCabc" will match the "1" character. In other words it matches the first occurrence in the search string of a character that matches the expression.

By using the "+" quantifier we can specify that we want one, or more, of the pattern. So the expression `/[A-Z\d]+/` applied against the string "abc123ABCabc" will match the "123ABC" characters.

We can use braces to limit the amount of matches. The syntax is best displayed in a table, where we match the expression against the string "abc123ABCabc"

Expression	Limit	Output
/[A-Z\d]+/	One or unlimited	123ABC
[A-Z\d]{3}	Exactly 3	123
[A-Z\d]{3,}	3 or more	123ABC
[A-Z\d]{3,5}	Between 3 and 5	123AB
[A-Z\d]{50}	Exactly 50	No match

Capturing groups

Capturing groups are delineated by brackets and allow you to apply a quantifier to the group.

They also produce numbered groups that store the value that was matched, and can be referenced elsewhere in your expression.

In this example we create a capturing group around the word "cheeseburger" and use the group to specify that zero or one of them will be matched.

```
<?php
$subject = "I can haz Cheeseburgers";
$pattern = "/I can haz (Cheeseburger)?/";
$matches = [];
preg_match($pattern, $subject, $matches);
var_dump($matches[0]);
```

This outputs `string(22) "I can haz Cheeseburger"`. Note that the "s" at the end of the string is not matched. As an exercise play with the regex in your favourite editor and see what happens if you use a subject "I can haz" (without a space at the end of the string).

To optimize your query you can use *non-capturing* groups. You should use these when you don't need to capture the match.

They're marked by placing a `?:` mark at the start of your group. The example above would be written as `/I can haz (?:Cheeseburger)?/`. Note that this expression will still return the string to PHP as above, but it just won't store the string Cheeseburger as a group for the expression to reference.

It may seem confusing that the `?` is a quantifier and also denotes a non-capturing group. Just remember that a quantifier cannot occur at the start of a group because there is nothing to quantify.

Greed and Laziness

By default matching is "greedy" and will match as much as possible of the string.

Lets consider an example that we'll work with. Lets imagine that we want to match HTML tags, so we try the following:

```
<?php
$subject = "Some <strong>html</strong> text";
$pattern = "/<.*>/";
$matches = [];
preg_match($pattern, $subject, $matches);
var_dump($matches[0]);
```

This outputs `string(21) "<strong>html</strong>"` which is clearly more than the html tag we were wanting.

It is greed that is to blame for this, the * quantifier is greedy and attempts to find the *longest* possible match. It returns the characters between the opening < of the strong tag and the last > of the closing tag, which is the longest possible match.

By contrast, a lazy search returns the *shortest* possible match. We can modify a quantifier to make it lazy by adding a `?` to it.

Changing the pattern in our example like this:

```
$pattern = "/<.*>/";
```

Will result in output like this: `string(8) "<strong>"`

There are a lot more options to modify quantifiers but they are outside of the scope of this manual.

Getting all matches

So far our expressions are returning just the first occurrence of the matching portion of a search string. Lets say that we want to find all of the matches in the string.

PCRE has a global modifier (more on those later), but PHP uses a separate function `preg_match_all()` to return all matches.

```
<?php
$subject = "Some <strong>html</strong> text";
$pattern = "/<.*?>/";
$matches = [];
preg_match_all($pattern, $subject, $matches);
var_dump($matches);
```

This outputs:

```
array(1) {
  [0] =>
  array(2) {
    [0] =>
    string(8) "<strong>"
    [1] =>
    string(9) "</strong>"
  }
}
```

Naming groups

You can name capturing groups by adding `?<name>` to the beginning of the brackets that opens the group. For example:

```
<?php
$subject = "test@example.com";
$pattern = "/^(?<username>\w+)@(?<domain>\w+).(?<tld>\w+)/";
$matches = [];
if (preg_match($pattern, $subject, $matches)) {
    var_dump($matches);
}
```

In this example we're naming the first part of the matching pattern as username, the next as domain, and the next as tld. This is a somewhat naïve example because it won't work for email addresses like test@example.co.uk but it does serve to show the syntax. The above example outputs:

```
array(7) {
  [0] =>
  string(16) "test@example.com"
  'username' =>
  string(4) "test"
  [1] =>
  string(4) "test"
  'domain' =>
  string(7) "example"
  [2] =>
  string(7) "example"
  'tld' =>
  string(3) "com"
  [3] =>
  string(3) "com"
}
```

So you are able to reference `$matches['username']` and receive "test" in response, which is convenient.

Pattern modifiers

You can add a modifier after the closing delimiter of an expression.

Modifier	Function
i	The expression is case-insensitive
m	Multi-line mode. Strings can span multiple lines and newline characters are ignored. Instead of matching the beginning and end of the string the ^ and $ symbols will match the beginning and end of the *line*
s	The . metacharacter will also match newlines
x	Ignore whitespace unless you escape it.
e	This causes PHP code to be evaluated and is highly discouraged. It is deprecated as of PHP 5.5
U	This makes the quantifiers lazy by default and using the ? after them instead marks them as greedy
u	This tells PHP to treat the pattern and string as being UTF-8 encoded. This means that characters instead of bytes are matched.

Arrays

PHP has a large number of array functions. The manual[26] lists them on a single page, and you should make sure that you study this page and each functions manual page. Rather than duplicating this information I'll focus on grouping and explaining some of these functions.

Declaring and referencing arrays

We will not dwell on what arrays are and will rather move straight onto the syntax used to declare arrays in PHP.

Indexes are created as a set of value-pairs that are separated by commas.

```php
<?php
// numeric index, auto assigned key
$arr = array(10, 'abc', 30);
// numeric index, key explicitly set
$arr = array(0 => 10, 1 => 'abc', 2 => 30 );
// associative
$arr = array('name' => 'foo', 'age' => 20);
// short syntax
$arr = ['name' => 'foo', 'age' => 20];
```

If you do not specify a key then PHP will assign an auto-incrementing numeric key. In the example the first two assignments are identical because PHP automatically assigns the key.

A key may be numeric or a string. An array may contain a mixture of numeric and string keys.

Arrays keyed on numbers are called *enumerative*. The first two examples above are enumerative. String keyed arrays are called *associative* arrays. The last two examples above are associative arrays.

> PHP 5.4 introduced a new short syntax to declare arrays. Instead of using the `array()` language construct you can use the [] operator

Arrays may be nested. In other words an array value can itself be an array. These are called multi-dimensional arrays.

An individual array element may be referenced using the [] operator like this:

```php
<?php
$arr = ['name' => 'foo', 'age' => 20];
echo $arr['age']; // 20
```

If you do not specify a key in the brackets PHP assumes that you are trying to reference a new element. You can use this to add an element to the end of an array:

[26] https://secure.php.net/manual/en/ref.array.php

```
<?php
$arr = [0 => 'id', 'name' => 'foo', 'age' => 20];
$arr[] = 'example';
print_r($arr);
```

This will output the following:

```
Array
(
    [0] => id
    [name] => foo
    [age] => 20
    [1] => example
)
```

Note that PHP chose the key by incrementing the highest numeric key in the array.

Quirks of PHP array keys

PHP arrays are zero based.

PHP array keys are case sensitive. `$arr['A']` and `$arr['a']` are different elements.

Keys may only be a string or an integer and other variable types are cast into one of these types before being stored.

Strings containing valid integers will be cast to the integer type. E.g. the key "8" will actually be stored under 8. On the other hand "08" will not be cast, as it isn't a valid decimal integer.

Floats are also cast to integers, which means that the fractional part will be truncated. E.g. the key 8.7 will actually be stored under 8.

Bools are cast to integers, too, i.e. the key `true` will actually be stored under 1 and the key `false` under 0.

Null will be cast to the empty string, i.e. the key null will actually be stored under "".

Arrays and objects cannot be used as keys. Doing so will result in a warning: Illegal offset type.

> If multiple elements in the array declaration use the same key, only the last one will be used as all others are overwritten.

Filling up arrays

You can use the `range()` function to add values to an array based on a range of values you specify. You specify the beginning, end, and step size for the range.

The PHP manual[27] has many useful examples, but here is one based on one of the comments:

```
<?php
print_r(array_combine(range(11, 20, 2),range(1,5)));
```

This will output:

```
Array
(
    [11] => 1
    [13] => 2
    [15] => 3
    [17] => 4
    [19] => 5
)
```

Another command `array_fill()` will let you fill up an array with a single value. It takes parameters for the starting index, how many values to fill, and the value to insert.

```
<?php
print_r(array_fill(10, 5, 'five'));
```

This outputs:

```
Array
(
    [10] => five
    [11] => five
    [12] => five
    [13] => five
    [14] => five
)
```

Related to this is the function `array_fill_keys()` which lets you specify which keys you want to fill with the value.

Push, pop, shift, and unshift (oh my!)

These four commands are used to add or remove elements from arrays.

Function	Effect
array_shift()	Shift an element off the beginning of array
array_unshift()	Prepend one or more elements to the beginning of an array
array_pop()	Pop the element off the end of array
array_push()	Push one or more elements onto the end of array

You'll probably notice that you can easily implement queues and stacks with these functions.

[27] https://secure.php.net/manual/en/function.range.php

The commands that remove an element from the array return it to you and shift all of the elements down. Numeric keys are reduced until they start counting from 0 and literal keys are left untouched.

The PHP Manual pages have examples, but this one from the array_shift[28] page shows exactly how the elements are shifted down:

```
<?php
$stack = array("orange", "banana", "apple", "raspberry");
$fruit = array_shift($stack);
print_r($stack);
```

In the output you will notice that "banana" now has the key of 0 where before it was 1:

```
Array
(
    [0] => banana
    [1] => apple
    [2] => raspberry
)
```

Comparing Arrays

It is possible to use the == and === operators to compare arrays. When applied to arrays the equality operator returns true if the arrays have the same keys and values, regardless of their type. The identity operator will only return true if the arrays have the same keys, values, in the same order, and of the same variable types.

```
<?php
$arr = ['1', '2', '3'];
$brr = [1, 2, 3];
var_dump($arr === $brr); // false
var_dump($arr == $brr);  // true
```

However, there are PHP functions devoted to array comparisons that make more sophisticated comparisons possible.

array_diff()

The first you'll need to know is `array_diff()` which takes a list of arrays as arguments. It will return an array containing the values from the first array that were not present in any of the other arrays.

This example examines a $net array against two other arrays.

[28] https://secure.php.net/manual/en/function.array-shift.php

```
<?php
$animals = ['dog', 'cat', 'cow'];
$birds = ['duck', 'chicken', 'goose'];
$net = ['dog', 'chicken', 'goose', 'hamster'];
print_r(array_diff($net, $animals, $birds));
```

This outputs an array with "hamster" in position 3. Note that you can use as many arrays in the list of parameters you want, and the index of the value in the return array is the same as it was in the original array.

`array_diff_assoc()` is an associative version of `array_diff()` and takes into account the array keys as well as their values. To see the difference we can use a very simple example:

```
<?php
$a = ['a' => 'apple', 'b' => 'banana'];
$b = ['a' => 'apple', 'd' => 'banana'];
print_r(array_diff($a, $b));
print_r(array_diff_assoc($a, $b));
```

The result of `array_diff()` is an empty array, but `array_diff_assoc()` returns an array consisting of `[b] => banana` because the key for the value "banana" is 'b' in the first array and 'd' in the second.

array_intersect()

The function `array_intersect()` also takes a list of arrays as parameters. It calculates which values from the first array are also present in all of the other arrays.

```
<?php
$birds = ['duck', 'chicken', 'goose'];
$net = ['dog', 'cat', 'chicken', 'goose', 'hamster'];
print_r(array_intersect($net, $birds));
```

This will output the elements that are in $net as well as in $birds:

```
Array
(
    [2] => chicken
    [3] => goose
)
```

Note that the keys are preserved.

`array_intersect_assoc()` includes an index check when matching elements. If we applied it to the arrays in the example above it would return an empty array. The return value is empty because although the values in the arrays match their index does not.

User defined matching functions

PHP provides functions that allow you to specify your own comparison function.

Consider `array_udiff()` as an example. It takes a list of array parameters followed by a callable as the last parameter.

Lets consider a trivial example, where we want to compare the lowercase value of the arrays to each other. More realistic use cases could involve more complicated operations, such as on objects for example.

```
<?php
$birds = ['duck', 'chicken', 'goose'];
$net = ['Dog', 'Cat', 'Chicken', 'Goose', 'Hamster'];
$diff = array_udiff($net, $birds, function($a, $b){
    $a = strtolower($a);
    $b = strtolower($b);
   echo $a . " - " . $b . PHP_EOL;
    if ($a < $b) {
        return -1;
    } elseif ($a > $b) {
        return 1;
    } else {
        return 0;
    }
});
print_r($diff);
```

Note the following:

- Since PHP 5.4 you can use closures as callables for any function that takes a callable as a parameter
- Since PHP 5.3 you can use lambdas as callables, also for any function that takes a callable as a parameter. In the example above we're using a lambda.
- The comparison function takes two arguments which will be the values to compare.
- From the manual[29]: The comparison function must return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second.

There are PHP functions to allow you to specify your own callable to compare keys, values, or both.

Quick list of comparison functions

This table shows the arrays for performing the difference between functions.

There are similar functions to perform the intersection. They have the same naming convention and parameters so I'm not listing them here.

[29] https://secure.php.net/manual/en/function.array-udiff.php

Function	Used for
array_diff	Computes the difference of arrays
array_diff_assoc	Computes the difference of arrays with additional index check
array_udiff	Computes the difference of arrays by using a callback function for data comparison
array_udiff_assoc	Computes the difference of arrays with additional index check, compares data by a callback function
array_udiff_uassoc	Computes the difference of arrays with additional index check, compares *data and indexes* by a callback function

Note that `array_udiff_uassoc()` takes two callable functions as parameters, one for the values and the last parameter for the indexes. Have a look at the manual page[30] and make sure you have studied all of its related functions.

Iterating through arrays

There are two ways to iterate through an array – by using a cursor and by looping through them.

Looping through arrays

An enumerative PHP array can be looped through by incrementing an index counter, but this won't work for associative arrays. A better and more robust approach is to use the `foreach()` construct.

Lets quickly look at two possible syntaxes that `foreach()` uses and then move on. You should already be familiar with its usage if you're considering sitting your exam so this is for the benefit of programmers from other languages.

```
<?php
$arr = [
    'a' => 'apple',
    'b' => 'banana',
    'c' => 'cherry'
];
foreach($arr as $value) {
    echo $value . PHP_EOL;
}
foreach($arr as $key => $value) {
    echo $key . ' = ' . $value . PHP_EOL;
}
```

The first `foreach()` loop will traverse the array and pass the array values into the code block. The second `foreach()` loop traverses it and passes the key and value.

Since PHP 5.5 the `list()` construct can be used in `foreach()` loops to unpack nested arrays. This is particularly useful when dealing with database results.

Here is an example from the manual[31]:

[30] https://secure.php.net/manual/en/function.array-udiff-uassoc.php

```php
<?php
$array = [
    [1, 2],
    [3, 4],
];
foreach ($array as list($a, $b)) {
    echo "A: $a; B: $b" . PHP_EOL;
}
```

Using array cursors

Every array has a cursor, or pointer, that points at the current element. A number of PHP functions use the cursor to determine which element to operate on.

Here are the basic cursor commands:

Function	Performs
reset	Moves the cursor to the beginning of the array
end	Moves the cursor to the end of the array
next	Advance the cursor
prev	Rewind the cursor
current	Returns the value of the element the cursor points at
key	Returns the key of the element the cursor points at

Objects can be iterated over using the same syntax, but it's important to know that they implement an interface iterator[32].

A less commonly seen use of a cursor is one such as this:

```php
<?php
$arr = [
    'a' => 'apple',
    'b' => 'banana',
    'c' => 'cherry'
];
while (list($var, $val) = each($arr)) {
    echo "$var is $val" . PHP_EOL;
}
```

`list()` is a language construct that assigns variables from a supplied array. The `each()` function returns the current key and value pair from an array and advance the array cursor.

[31] https://secure.php.net/manual/en/migration55.new-features.php
[32] https://secure.php.net/manual/en/class.iterator.php

Walking through arrays

The `array_walk()` function applies a user callable to every element in an array. It takes two parameters – a reference to the array and the callable.

The callable function will be passed two parameters, the first is the value of the element from the array and the second its index.

Some internal functions, such as `strtolower()` for example, will throw a warning if they receive too many parameters and so are not suitable as a callback for `array_walk()`.

If you need your callback function to alter the value of the array you should make sure that the first parameter is passed by reference.

Here is an example that will convert all the elements of an array to uppercase:

```
<?php
$arr = [
    'a' => 'apple',
    'b' => 'banana',
    'c' => 'cherry'
];
array_walk($arr, function(&$value, $key) {
    $value = strtoupper($value);
});
print_r($arr);
```

Note that I pass the value by reference into my lambda function, so changing it in the lambda will affect the `$arr` variable.

If we had used strtoupper() as a callback PHP would generate warnings.

Sorting Arrays

PHP offers several sort functions.

They follow a naming convention whereby the base “sort” function is prefixed with “r” for reverse and “a” for associative.

All of the sort functions take a *reference* to the array as their parameter and return a Boolean value indicating success or failure.

Function	Used for
sort	Sorting arrays alphabetically
rsort	Reverse alphabetical sort
asort	Associative sort
arsort	Reversed associative sort
ksort	Key sort
krsort	Reverse key sort
usort	User defined comparison function for sorting
shuffle	Pseudo-random sort

The associative sorts will sort by value and maintain the index association. Have a look at one of their manual pages for an example.

All of the functions (except usort) accept an optional parameter to indicate the sort flag. These flags are predefined constants:

Flag	Meaning
SORT_REGULAR	Compare items normally – don't change types
SORT_NUMERIC	Cast items to numeric values and then compare
SORT_STRING	Cast items to strings and then compare
SORT_LOCALE_STRING	Use locale settings to cast items to strings
SORT NATURAL	Use natural order sorting, like the function *`natsort()`*
SORT_FLAG_CASE	Can be combined with SORT_STRING and SORT_NATURAL to sort strings case-insensitively

Natural order sorting

Natural ordering is a sort order that makes sense to human beings. It is an alphabetic sort order, but multiple digits are treated as a single character.

The function *`natsort()`* does not take flags and is the same as called *`sort()`* with the SORT_NATURAL flag set.

As an example lets start with a string that looks sorted to our human eyes, shuffle it up, and then use both forms of sorting to see how it comes out:

```
<?php
$a = $b = explode(' ', 'a1 a2 a10 a11 a12 a20 a21');
shuffle($a);
shuffle($b);
natsort($a);
sort($b);
print_r($a);
print_r($b);
```

Note that I've used the explode function to break up a string into an array. This outputs:

```
Array
(
    [5] => a1
    [2] => a2
    [0] => a10
    [4] => a11
    [6] => a12
    [3] => a20
    [1] => a21
)
Array
(
    [0] => a1
    [1] => a10
    [2] => a11
    [3] => a12
    [4] => a2
    [5] => a20
    [6] => a21
)
```

Standard PHP Library (SPL) – ArrayObject class

The SPL[33] library includes the ArrayObject class that allows you to create objects from arrays. These objects can use the methods of the ArrayObject class, which are listed on the manual page[34].

This lets you work with arrays as objects, as in this example from the PHP manual:

```php
<?php
$fruits = array("d" => "lemon", "a" => "orange", "b" => "banana", "c" =>
"apple");
$fruitArrayObject = new ArrayObject($fruits);
$fruitArrayObject->ksort();
foreach ($fruitArrayObject as $key => $val) {
    echo "$key = $val\n";
}
```

When constructing an ArrayObject you pass in an input that can be either an array or an object.

You can also optionally specify flags:

[33] https://secure.php.net/manual/en/book.spl.php
[34] https://secure.php.net/manual/en/class.arrayobject.php

Flag	Effect
ArrayObject::STD_PROP_LIST	Properties of the object have their normal functionality when accessed as list (var_dump, foreach, etc.).
ArrayObject::ARRAY_AS_PROPS	Entries can be accessed as properties (read and write).

These flags can be set with the `setFlags()` method, as in this example from the manual[35]:

```
<?php
// Array of available fruits
$fruits = array("lemons" => 1, "oranges" => 4, "bananas" => 5, "apples" => 10);

$fruitsArrayObject = new ArrayObject($fruits);

// Try to use array key as property
var_dump($fruitsArrayObject->lemons);
// Set the flag so that the array keys can be used as properties of the
ArrayObject
$fruitsArrayObject->setFlags(ArrayObject::ARRAY_AS_PROPS);
// Try it again
var_dump($fruitsArrayObject->lemons);
```

This example will output:

```
NULL
int(1)
```

[35] https://secure.php.net/manual/en/arrayobject.setflags.php

Object Orientated PHP

Object Orientated code runs slower than procedural code but makes it easier to model and manipulate complex data structures. PHP has supported object orientated programming since version 3.0 and since then it's object model has been extended and reformed extensively.

This reference is not going to try to teach Object Orientated programming but will rather focus on the PHP implementation.

Declaring Classes and Instantiating Objects

Classes are declared using the `class` keyword.

```
<?php
class ExampleClass
{
    // class code
}
```

Classes can be named using the same rules as variables. Your coding standards will determine the case convention you use.

To instantiate an object from a class you use the `new` keyword.

```
<?php
$exampleObject = new ExampleClass();
```

Autoloading Classes

Classes should be defined before they are used, but you can make use of autoloading[36] to load classes when they are required. Together with coding standards like PSR4[37] that govern where PHP will look for a class this can be an indispensable feature.

You won't be asked questions about PSR4 in the Zend examination, but the standards put forward by the FIG group are very important in the PHP world and if you're new to PHP you should be aware of them.

Autoloading in PHP is accomplished the the `spl_autoload_register()` function[38]. A PSR4 compliant implementation is given on the PHP FIG group webpage, but lets have a look at a more simple demonstration from the PHP manual for an example:

[36] https://secure.php.net/manual/en/language.oop5.autoload.php
[37] http://www.php-fig.org/psr/psr-4/
[38] https://secure.php.net/manual/en/function.spl-autoload-register.php

```php
<?php
function my_autoloader($class) {
    include 'classes/' . $class . '.class.php';
}

spl_autoload_register('my_autoloader');

// Or, using an anonymous function as of PHP 5.3.0
spl_autoload_register(function ($class) {
    include 'classes/' . $class . '.class.php';
});
```

Using spl_autoload_register() lets you specify what function PHP will call if it is unable to load a class. You can include files in this function and so declare the class. If PHP is unable to find the class after this function has run then it will throw a fatal error.

Visibility or Access Modifiers

The visibility of a method or property can be set by prefixing the declaration with `public`, `protected`, or `private`.

Public class members can be accessed from anywhere.

Protected class members can be accessed from within the class and by its children.

Private class members can only be accessed from within the class itself.

If you don't explicitly specify a visibility then it will default to public.

Instance Properties and Methods

Properties

Class properties are declared by using one of the visibility modifiers followed by the name of the property. Property names follow the same naming rules as variables.

```php
<?php
class Properties
{
    public $email;
    protected $name = 'Alice';
    private $balance = 60 * 5;
}
```

Properties can be initialized to default values. They can be initialized with expressions, but not functions.

```php
<?php
class BrokenPropertyInit
{
    private $lastLogin = time();
}
```

This example won't run because you cannot initialize the class property using a function.

Methods

Methods are functions within a scope construct. They are declared in a function by using a visibility modifier followed by the function declaration. If you omit a visibility modifier the method will have public visibility.

```php
<?php
class MethodExample
{
    private $name;

    public function setName($name) {
        $this->name = $name;
    }
}
```

Methods can access non-static object properties using the `$this` pseudo-variable.

> The `$this` pseudo-variable is defined in objects and refers to the object itself. In static methods there is no object and so `$this` is not available.

Static Methods and Properties

Declaring a method or property as static makes it available without needing a concrete implementation of the class.

Because a static method can be called without an instantiated object the pseudo-variable `$this` is not accessible in these methods.

You should not call a non-static method statically. This is deprecated in PHP7 and discouraged in PHP5.

Referencing a static property or method is done using the scope resolution operator, which is a double-colon.

```php
<?php
class MyClass
{
    public static function sayHello() {
        echo "Hello World" . PHP_EOL;
    }

    public function someFunction() {
        self::sayHello();
    }
}
MyClass::sayHello(); // Hello World
$object = new MyClass();
$object->someFunction(); // Hello World
```

When we reference a static property from within the class we can use `self`, `parent`, or `static` to refer to it. We'll deal with the `static` keyword in the section on "Late Static Binding" in this chapter.

When referencing the static class member from outside the class you prefix the scope resolution operator with the name of the class. In the example above we referenced the static function with `MyClass::sayHello()`.

Working with Objects

Copying Objects

Objects are always passed by reference. If you want to create a copy of the object you must use the `clone()` keyword.

```php
<?php
$objectCopy = clone $originalObject;
```

PHP will create a shallow copy of the object.

In a shallow copy, if the source contains references to variables or other objects then those references are copied into the new object. This means that the original and cloned object share a reference to the same target object.

By contrast a deep copy creates new versions of referenced objects and inserts references to these into the cloned object. This is slower and more expensive because it involves creating a lot more objects. The cloned object will contain references to new copies of objects that the original references.

When cloning an object PHP will try to execute the `__clone()` method in the object. You can override this method to include your own behaviour for cloning an object. This method cannot be called directly.

Serializing Objects

Object serialization is accomplished with the `serialize()` and `unserialize()` functions. These functions support any type of PHP variable, except for resources.

When an object is serialized PHP will try to call the `__sleep()` method on it, and when it is unserialized the `__wakeup()` function is called.

Serializing an object gives a byte-stream representation of any value that can be stored in PHP. Resources cannot be serialized. Strings in PHP can contain byte-streams so you can place serialized objects into them.

The string will refer to the class of the object serialized and will contain all of the variables associated with it. References to anything outside of the object cannot be stored and will be lost, but circular references to anything inside the object will be retained.

When you unserialize the object PHP must have the class declared. If it does not have the class defined it will be unable to make an object of the correct type and will instead create one of type `__PHP_Incomplete_Class_Name` which has no methods.

```
<?php
$objectOriginal = new A;
$string = serialize($objectOriginal);
file_put_contents('serialize.txt', $string);
// in another PHP file
$string = file_get_contents('serialize.txt');
$objectCopy = unserialize($string);
```

Casting Objects to String

You can define how your object will be cast to string by declaring the `__toString()` method. PHP will call this method and return its result when it tries to cast your object to string.

```
<?php
class User
{
    private $firstName = 'Example';
    private $lastName = 'User';
    function __toString() {
        return $this->firstName;
    }
}
$user = new User;
echo $user; // Example
```

This lets you build and format a string that is meaningful for your object.

If you do not declare this method on your object then PHP will generate a catchable fatal error telling you that it cannot convert an object to a string.

Constructors and Destructors

A constructor is a method that is run when an object is instantiated from a class. Similarly, a destructor is made when the object is being unloaded.

They are declared as in this example:

```
<?php
class constructorExample
{
    public function __construct() {
        // called when instantiated
    }

    public function __destruct() {
        // called when unloaded
    }

    public function constructorExample() {
        // PHP4 style constructor
    }
```

In PHP4 constructor methods were identified by having the same name as the class they were defined in. In the example above the function `constructorExample()` would be a constructor in PHP4.

For backwards compatibility PHP5 will search for a function with the same name as the class if it cannot find a `__construct()` function. This functionality is deprecated in PHP7 and will be removed in future PHP versions.

Constructor parameters

If a class constructor takes a parameter you need to pass it in when instantiating an instance of the class.

```
<?php
class User {
    public function __construct($name) {
        $this->name = $name;
    }
}

$user = new User('Alice');
```

In this example we would pass the string "Alice" to the constructor function.

Inheritance

PHP supports inheritance in its object model. If you extend a class then the child class will inherit all of the public properties and methods of the parent class. You can override them in the child class, but they will otherwise have the same functionality.

PHP does not support inheriting from more than one class at a time.

The syntax to cause a class to inherit is very simple. When declaring the class we simply indicate the name of the class it is extending, as in this example:

```php
<?php

class ParentClass
{
    public function sayHello() {
        echo __CLASS__;
    }
}

class ChildClass extends ParentClass
{
    // nothing in this class
}

$kid = new ChildClass;
$kid->sayHello(); // ParentClass
```

In this example the ChildClass is declared as extending the ParentClass. It inherits the `sayHello()` method.

The magic constant `__CLASS__` gives the name of the class that is currently being executed. Note that we're calling the inherited method in the child class, but it is executing the function in the parent class and so reporting that the class name is ParentClass.

The "Final" Keyword

PHP5 introduced the "`final`" keyword. You can apply it either to a class as a whole, or to specific methods within a class. The effect of the `final` keyword is to prevent classes from being extended or methods from being overridden.
The visibility of all final properties and methods is public.

PHP will issue a fatal error if you try to override a final method in a child class or if you try to declare a class that extends a class that is marked final.

> This is somewhat different from the use of final in Java, the PHP equivalent of the Java final keyword is "`const`".

Overriding

A child class may declare a method with the same name as the parent class, providing that the method is not marked final in the parent.

If a function is overridden like this and called on the child then the parent's class will not be called.

This applies to constructors and destructors, but in these cases this is quite often worked around like this:

```
<?php
class ChildClass extends ParentClass
{
    public function __construct() {
        parent::__construct();
        // more constructor functions here
    }
}
```

The call to `parent::__construct()` will call the constructor method of the parent class. When control flow returns to the child the remaining functions in its constructor will be called.

If a child overrides a method from a parent class then the child's class cannot have a lower visibility than the parent class. In other words if the parents method is public then the child cannot override the method as being protected or private.

Interfaces

Interfaces allow you to specify what methods a class must implement without specifying the details of the implementation.

They are commonly used to define a *contract* in the service-orientated architecture paradigm, but can also be used whenever you want to stipulate how future classes are expected to interact with your code.

All methods in an interface must be declared as public and may not have any implementation themselves.

Interfaces are declared as in this example:

```
<?php
interface PaymentProvider
{
    public function showPaymentPage();
    public function contactGateway();
    public function notify();
}
```

A class would be declared as implementing it like this:

```php
<?php
class CreditCard implements PaymentProvider
{
    public function showPaymentPage() {
        // implementation
    }

    public function contactGateway() {
        // implementation
    }

    public function notify() {
        // implementation
    }
}
```

Classes may implement more than one interface at a time by listing the names of the interfaces separated by commas.

> Classes may inherit from only one class but may implement many interfaces.

Exceptions

Exceptions offer a unified approach to handling errors. Exceptions in PHP are similar to those in other languages.

PHP includes a number of standard exception types[39], and the standard PHP library (SPL) includes several more[40]. Although you don't have to use these exceptions doing so allows you the chance to use more fine-grained error detection and reporting.

Let's have a look at the syntax and then discuss Exceptions in more detail:

[39] https://secure.php.net/manual/en/reserved.exceptions.php
[40] https://secure.php.net/manual/en/spl.exceptions.php

```php
<?php
class ParentException extends Exception {}
class ChildException extends ParentException {}

try {
    // some code
    throw new ChildException('My Message');
} catch (ParentException $e) {
    // matches this class because of inheritance
    echo "Parent Exception :" . $e->getMessage();
} catch (ChildException $e) {
    // matches this class exactly
    echo "Child Exception :" . $e->getMessage();
} catch (Exception $e) {
    // matches this class because of inheritance
    echo "Exception :" . $e->getMessage();
}
```

The output of this example is “Parent Exception :My Message”

Extending Exception classes

The base PHP Exception class can be extended to create custom exceptions. Only a class that inherits from (or is) the base Exception class can be used with the `throw` keyword.

We are throwing a ChildException which inherits from ParentException which in turn extends the base Exception class.

A `try` block must have at least one `catch` block but can have multiple blocks. The blocks are evaluated in order from top to bottom.

The class of the exception being thrown is matched against the name of the class given as a parameter to the catch clause. If the classes match then the code in the block is executed, otherwise the next catch statement is evaluated.

The matching criteria are that the classes are either exactly the same or the thrown exception inherits from the exception in the catch statement.

In our example we threw an exception of the ChildException which inherits from the ParentException. The exception is therefore matched against the first catch block and the code is executed.

> I put the base Exception at the bottom of the list of catch blocks because all custom exceptions inherit from it, which makes it a catchall.

Finally

PHP 5.5 introduced the `finally` keyword which is used to denote a code block that will always be executed, regardless of whether an exception is caught or not.

```
<?php
try {
    // some code
} catch (Exception $e) {
    // error reporting
} finally {
    // always do this
}
```

Reflection

The PHP reflection API allows you the ability to inspect PHP elements at runtime and retrieve information about them.

The Reflection API was introduced with PHP5.0 and since PHP 5.3 has been enabled by default.

One of the common places that reflection is used is in unit testing. One example of where Reflection is useful is in testing the value of a private property in a class. You can use reflection to make the private property accessible and then make assertions.

There are several reflection classes that allow you to inspect different types of variables. Each of these classes is named for the type of variable you can use it to inspect.

Class	Used to inspect
ReflectionClass	Classes
ReflectionObject	Objects
ReflectionMethod	Methods of objects
ReflectionFunction	Functions like PHP or user functions
ReflectionProperty	Properties

The PHP manual[41] has exhaustive documentation on these classes and their methods.

Let's briefly look at an example of using ReflectionClass.

```
<?php
$reflectionObject = new ReflectionClass('Exception');
print_r($reflectionObject->getMethods());
```

The parameter passed to the contructor of the reflection class is either the string name of the class, or a concrete instantiation (object) of the class.

[41] https://secure.php.net/manual/en/book.reflection.php

The ReflectionClass object has a number of methods that allow you to retrieve information about the inspected class. In the example above we are outputting an array of all of the methods that the Exception class has.

Type Hinting

Type hinting allows you to specify the variable type that a parameter to a function is expected to be.

In the following example we specify that the parameter `$arr` being passed to the `printArray()` function must be an array.

```
<?php
function printArray(array $arr) {
    echo "<pre>" . print_r($arr,true) . "</pre>";
}
```

In PHP5 if you pass a parameter of the wrong type then a recoverable fatal error will be generated. In PHP7 a TypeError exception is thrown.

> As of PHP7 type hinting is being referred to as "type declarations". I'm going to use this new nomenclature but the terms are interchangeable within the context of PHP at the moment.

In PHP5 you can only specify composite types and callables (PHP5.4+) as type hints. PHP7 includes the ability to specify scalar variable types.

Additionally, the NULL type hint is allowed to be used if NULL is used as the default parameter for a function.

```
<?php
function nullExample(null $msg = null) {
    echo $msg;
}
```

If you specify a class as a type hint then all of its children and implementations will be valid parameters.

Class Constants

A constant is a value that is immutable. Class constants allow you to define such values on a per-class basis.

Class constants follow the same naming rules as variables but do not have a `$` symbol prefixing them. By convention constant names are declared in uppercase.

Let's consider an example:

```php
<?php
class MathsHelper
{
    const PI = 4;

    public function squareTheCircle($radius) {
        return $radius * (self::PI ** 2);
    }
}

echo MathsHelper::PI; // 4
```

Class constants are public and so are accessible from all scopes. We use the scope resolution operator and the name of the class it is declared in when we access it from outside the class.

Class constants, like traditional constants, may only contain scalar values.

Late Static Binding

Late static binding was introduced in PHP 5.3.0 and is a method to reference the called class (as opposed to the calling class) in the context of static inheritance.

The idea was to introduce a keyword that would reference the class that was initially called at runtime, rather than the class that the method was defined in.

Rather than introduce a new reserved word the decision was made to use the `static` keyword.

Forwarding calls

A "forwarding" call is a static call that is introduced by `parent::`, `static::`, or one called by the function `forward_static_call()`.

A call to `self::` can also be a forwarding call if the class falls back to an inherited class because it does not have the method defined.

Late static binding works by storing the class in the last "non-forwarding call". In other words late static bindings' resolution will stop at a *fully resolved* static call.

I am going to take a detailed walk through a modified example of the PHP manual example[42].

[42] https://secure.php.net/manual/en/language.oop5.late-static-bindings.php

```php
<?php
class A {
    public static function foo() {
        echo static::who();
    }

    public static function who() {
        return 'A';
    }
}

class B extends A {
    public static function test() {
        A::foo();
        parent::foo();
        self::foo();
    }
}

class C extends B {
    public static function who() {
        echo 'C';
    }
}

C::test(); // ACC
```

The output of ACC might be counter-intuitive at first but lets step through it slowly.

The call to `C::test()` is fully resolved and so class C is initially stored as the last non-forwarding call.

There is no `test()` method in the function C so the call is forwarded implicitly to its parent. So the `test()` method in class B is being called.

The call to A::foo()
The first call in `test()` specifically names class A as the scope. This means that the call is fully resolved.

So the class being stored as the last *non-fowarding* call is changed to be A.

The `foo()` method in A is called and the static keyword is resolved to find which class to call the `who()` method on.

The last non-fowarding call was to a class in A and consequently the `who()` method in class A is called.

The call to parent::foo()
The next call in `test()` refers to the parent of B so the call is being explicitly forwarded to the parent of B, which is A.

This is a forwarded call so the stored value stored as the last fully resolved static call (which is C) is left unaltered.

The `foo()` method in A is called and the static keyword is resolved to find which class to call the `who()` method on.

The last non-fowarding call was to a class in C and consequently the `who()` method in class C is called.

The call to self::foo()
Class B does not have the `foo()` method defined and so the call is implicitly passed to the parent, class A.

This is a forwarded call so the stored value stored as the last fully resolved static call (which is C) is left unaltered.

This results in the `who()` method of class C being called when the static keyword is resolved in class A.

Magic (_*) Methods

PHP treats any method with a name prefixed by two underscores as magical. There are fifteen predefined magical functions and it is recommended to avoid naming other functions with the double underscore prefix.

__get and __set

These magic methods are called when PHP tries to read (get) or write (set) inaccessible properties.

```
<?php
class BankBalance {
    private $balance;

    public function __get($propertyName) {
        echo "No property " . $propertyName;
    }

    public function __set($propertyName, $value) {
        echo "Cannot set $propertyName to $value";
    }
}
$myAccount = new BankBalance();
$myAccount->balance = 100;
echo $myAccount->nonExistingProperty;
```

The `__get()` method is passed the name of the property that was being looked for.

An additional parameter, the $value, is passed to `__set()`.

__isset and __unset

The `__isset()` method is triggered by calling the `isset()` function or `empty()` on an inaccessible property.

The `__unset()` method is triggered by calling the unset() function on an inaccessible property.

Both methods accept a string parameter that contains the name of the property that was being passed as a parameter to the function.

You can use these magic methods to allow the `isset()`, `empty()`, and `unset()` functions to work on private and protected properties.

__call and __callStatic

These magic methods are called if you try to call a non-existing method on an object. The only difference is that `__callStatic()` responds to static calls while `__call()` responds to non-static calls.

```
<?php
class Politician {
    public function __call($method, $arguments) {
        echo __CLASS__ . ' has no ' . $method . ' method';
    }
}

$jacob = new Politician();
$jacob->honesty();  // Politician has no honesty method
```

In both cases the magic method is passed a string containing the name of the method that the call is trying to find, and an array of the arguments that were passed.

__invoke

The magic method `__invoke()` is called when you try to execute an object as a function.

```
<?php
class Square
{
    public function __invoke($var) {
        return $var ** 2;
    }
}
$callableObject = new Square;
echo $callableObject(10); // 100
```

This syntax may be confused with variable function names so watch out for that.

__debugInfo

This magic method is called by `var_dump()` when dumping the object to determine what properties should be output.

By default `var_dump()` will output all public, protected, and private properties of the object.

```
<?php
class Dictatorship {
    private $wmd = 'Nuke';
    public $oil = 'Lots';

    public function __debugInfo() {
        return [
            'oil' => $this->oil
        ];
    }
}

$country = new Dictatorship();
var_dump($country);
```

This example will prevent the `$wmd` variable from being included in the var_dump.

More magic functions

We have dealt with `__construct()` and `__destruct()` functions in the section on "Constructors and Destructors".

We have dealt with `__sleep()` and `__wake()` in the section on "Serializing Objects".

We looked at `__clone()` when discussing "Copying Objects" and `__toString()` in the section named "Casting Objects to String".

Standard PHP Library (SPL)

The standard PHP library is a collection of classes and interfaces that are recipes for solving common programming problems. It is available and compiled in PHP from version 5.0.0.

The classes fall into categories. For a complete list of the classes refer to the PHP Manual[43].

[43] https://secure.php.net/manual/en/book.spl.php

Category	Used for
Datastructures	Standard data structures, like linked lists, doubly linked lists, queues, stacks, heaps, etc
Iterators	
Exceptions	
File Handling	
ArrayObject	Accessing object with array functions
SplObserver and SplSubject	Implementing the observer pattern

There are a number of functions that spl provides. They mostly fall into broad reflection and autoloading categories.

Generators

Generators provide you with an easy way to create iterator objects.

The advantage to using an iterator with a generator is that you can build an object that you can traverse over without needing to calculate the entire data set. This saves processing time and memory.

The use case could be to replace a function that normally returns an array. The function would calculate all of the values, allocating an array variable to store them, and return the array.

A generator only calculates and stores one value, and yield it to the iterator. When the iterator requires the next value it calls the generator. When the generator runs out of values it simply exits without yielding a value.

A generator cannot return a value. An empty `return` statement can be used and this will terminate the generator.

Yield keyword

The yield keyword is similar to a function return, except that it is used to yield a value back to the iterator while pausing execution of the generator.

The scope of the generator is maintained between calls. Variables will not lose their value after the generator yields.

If you use the yield keyword as part of an expression in PHP5 you must surround the yield statement with parentheses.

```
<?php
function exampleGenerator() {
    // some functions
    $data = (yield $value); // this is valid
    $data = yield $value; // this is not valid
}
```

This requirement is removed in PHP7.

Yielding with keys

It is possible to yield key-value pairs that perform as associative arrays for functions using the generator.

If you don't explicitly yield with keys then PHP will pair yielded values with increasing sequential keys, just as for a enumerative array.

The syntax to yield a key-value pair is very similar to declaring associative arrays:

```
<?php
function myGenerator() {
    // some functions
    yield $key => $value;
}
```

Yielding NULL

Calling `yield` without an argument causes it to yield a NULL value with an automatic increasing sequential key.

Yielding by Reference

Generator functions can yield variables by reference and the syntax to do so is to prepend an ampersand to the function name.

```
<?php
function &referenceGenerator() {
    // some functions
    yield $value;
}
```

Traits

Traits were introduced in PHP 5.4.0 and are designed to help alleviate some of the limitations of a single inheritance language.

A trait contains a set of methods and properties just like a class, but cannot be instantiated by itself. Instead the trait is included into a class and the class can then use its methods and properties as if they were declared in the class itself.

In other words traits are flattened into a class and it doesn't matter if a method is defined in the trait or in the class that uses the trait. You could copy and paste the code from the trait into the class and it would be used in the same manner.

The code that is included into a trait is intended to encapsulate reusable properties and methods that can be applied to multiple classes.

Declaring and Using Traits

We use the `trait` keyword to declare a trait and to include it in a class we employ the `use` keyword. A class may use multiple traits.

```php
<?php
trait Singleton
{
    private static $instance;

    public static function getInstance() {
        if (!(self::$instance instanceof self)) {
            self::$instance = new self;
        }
        return self::$instance;
    }
}

class UsingTraitExample
{
    use Singleton;
}

$object = UsingTraitExample::getInstance();
var_dump($object instanceof UsingTraitExample); // true

exit;
```

In the example above we declare a trait that includes the methods and properties needed to implement the Singleton pattern.

When we want to make a new class follow the Singleton pattern we can do so just by using the trait. We don't have to implement the pattern within the class, and don't have to include the pattern in the inheritance hierarchy.

Namespacing Traits

PHP will generate a fatal error if traits have conflicting names, but traits
may be defined in namespaces.

If you are trying to use the trait in a class that is not in the same namespace hierarchy then you will need to specify the fully-qualified name when you include it.

Inheritance and Precedence

Traits may not extend other traits or classes, but you can simply use a trait inside another.

Methods declared in a class using a trait take precedence over methods declared in the trait. However, methods in a trait will override methods inherited by a class.

Expressed more simply, precedence in traits and classes is as follows:

Conflict Resolution

PHP will generate a fatal error if two traits attempt to insert a method with the same name unless you explicitly resolve the conflict.

PHP allows you to use the `insteadof` operator to specify which of the conflicting methods you want it to use.

This lets you exclude one of the trait methods, but if you want to keep both methods you need to use the `as` operator. The `as` operator allows you to include one of the conflicting methods, but use a different name to reference it.

Here is a rather long example that shows this usage:

```php
<?php
trait Dog {
    public function makeNoise() {
        echo "Woof";
    }

    public function wantWalkies() {
        echo "Yes please!";
    }
}

trait Cat {
    public function makeNoise() {
        echo "Purr";
    }

    public function wantWalkies() {
        echo "No thanks!";
    }
}

class DomesticPet
{
    use Dog, Cat {
        Cat::makeNoise insteadof Dog;
        Cat::wantWalkies as kittyWalk;
        Dog::wantWalkies insteadof Cat;
    }
}

$obj = new DomesticPet();
$obj->makeNoise();  // Purr
$obj->wantWalkies(); // Yes please!
$obj->kittyWalk(); // No thanks!
```

It is not enough to use `as` by itself. You still need to use `insteadof` to exclude the method you don't want to use, and can only then use `as` to make a new way to reference the old method.

Visibility

You can apply a visibility modifier to functions by extending the `use` keyword, as in this example:

```php
<?php
trait Example {
    public function myFunction() {
        // do stuff
    }
}

class VisbilityExample {
    use Example {
        myFunction as protected;
    }
}

$obj = new VisbilityExample();
$obj->myFunction(); // PHP Fatal error:  Call to protected method
```

We specify that the method should be made protected in the class, even though it is declared as public in the trait. You can include multiple functions in the block, each of which may have it's own visibility.

Security

Security is a major concern for web applications. Even major organizations such as the United Nations have been hacked using very simple security flaws.

I'm of the opinion that there is no such thing as a completely secure system. My aim when securing an application is twofold. Firstly, I aim to make it take as long as possible for an attacker to gain access. My next aim is to minimize the value of any information they can retrieve.

This reduces the feasibility of hacking my application for a hacker – it will take a long time to get in, and when they do they will need to expend considerable effort to get any valuable information.

When you are being chased by a tiger you don't need to run faster than the tiger. You just need to run faster than the chap next to you.

One of the major flaws in security is social engineering. A discussion on social engineering is not in scope for a PHP manual, but you must always remember that it is not just your code and servers that are entry points to your data.

Configuration

Some of the best advice in configuring PHP is to make sure that you keep up to date with the releases and make use of the improvements they bring.

There are valid use cases where you need to remain on a particular version of PHP, for example compliance with a software auditing process in the context of a financial institution.

You should have a very strong reason if you are not using the most current stable release of PHP in favour of an older version.

Errors and warnings

You should configure PHP to hide warnings and errors while in production. Errors and warnings can give a person a clue about the internal workings of your code such as directory names and what libraries you are using.

Setting	Value
display_errors	Off
log_errors	On
error_reporting	E_ALL & ~E_DEPRECATED & ~E_STRICT

In development your error_reporting setting should be E_ALL and your code must run without warnings - don't use deprecated functions.

Disabling functions and classes

You can use the `disable_functions` and `disable_classes` directives to prevent functions and classes from being used.

Common functions to disable include those which allow PHP to execute system commands: `exec`, `passthru`, `shell_exec`, `system`

The `DirectoryIterator` and `Directory` classes are also commonly disabled as these can also be used by an attacker.

PHP as an Apache module

If PHP is running as an Apache module it will be run using the same user as the Apache server. This means that it will have the same permissions and access as the Apache user.

It is best practice to set up a user for Apache rather than run it as "nobody". The Apache user should have limited access to the file system, and should definitely not be on the sudoers list.

You should use the PHP `open_basedir`[44] setting to limit what directories PHP is able to be run in.

PHP as a CGI binary

If you're running PHP as a CGI binary, then you need to be aware of a number of issues. PHP does attempt to mitigate some of these attacks[45] using default configuration settings, and you should know what these are.

Usually the query information in a URL after the question mark is passed as command line arguments to the interpreter by the CGI interface. So this url http://my.host/cgi-bin/php?/etc/passwd would attempt to pass /etc/passwd to the PHP binary. Usually CGI interpreters open and execute the file specified as the first argument on the command line. PHP refuses to interpret the command line arguments when invoked as a CGI binary.

The directives `cgi.force_redirect`, `doc_root` and `user_dir` are used to prevent access to private documents by PHP.

Setting `cgi.force_redirect` blocks PHP from being able to be called directly from a URL – it will only execute if it is being called on a redirect from a webserver like Apache.

When working with PHP as a CGI binary you should consider moving the php binary outside of the document tree and separating your executable PHP scripts from your static scripts.

Session Security

The two areas of focus that you need to be aware of are "session hijacking" and "session fixation". You should study the PHP manual[46] page on session security in addition to this document.

Session Hijacking

HTTP is a stateless protocol and a web server can be expected to be serving multiple different visitors at the same time.

The server needs to be able to tell clients apart and does so by assigning each client a session identifier. The session identifier can be retrieved by calling `session_id()`. It is created after the `session_start()` function is called.

When the client makes subsequent requests to the server they provide the session identifier that then lets the server associate the request with a session.

---

[44] https://secure.php.net/manual/en/ini.core.php#ini.open-basedir
[45] https://secure.php.net/manual/en/security.cgi-bin.attacks.php
[46] https://secure.php.net/manual/en/session.security.php

It should be apparent to you that if you are able to present somebody else's session identifier to the server then you will be able to masquerade as that user.

Obtaining the session identifier of another user can be accomplished in a number of ways.

Firstly, if the session identifier follows a predictable pattern then the attacker could try and determine what it will be for a user. PHP uses a very random way to generate session identifiers so you don't need to worry about this.

Next, by inspecting network traffic between the client and the server the attacker could read the session identifier. You can set `session.cookie_secure=On` to make session cookies only available over HTTPS to mitigate this.

Attacks made against the client, such as an XSS attack or Trojan program running on their computer could also reveal the session identifier. This can be partially mitigated by setting the `session.cookie_httponly` directive on.

Session fixation

Session fixation exploits a weakness in the web application. Some applications do not generate a new session id for a user when authenticating them. Instead they allow an existing session id to be used.

The attack occurs when an opponent creates a session on the webserver. They know the session id for this session. They then trick a user into using this session and authenticating themselves. The attacker is then able to use the known session id and has the privileges of the authenticated user.

There are several ways to set the session id and the actual method used will depend on how the application accepts the identifier.

The most simple way to do it would be to pass the session identifier in the URL, like this http://example.org/index.php?PHPSESSID=1234.

The best way to mitigate the risk of session fixation is to call the function `session_regenerate_id()` every time the privilege level changes, for example after logging in.

You can set `session.use_strict_mode=On` in your config file. This setting will force PHP to only use session identifiers that it creates itself. It will reject a user supplied session identifier. This will mitigate attempts to manipulate the cookie.

The settings `session.use_cookies=On` and `session.use_only_cookies=On` will prevent PHP from accepting the session identifier from the URL.

Improving Session Security

In addition to the mitigation strategies that I have already mentioned you should also do the following:

- Check that the IP address remains the same between calls. This is not always feasible for mobile phones which move between towers and so change connections.
- Use short session timeout values to reduce the window for fixation
- Provide a means for users to logout that calls `session_destroy()`

Cross-Site Scripting

Cross site scripting (XSS) attacks are an attack where malicious code is injected onto an otherwise benign site. Usually malicious browser-side code like Javascript is placed onto the website to be downloaded and run by clients.

The attack is effective because the client thinks that the code originated from the website that it trusts. The code is able to access session identifiers, cookies, html storage data, and other information related to the site.

There are a few broad types of XSS attacks: stored, reflected, and DOM

In a stored XSS attack the opponent is able to place input into a stored location on the server. Examples could be in user comments displayed on the site and stored in the database. When the site outputs the list of user comments to another visitor they would receive the malicious code.

In a reflected XSS attack the opponent is able to get the website to output something directly. The most common form of this attack is a form fill error that prefills the input fields with the previously submitted fields, or outputs the erroneous field value. By sending the visitor to a crafted URL that includes malicious code as an error message (for example) the attacker is able to trick the client into executing it within the context of the trusted site.

A DOM attack is one that rests entirely within the page. The malicious code is read from an element in the page and the call to the code is made within the page itself.

Furthermore, XSS attacks can be classed either as server side or client side attacks. A server side attack is one where the server delivers the malicious code. Client XSS occurs when untrusted user supplied data is used to update the DOM with an unsafe JavaScript call.

Mitigating XSS attacks

The most important rule to follow is never to allow unescaped data to be output to the client. Always filter data and strip out harmful tags before allowing it to be sent to the client.

Three useful functions for this are `htmlspecialchars()`, `htmlentities()`, and `strip_tags()`.

The most safe method to escape output before displaying it is to use `filter_var($string, FILTER_SANITIZE_STRING)`.

Because of the wide variety of formats that can be used in URL's and HTML to output data it is not safe to blacklist codes. You should rather whitelist the specific tags that you want to allow. Take a look at the OWASP filter evasion cheat sheet[47] to see just how many ways there are to evade a blacklist.

You also need to mitigate XSS in your Javascript, but this manual is about PHP.

Cross-Site Request Forgeries

CSRF attacks exploit the trust that a website has in a client. In these attacks the opponent tricks the client into executing a command on a website that trusts that client.

The most common form would be to send a POST request to a form input.

[47] https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet

Imagine that Alice is logged onto her bank website that has a form that allows her to transfer money to another account. Chuck knows the endpoint of that form and what input fields it has. He somehow manages to trick Alice's web-browser into sending a POST request to that form instructing the bank to transfer money into his account. The bank trusts Alice's web browser because it has a valid session and performs the request.

There are many ways for Chuck to trick Alice's web browser, including using iframes and Javascript.

To mitigate these requests you should generate a unique and very random token which you store in Alice's session. When you output the form you include this token so that when Alice submits the form she also submits the token. Before you process the form you check that the submitted token matches the token stored in her session.

Chuck has no way of knowing what token is in Alice's session and so won't be able to include it in his POST. Your code will reject the request that he tricked Alice into making because it doesn't have a valid token.

Actual banks often require a person to re-authenticate when performing a sensitive operation, and will often require two-factor authentication as part of this process.

SQL Injection

SQL injection is the most common form of attack on the web, and also one of the easiest to defend against. SQL injection occurs when the attacker is able to insert malicious commands into a SQL statement for execution by the database.

Many database setups allow the database to write files to disk. This feature allows hackers to create a backdoor by using the database to write PHP scripts to a directory where the web server will serve it.

This means that the effect of SQL injection is not limited to having your database compromised, but could lead to the attacker being able to execute arbitrary code on your database.

At it's heart the problem with SQL injection comes from the fact that a SQL statement has a mix of data and syntax. By allowing user supplied data to be incorporated with function syntax we create the possibility that malicious data can interfere with the syntax.

The most effective way to start to mitigate SQL injection in the PHP language is to exclusively use prepared statements[48] to interact with your database. This will help exclude the majority of SQL injection attacks, but is not sufficient by itself to be fool proof.

Additional improvements to your PDO security will be to make sure that you're using an up-to-date version of MySQL and enforcing the use of a character set in the client DSN. There is a very subtle way to use mismatching character sets in certain vulnerable encoding schemes to deploy a SQL injection, see the second answer (not the accepted one) on this StackOverflow article for an exposition[49].

A less effective way to mitigate SQL injection is to escape special characters before sending them to the database. This is more prone to error than using prepared statements.

If you are going to try escaping special characters you must use the database specific function (e.g.: `mysqli_real_escape_string()`) or `PDO::quote()` and not a generic function like `addslashes()`.

[48] https://secure.php.net/manual/en/pdo.prepared-statements.php
[49] https://stackoverflow.com/questions/134099/are-pdo-prepared-statements-sufficient-to-prevent-sql-injection

You should also always connect to the database with a user that has the least amount of privileges that are required for the application to function. Never allow your web application to connect to the database as its root user.

Remote Code Injection

Remote code injection is an attack where an opponent is able to get the server to include and execute their code.

Malicious system calls

Certain functions like `eval()`, `exec()`, and `system()` are susceptible to remote code injection exploits. If you are executing a variable that includes user input they will be able to inject commands using escape characters.

You can mitigate this by using `escapeshellargs()` to escape the arguments passed to the shell command. The function `escapeshellcmd()` will escape the shell command itself.

If you're not explicitly using these functions you should disable them in your php.ini

preg_replace and /e

PHP supports the use of the "e" modifier on `preg_replace()`. Using it will cause PHP to execute the result of the operation as PHP code.

If an attacker is able to engineer situation where they can have their input placed into a `preg_replace()` function they will be able to have code executed by the server.

The solution is to use the function `preg_quote()` which will escape harmful characters. This also has the side-effect of eliminating regex, so you might need to escape your code manually.

Gaming include and require

Both `include()` and `require()` allow the possibility of including files specified by url if the PHP configuration setting `allow_url_fopen` is on.

The most common occurrence of this is when people use a GET variable in the url to determine some dynamic content to include. This is very much a rookie mistake.

For example a site could have a URL like: http://example.com/index.php?sidebar=welcome and then dynamically include the welcome.php file into the sidebar.

An opponent could provide a url instead of the "welcome" string and have their own code executed on the server with the same privilege level as the webserver user.

To counter this sort of problem you can turn `allow_url_fopen` to Off, use `basename()` against the variable you are including so that paths are removed, and only include against a whitelist.

```php
<?php
$page = $_GET['page'];
$allowedPages = array('adverts','contacts','information');
if ( in_array($page, $allowedPages) ) {
    include basename($page . '.html');
}
```

Email Injection

When setting up your mail server you must make sure it is not configured as an open relay that allows anybody on the Internet to use it to send mail. You should also consider closing port 25 (SMTP) on your firewall so that outside hosts are unable to reach your server.

It is possible for a user to supply hexadecimal control characters that allow them to change the message body or recipient list.

For example, if your form allows the person to enter their email address as a "from" field for the email then the following string will cause additional recipients to be included as cc and blind carbon copy recipients of the message:

```
sender@example.com%0ACc:target@email.com%0ABcc:anotherperson@emailexample.com,stranger@shouldhavefiltered.com
```

It is also possible for the attacker to provide their own body, and even to change the MIME type of the message being sent. This means that your form could be used by spammers to send mail from.

You can protect against this in a couple of ways.

Make sure that you properly filter input that you use when sending mails.

```php
<?php
$from = $_POST["sender"];
$from = urldecode($from);
if (preg_match("(\r|\n)", $from)) {
    die("Invalid email field");
}
```

You could also install and use the Suhosin PHP extension. It provides the `suhosin.mail.protect` directive that will guard against this.

You could implement a tarpit to slow bots down or trap them indefinitely. Take a look at msigley/PHP-HTTP-Tarpit on Github[50] as an example of a tarpit.

Filter Input

When approaching security it is best to plan for the worst-case scenario and assume that all input is tainted, and that all user behaviour is malicious. You should only use input that you've manually confirmed to be safe.

[50] https://github.com/msigley/PHP-HTTP-Tarpit

It is possible for input to be in a format that will be ignored by a filter and then parsed by the browser. The XSS evasion cheat sheet that I referred to earlier has a great many examples of where special characters are used to evade detection.

It is possible for input to use a non-standard character set which might not be properly understood by filtering functions. You should use the database native filter functions when working with filtering SQL.

PHP has a very robust filtering function, `filter_var()`, which can be used to perform a number of different filter and sanitizing operations. You can find a list of the filters in the PHP manual[51].

There are also a number of functions that can be used to check for individual types of strings. They are locale aware and so will take language characters into account. The functions will return true if the string contains only characters in the filter, and false otherwise.

Function	Filters
ctype_alnum()	Alphanumeric characters only
ctype_alpha()	Alphabetic characters only
ctype_cntrl()	String is control characters only
ctype_digit()	String is digits only
ctype_graph()	Only printable characters and space
ctype_lower()	Only lower case letters
ctype_print()	Printable characters
ctype_punct()	Any printable which is not whitespace or alphanumeric
ctype_space()	Check for whitespace characters
ctype_upper()	Only upper case letters
ctype_xdigit()	Hexadecimal digits

It is common to perform filtering on the client side, for example using Javascript in the browser. This is not sufficient and you must filter and validate on the server side as well.

Escape Output

One of the cardinal rules for writing secure PHP code is to filter input and escape output.

Before you emit data you must make sure that it is safe for the client. Recall how XSS attacks work as an example of why you need to make sure that what you send to the client is properly sanitized.

If the data you send to a client includes instructions for it to execute code then it will do so blindly. You must make sure that you send only code you intend for the client to execute, and not code injected by an attacker.

As with filtering input you must not rely on the client to filter output sent to it. Not all clients will have Javascript enabled, and its possible that a hacker bypasses client filtering.

The most secure way to filter output is using `filter_var()` with the FILTER_SANITIZE_STRING flag. There might be use cases where this is too restrictive for you, in which case you will need to look at functions like `htmlspecialchars()`, `strip_tags()`, and `htmlentities()`.

[51] https://secure.php.net/manual/en/filter.filters.php

Log files as output

If you're logging error messages, information messages and the like you need to take some precautions with what you log.

Obviously you must never log sensitive information like user passwords or credit cards. If you're passing this to a logging function then make sure you obfuscate it. So a credit card number would be a sequence of asterisks in your log file, rather than the actual number.

Make sure that you filter out executable code and personal information before logging it.

Encryption, Hashing algorithms

Encryption and hashing are different concepts and you should make sure you understand the difference. Encryption is a two-way operation; you can encrypt and decrypt. Hashing is a one-way operation and by design it is difficult or time-consuming to take a hash and reverse it to the original string.

We store passwords in the database as hashes. This way if an attacker is able to get a copy of your database they are unable to obtain user passwords unless they can reverse the hash. Typically reversing the hash will take a significant amount of time, and hopefully you will have enough time to notice the breach of security and alert your users that they need to change their passwords.

The amount of time that it takes to calculate a hash will determine how long a hacker will take to guess passwords by brute force.

Hash functions

Older hashes like `MD5` and `SHA1` are very quick to calculate and so you must not use them in any place where security is involved. They are still very useful in other areas of programming, but not in any place where you're relying on them being a one-way operation.

PHP 5.5.0 introduce a new function `password_hash()` which provides a convenient way to generate secure hashes. You should read the manual page[52] for this function.

For older versions of PHP you should use the `crypt()` function.

By default the `password_hash()` function uses the bcrypt algorithm to hash the password. The bcrypt algorithm has a parameter that includes how many times it should run on the password before returning the hashed result. This is referred to as the "cost" of the algorithm.

By increasing the number of times the algorithm must run you can increase the length of time that it takes to calculate a hash. This means that as computers get faster you can increase the number of iterations in your bcrypt algorithm to keep your passwords secure from brute force attacks.

You can use the `password_info()` function to retrieve information about how a hash was calculated. This function will tell you the name of algorithm, the cost, and the salt.

The `password_needs_rehash()` function will compare a hash against the options you specify to see if it needs to be rehashed. This will let you change the algorithm used to hash your passwords, for example increasing the cost over time.

[52] https://secure.php.net/manual/en/function.password-hash.php

Salting passwords

A salt string is an additional string that is added to the password. It should be randomly generated for every password. It is used to prevent dictionary attacks and pre-computed rainbow attacks.

You can specify a salt for the `password_hash()` function, but if you omit it then PHP will create one for you. The PHP manual[53] notes that the intended mode of operation is for you to let it create the random salt for the password.

The `crypt()` function accepts a salt string as a second parameter but will not automatically generate a salt if you don't provide your own. PHP 5.6.0+ will issue a notice if you fail to provide a salt.

Checking a password

If it is possible for an attacker to accurately measure the time it takes to run your password checking routine then they will be able to glean information that can help them in breaking the password. These attacks are referred to as timing attacks

The PHP 5.5.0 function `password_verify()` is a timing attack safe way to compare hashes created by `password_hash()`.

If you're unable to use this function then you will need to calculate the hash for the password supplied by the user and then compare the hash against the one stored. Comparing the hashes is vulnerable to timing attacks.

PHP 5.6.0 introduced the `hash_equals()` function which is a timing attack safe way of comparing strings. You should use this function when comparing `crypt()` generated hashes.

A quick note on error messages

You should never confirm to a person that they have entered an incorrect username. Your error message should be that they have entered either an incorrect username or password. The less information you give to an attacker the longer it will take for them to gain access to your system.

File uploads

File uploads are a major risk for a web application and need to be secured in several ways.

Recall that the `$_FILES[]` superglobal contains information about the files that were uploaded by the client. You should treat everything in this array as suspicious and make sure that you manually confirm every piece of information yourself.

The way PHP handles file uploads is to save them to a temporary directory. You can operate on them there and then move them to the location where you want them.

You should check that the file you're working with is a valid uploaded file and that the client has tried to forge its filename and location in the temporary folder.

Use the function `is_uploaded_file()` to make sure that the file you're referencing was actually uploaded. Use the `move_uploaded_file()` instead of other methods to move it from the temporary directory to your final location.

[53] https://secure.php.net/manual/en/password.constants.php

When referring to a file use the `basename()` function to strip out paths to prevent a person from spoofing the filename.

Don't trust the MIME type specified by the user. Ignore the MIME type supplied by the user and use `finfo_file()` to determine the MIME type if you need it.

If you're allowing a user to upload an image you should use a GD image function like `getimagesize()` on it to confirm that it is a valid image. If this function fails then the file is not a valid image.

Generate your own file name to store the file as and do not use the one supplied by the user. Using a random hash for the filename and setting the extension manually by inspecting the MIME type is strongly suggested.

Make sure that the folder where you are storing the files only allows access to the webserver user.

If you don't need to serve the files that are uploaded then keep the uploads folder outside of the document root.

Database storage

In addition to avoiding SQL injection you should apply some security principles to how you interact with the database.

You should separate your database servers for your different code environments. Your QA, test, development, and production servers should all use different database servers and should not be able to access each other's databases.

You must prevent the Internet from having access to your database server.

This can be accomplished by using a firewall to close the port from outside traffic, using a private subnet that has no route to the Internet, or configuring your database server to listen only to specific hosts.

It's not sufficient to change the port that your database listens on. I'd go so far as to say it's not worth bothering because it's not even a speed bump to an attacker and just makes your server environment harder for your colleagues to use.

If you run several applications on a single database server then make sure that each application has its own username and password on the server. Each application user should have only the least amount of privileges it needs and should never be able to read another applications database.

Avoid using predictable usernames and make sure that you use secure passwords. For example I usually use a randomly generated version 4 UUID as a password.

Encrypt sensitive data with `mcrypt()` and `mhash()` before placing it into the database.

You should examine your database logs from time to time. You'll be able to spot attempted injection attacks and other patterns that will let you identify breaches or tighten areas of code.

Avoid publishing your password online

A good piece of advice is to avoid publishing your database or API credentials online where people can read it. Okay, I'm being facetious, but seriously when would you be likely to publish all of your access credentials for the world and his dog to read?

One time you could do this is when committing to a Git repository and pushing it to a service like Github or Bitbucket.

Make sure that any configuration files are ignored by your version control system and are never committed or pushed to upstream repositories. There are bots that scrape Github[54] for credentials that will punish you for these mistakes.

Just as an aside related to this link, you should not hard code Amazon credentials into an application. Rather set an IAM role that allows access to the service you want to use and apply the role to your VM.

[54] http://www.devfactor.net/2014/12/30/2375-amazon-mistake/

Data Formats and Types

XML

The basics of XML

XML stands for eXtensible Markup Language and is a way to store data in a structured manner. An advantage of using XML is that it is a well recognized data standard and so is a convenient way to exchange data between systems.

In the industry there has been a shift away from XML and towards JSON as a data exchange process, but XML is still relevant to everyday practice and is part of the Zend examination.

You need to be completely familiar with the following terms:

Term	Description
SGML	Standardized General Markup Language. XML is a subset of this.
Document Type Declaration	The DTD defines the legal building blocks of an XML document structure with a list of legal elements and attributes[55].
Entity	An entity can declare names and values that are not permitted in the rest of the XML document. For example HTML declares > as an entity to represent >. These declarations can also be used as shortcuts and to maintain consistency of spelling and value throughout a document.
Element	Elements are the building blocks of an XML document. Elements can be nested and contain elements, or they can contain a value. Elements may have attributes.
Well-formed	A well-formed document in XML is a document that adheres to the syntax rules specified by the XML 1.0 specification in that it must satisfy both physical and logical structures[56].
Valid	An XML document validated against a DTD is both "Well Formed" and "Valid".

If you're at all shaky about the above definitions then please make sure that you read a comprehensive tutorial on XML[57] and read the linked footnotes from this page.

Well-formed and valid

We should expand on what these terms mean.

A document is *well-formed* if:

- It has a single root element

[55] https://en.wikipedia.org/wiki/Document_type_definition
[56] https://en.wikipedia.org/wiki/Well-formed_document
[57] For example the one at http://www.w3resource.com/xml/xml.php

- Tags are opened and closed properly
- Entities are well-formed
    - Contain only properly encoded Unicode characters
    - No syntax marks like < or & appear
    - Tag names must match exactly and may not contain symbols

A document is *valid* if it is well-formed and conforms to the DTD.

PHP does not require XML documents to be valid but it does require them to be well-formed in order to parse them.

XML Processing Instructions

Processing Instructions[58] allow documents to contain instructions for applications. They are enclosed in `<?` and `?>` marks and look like this, for example:

```
<?PITarget PIContent?>
```

One use case could be to inform an application that an element is to be a particular data type, as in this example:

```
<?var type="string" ?>
```

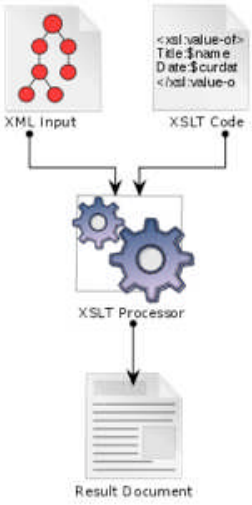
The most common usage is to include an XSLT or CSS stylesheet, like so:

```
<?xml-stylesheet type="text/xsl" href="style.xsl"?>
<?xml-stylesheet type="text/css" href="style.css"?>
```

XML transformations with PHP XSL

The PHP XSL extension allows PHP to apply XSLT transformations.

Although this is commonly used to apply stylesheets it is important to know that many other forms of transformation are possible.



XSL is a language for *expressing stylesheets* for XML documents. It is like CSS in that it describes how to display an XML document.

XSL defines XSLT[59] that is a *transformation language* for XML documents that allows XML documents to be processed into other documents.

An XSLT processor takes an input XML file, some XSLT code, and produces a new document. The diagram to the right, taken from Wikipedia Creative Commons[60], illustrates this.

A use case for this could be to create an XHTML document that can be rendered by a browser.

Input XML would be received from a PHP program that includes processing instructions about where to retrieve an XSL stylesheet. The browser would retrieve this stylesheet and apply the XSLT code in it to produce the XHTML.

---

[58] http://www.w3.org/TR/REC-xml/#sec-pi
[59] https://en.wikipedia.org/wiki/XSLT
[60] https://en.wikipedia.org/wiki/XSLT#/media/File:XSLT_en.svg

Acronym	What is is
XSL	Language to express style sheets
XSLT	Transformation language to process XML into other XML

The PHP manual[61] has a simple example of how to use PHP5 to transform an XML file using an XSL:

```
<?php

$xslDoc = new DOMDocument();
$xslDoc->load("collection.xsl");

$xmlDoc = new DOMDocument();
$xmlDoc->load("collection.xml");

$proc = new XSLTProcessor();
$proc->importStylesheet($xslDoc);
echo $proc->transformToXML($xmlDoc);
```

Parsing XML in PHP

There are two types of XML parser available in PHP. All of the PHP XML extensions use the same underlying library so it is possible to pass data between them.

All XML routines require both the LibXML extension and the Expat library to be enabled. These are both enabled by default in PHP.

Tree parsers

Tree parsers attempt to parse the entire document at once and transform it into a tree structure. It should be clear that this could present problems if you're trying to parse a very big document.

There are two tree parsers in PHP:

- SimpleXML
- DOM

Event-Based parsers

These parsers are quicker and consume less memory than tree parsers. They work by reading through the XML document node by node and providing you the opportunity to hook into events associated with this reading process.

Two examples of event-based parsers are:

- XMLReader
- XML Expat parser

The XML Expat parser is a non-validating event based parser that is also built into PHP's core. It does not require a DTD because it does not validate XML and only requires that XML be well-formed.

[61] https://secure.php.net/manual/en/xsl.examples.php

Error codes

The PHP manual lists a number of XML error codes[62] which is a subset of the 733 error codes of the underlying libxml library[63].

Here is a partial list of XML constants[64] that you should be familiar with:

Prefix	Code	Description
XML_ERROR_		
	SYNTAX	
	INVALID TOKEN	
	UNKNOWN_ENCODING	
XML_OPTION_		
	CASE_FOLDING	Enabled by default and sets element names to uppercase.
	SKIP_WHITE	Skips excess whitespace in the source document

Character encoding

When PHP parses an XML document it performs a process called *source encoding*[65] to read the document.

There are three forms of encoding that are supported:

1. UTF-8
2. ISO-8859-1 (default)
3. US-ASCII

UTF-8 is a multibyte[66] encoding scheme which means that a single character may be represented by more than one byte. The other two schemes are both single byte.

PHP stores the data internally and then performs *target encoding* when it passes the data to functions.

The target encoding is set to the same as the source encoding by default, but this can be changed. The source encoding, however, *cannot* be changed after the parsing object has been created.

If the parser encounters a character that the source encoding cannot represent, it will return an error.

If the target encoding scheme cannot contain a character then that character will be demoted to fit the encoding scheme. In practice this means that they are replaced with a question mark.

The XML extension

The XML extension allows you to create XML parsers and define handlers. You should be familiar with the following functions.

xml_parser_create($encoding)

Create an XML parser with the specified encoding.

---

[62] https://secure.php.net/manual/en/xml.error-codes.php
[63] http://www.xmlsoft.org/html/libxml-xmlerror.html#xmlParserErrors
[64] https://secure.php.net/manual/en/xml.constants.php
[65] https://secure.php.net/manual/en/xml.encoding.php
[66] https://en.wikipedia.org/wiki/Variable-width_encoding

xml_parser_create_ns($ecoding, $seperator=":").
Creates an XML parser with the specified encoding that supports XML namespaces.

xml_parser_free($xmlparser)
Frees up an xml parser.

xml_set_element_handler($xmlparser, $start, $end)
This tells the parser which functions to call at the start and end of each element in the XML document. You can pass FALSE to disable a particular handler.

Both $start and $end must be callable and are usually the string names of a function that exists in scope.

The function that handles the start of an element must accept three parameters:
1. The xml parser resource
2. A string that will contain the name of the element being parsed
3. An array of attributes that the element has

The end handler function must accept two parameters:
1. The xml parser resource
2. A string that will contain the name of the element being parsed

xml_set_object($xmlparser, $object)
This function allows the xml parser to be used within the object. This means that you can set the methods of the object as functions for the setting element handler.

xml_parse_into_struct($parser, $xml, $valueArr, $indexArr)
This function parses an XML string into 2 parallel array structures, one (*index*) containing pointers to the location of the appropriate values in the *values* array. These last two parameters must be passed by reference.

DOM

DOM is an acronym of Document Object Model. The DOMDocument class[67] is useful for working with XML and HTML.

It uses UTF-8 encoding and requires the libxml2 extension (Gnome xml library) and expat library. It is a tree parser and reads the entire document into memory before creating an internal tree representation.

Here is a basic example of some DOMDocument syntax:

[67] https://secure.php.net/manual/en/class.domdocument.php

```php
<?php
$domDoc = new DomDocument();
$domDoc->load("library.xml");
// $domDoc->loadXML($xmlString);
// $domDoc->loadHTMLFile("index.html");
// $domDoc->loadHTML($htmlDocumentString);
$domDoc->save(); // (to a file in XML format)
$xmlString = $domDoc->saveXML();
$htmlDocumentString = $domDoc->saveHTML();
$domDoc->saveHTMLFile(); // (to a file in HTML format)
$xpath = new DomXpath($dom);
$elements = $xpath->query("//*[@id]"); // find all elements with an id
echo "I found {$result->length} elements<br>";
if (!is_null($elements)) {
    foreach ($elements as $element) {
        echo "<br/>[". $element->nodeName. "]";

        $nodes = $element->childNodes;
        foreach ($nodes as $node) {
            echo $node->nodeValue. "\n";
        }
    }
}
```

You should be familiar with the following methods of the DOM class:

Method	Description
createElement	Creates a node element that can be appended with the *appendChild* method of the node class
createElementNS	As with createElement but supports documents with namespaces
saveXML	Dumps the XML tree back into a string
save	Dumps the XML tree back into a file
createTextNode	creates a new instance of class DOMText

DOM Nodes

The DOMNode class[68] is used to work with nodes in the DOM tree.

You can retrieve nodes by calling one of these methods of the DOMDocument:

- *getElementsById*
- *getElementsByTagName*
- *getElementsByTagNameNS*

[68] https://secure.php.net/manual/en/class.domnode.php

These methods return a DOMNodeList object which can be traversed over using `foreach.`

You should be familiar with these methods of the DOMNode class.

Method	Description
appendChild	Adds a new child node at the end of the children
insertBefore	Adds a new child before a reference node
parentNode	The parent of the node, or null if there is no parent
cloneNode	Clones a node and optionally all of its descendent nodes
setAttributeNS	Adds a new attribute

> Note that you need to pass a Node as an argument to these functions. So if you're trying to use `appendChild()` then you must first use a function like `DOMDocument::createElement()` to create the node.

SimpleXML

SimpleXML is an extension that sacrifices robust handling of complex requirements in favour of offering a simple interface. It requires the simpleXML extension and only supports version 1.0 of the XML specifications.

SimpleXML is a tree parser and loads the entire document into memory when parsing it. This may make it unsuitable for very large documents.

SimpleXML offers an object orientated approach to accessing XML data. All of the objects that it makes are instances of the `SimpleXMLElement` class. Elements become properties of these objects and attributes can be accessed as associative arrays.

Creating SimpleXML objects

You can create SimpleXML objects using procedural methods, or through an object orientated approach.

```
$xml = simple_xml_load_string($string_of_xml);
$xml = simple_xml_load_file('filename.xml');
$xml = new SimpleXMLElement($string_of_xml);
```

Iterating over SimpleXML objects

The `children()` method returns a traversable array of child objects.

You can create an algorithm that inspects the children of a node and then iterates through them recursively. There is such an example on the PHP manual page[69].

Retrieving information

[69] https://secure.php.net/manual/en/simplexmlelement.children.php

Function	Action
SimpleXMLElement::construct()	Creates a new SimpleXMLElement object
SimpleXMLElement::attributes()	Identifies an elements attributes
SimpleXMLElement::getName()	Retrieves an elements name
SimpleXMLElement::children()	Returns the children of the given node
SimpleXMLElement::count()	Returns how many children a node has
SimpleXMLElement::asXML()	Returns the element as a well-formed XML string
SimpleXMLElement::xpath()	Runs an xpath query on the current node

xpath

XPath is a language to define parts of an XML document. It models an XML document as a series of nodes and uses path expressions for navigating through and selecting nodes from the document.

`SimpleXMLElement::xpath()` runs an XPath query on XML data and returns an array of children that match the path specified.

W3Cschools has a number of examples of XPath usage[70].

You should note that unlike PHP structures XPath results are not zero based. The XPath `/college/student[1]/name` will return the *first* student, not the second as would be the case if it were zero based.

PHP arrays containing xpath results are zero based. In other words if you store your results in an array variable called `$array` then `$array[0]` will correspond to `college/student[1]/name` in the previous example.

You can retrieve text values by using an XPath like this: `/college/student/name[text()]`

You can specify ranges like this: `/college/student[attendance<80]/name`

Exchanging data between DOM and SimpleXML

The function `simple_xml_import_dom()` will convert a DOM node into a SimpleXML object.

You can convert a SimpleXML object to a DOM with `dom_import_simplexml()`

SOAP

SOAP is an acronym of Simple Object Access Protocol. Versions 1.0 and 1.1 were released by the industry. As of version 1.2 the standard is controlled by the W3C and the acronym has fallen away, making SOAP just a plain name.

The PHP SOAP extension is used to write SOAP servers and clients. It requires that libxml is enabled, which is the case in default PHP installations.

SOAP cache functions are configured in the php.ini file with the `soap.wsdl_cache_*` settings.

If SOAP is available then it makes available a set of predefined constants. These constants relate to soap versions, encoding, authentication, caching, and persistence.

[70] http://www.w3schools.com/xsl/xpath_examples.asp

There are two SOAP functions:

- `is_soap_fault` returns whether a SOAP call has failed.
- `use_soap_error_handler` is used for the SOAP server and sets whether PHP should use the SOAP error handler or not. If it is set to false the PHP error handler is used instead of sending a SOAP error to the client.

The rest of SOAP functionality is provided in a number of classes.

What SOAP does

SOAP allows complex data types to be defined and exchanged and provides a mechanism for various messaging patterns, the most common of which is the Remote Procedure Call (RPC).

This in effect allows a developer to execute a function on a server, pass it complex data as parameters and receive complex data back.

SOAP web services are defined by a WSDL (Web Service Description Language). Most people pronounce this acronym as "whizz-dill".

The WSDL defines the data types using an XML structure. It also describes the methods that may be called remotely, specifying their names, parameters and return types.

SOAP messages between a server and client are sent in XML structures called SOAP envelopes.

Using a SOAP service

The SoapClient class is used to connect to and use a SOAP service.

It is able to parse a WSDL file to discover what methods are available and then present these to you in an easy-to-use manner.

```
<?php
$client = new SoapClient("http://example.com/login?wsdl");
$params = array('username'=>'name', 'password'=>'secret');
// call the login method directly
$client->login($params);

// If you want to call __soapCall, you must wrap the arguments in another array
as follows:
$client->__soapCall('login', array($params));
```

In the above example we connect to an example wsdl and call the login method using two different methods. Note that using the `SoapClient::__soapCall()` method requires you to wrap the parameters in an array.

It is not compulsory for a SOAP service to provide a WSDL. If you need to use such a service you may pass null as the WSDL file but then need to provide information about the service endpoint. You must provide the *location* and *uri* options and may optionally provide other information about the version of the SOAP service, as in this example:

```
<?php
$client = new SoapClient(null,
    ['location' => 'http://example.com/soap.php',
        'uri' => 'http://test-uri/',
        'style'    => SOAP_DOCUMENT,
        'use'      => SOAP_LITERAL));
    ]);
```

When you construct the SoapClient class you can set the `trace` parameter to true to enable debugging the raw SOAP envelope headers and body.

The following two debugging commands require that trace be true and allow you to inspect details of the request:

- `SoapClient::__getLastRequestHeaders()`
- `SoapClient::__getLastRequest()`

Offering a SOAP service

The SoapServer class provides a SOAP server. It supports version 1.1 and 1.2 and can be used with or without a WSDL service description.

Here is an example of setting up a SOAP server:

```
<?php
$options = ['uri'=>'http://localhost/test'];
$server = new SoapServer(NULL, $options);
$server->setClass('MySoapServer');
$server->handle();
```

We can see that we first create the server with an array of options. In the example above we are not supplying a WSDL in the first parameter and so we have to supply the uri of the server namespace in the options array.

Once we have an instance of the SoapServer class we pass in the name of the class that it will use to serve requests. The methods in the class will be callable by a SOAP client connecting to the server.

Instead of setting a class you may also use a concrete object to handle SOAP requests by passing it as a parameter with the `SoapServer::setObject()` function.

REST web services

REST is an acronym for Representational State Transfer and is an architectural style rather than a PHP extension or set of commands. REST has a number of constraints that are intended to improve performance and maintainability of web services.

REST has a number of verbs that are similar to HTTP request types. This leads to some confusion, but it is important to note that REST does not have to use HTTP as a transport layer to communicate. HTTP just happens to be very convenient for REST because it is stateless and the request types translate well into REST verbs.

REST exposes Uniform Resource Identifiers (URI) that are linked to resources. These links are called REST endpoints. Depending on the HTTP type used to access them will perform an action on the resource (change its state). The HTTP type is used to signal the REST verb to be performed.

REST focuses on resources and providing access to those resources. A resource could be something like a "user". Much like a database schema represents the user entity, REST will represent the user in a JSON or XML structure.

A representation should be readable by both the server and the client. REST can be used to transfer JSON, XML, or both. We'll look at this in a bit more detail later.

In PHP one of the most common uses for REST API's is to provide services for an AJAX enabled frontend, such as one written in Angular.

Application and resource states

A REST server should not remember the state of the application and the client should send all the information necessary for execution.

This means that every request to a server is self-contained. If a request to a server failed it will not affect the success or failure of other requests. This improves the reliability of the application.

The server is not responsible for remembering what state the application is in and relies on the client to send all the information it needs in order to process the request. This means that *application state* is stored by the client.

Application statelessness has important implications for scaling horizontally. Because no individual server is maintaining state a request can reach any server in a group and be handled correctly.

The resource that REST is providing access to has state that is expected to persist between requests. *Resource state* is maintained on the server.

REST verbs

REST has a number of verbs that are used to alter the state of a resource on the server.

Verbs operate either on a single resource, or a *collection* of resources.

Resource	GET	PUT	POST	DELETE
Collection	Lists the URI's where you can retrieve the members	Replace the collection with another collection	Create a new entry in the collection	Deletes the entire collection
Single	Retrieve a representation of the single element	Replace the element, or create it if it doesn't exist	Creates a new member	Deletes the member

PUT and POST look similar, but have an important distinction. POST requires you to specify all of the required attributes for an element and will create a fresh element. PUT will replace the attributes *you specify* for an existing record and you don't need to supply all the attributes unless you're creating a new record.

To explain with an example, lets consider a user who has a name and a title. First we POST to create a new user with a name "Alice" and the title "Mrs". Then Alice graduates and becomes a doctor, so we PUT to her record and include just the title as "Dr". We don't have to specify her name and because we don't her name will not be changed.

Request headers

HTTP allows passing headers in its request. REST clients will use these to indicate to the server what they are providing and what they are expecting back.

A REST client should use the *accept* header to indicate to the server what sort of content (representation) it wants back. For example, if a client sets the accept header to `text/xml` it is telling the server that it wants an xml formatted response.

The client will also set a *content-type* header to inform the server of the MIME type of its payload. See the section in the response header for more detail.

Response headers and codes

The *content-type* header is sent by the server and defines the MIME type of the body that is being sent. For example a server may set the content-type to `application/json` to indicate that the body of the response contains JSON formatted text.

The server will also set a status code that informs the client of the result of the request. Some of the common codes are listed below, but there are many more[71].

Code	Meaning
200	The request processed successfully
201	The resource was created
202	The resource was accepted for processing, but has not yet been processed
400	Bad request (client error)
401	Unauthorized, the client may not perform this action
500	Server error

It is very poor practice to send an error message in the response body but have a successful status code. This is a fairly common problem in API's that you will encounter in the wild.

Within the Zend framework the term "context switching" refers to changing the output of your program depending on whether it is responding to a REST request or some other request.

For example you may respond with an HTML page for normal requests or respond with JSON if the request originated via XMLHttpRequest (AJAX).

You could also respond with XML or JSON depending on what content type the client indicates it wants as a response.

Another example could be to respond with different layouts depending on what sort of browser is being used (mobile device versus desktop for example).

You should be familiar with the concept of the server responding differently to a call to the same URL depending on how the client sets up its request.

[71] https://en.wikipedia.org/wiki/List_of_HTTP_status_codes

Sending requests

The curl extension[72] is a common way to send REST requests in PHP. Curl lets you specify headers and request types.

There are libraries that wrap the curl functions. One of the popular ones is Guzzle[73] which is easy to install and use.

JSON

JSON is an acronym of Javascript Object Notation. In PHP it is used a lot with Ajax which is an acronym for Asynchronous Javascript and XML.

JSON lets you serialize an object as a string so that it can be transported between services. Ajax is a means to transport the string.

Together these technologies allow you to communicate between Javascript applications in the browser and PHP applications on the server.

The JSON extension is loaded in PHP by default and provides methods to handle converting to and from JSON.

It provides a number of constants including:

Constant	Meaning
JSON_ERROR_NONE	Confirms whether a JSON error occurred or not
JSON_ERROR_SYNTAX	Confirms if there was a syntax error parsing JSON and helps detect encoding errors.
JSON_FORCE_OBJECT	If an empty PHP array is encoded this option will force it to be encoded as an object

There are three functions provided by the extension.

`json_decode()` takes a string as its first argument and returns an object. If the second parameter is set true it will return an associative array.

From PHP 5.3 onwards two additional options are supplied - `$depth` and `$options`. Depth refers to the recursion depth and currently the only option is `JSON_BIGINT_AS_STRING` which changes casting large integers as floats to be cast as strings.

`json_encode()` takes a variable of any type (other than a resource) as a parameter and returns the JSON representation. It has two optional parameters, `$depth` and `$options` which are the same as described above.

`json_last_error()` returns the last error that occurred in either of the above functions.

Date and Time

PHP supplies a number of functions that retrieve the date and time from the server. As of PHP 5.1 you should set a default time zone in your configuration, or set it at runtime in your script. You should set the time zone to

[72] https://secure.php.net/manual/en/book.curl.php
[73] https://github.com/guzzle/guzzle

match the time zone that your server is in, so that PHP can correctly interpret the server time. This also lets your script be aware of adjustments like daylight savings time.

PHP 5.2 introduced the `DateTime` class which deals with a wide range of date and time calculations. It is recommended to use this class instead of working with the functions like `date()` and `time()`. It's worth mentioning that the class does not handle microseconds and if you need to work with timestamps that include them you will need to use the `microtime()` function.

To create a new DateTime object you pass it a string that it can parse. It understands a wide range of string formats, such as shown in this example:

```
<?php
$strings = [
    'Next monday',
    'Yesterday',
    '', // now
    '2015-12-25',
    '25 December 2015',
    '-1 week',
    '+1 days'
];
foreach ($strings as $example) {
    $dateTime = new DateTime($example);
    echo $dateTime->format(DateTime::COOKIE) . PHP_EOL;
}
```

All of the strings in the array will be understood.

If a date format is ambiguous then you can use the `DateTime::createFromFormat()` command to create the object.

For example the date 3 June 2013 would be written as 06-03-2013 by an American while the rest of the world would write it as 03-06-2013. If you gave either of these strings to PHP it would not know whether you mean 3 June 2013 or 6 March 2013.

To resolve the ambiguity you can specify which format you're using in your string, like this:

```
<?php
$dateTime = DateTime::createFromFormat('d-m-Y', '06-03-2013');
echo $dateTime->format(DateTime::COOKIE);
```

This script will output something like `Wednesday, 06-Mar-2013 12:56:42 CET`. Note that if you omit the time when creating a DateTime class the time that the script is running at will be used.

Formatting dates

In the examples above we've used one of the class constants provided by DateTime to format our date.

The manual[74] has a list of these constants, which are common use cases for date display or storage. They appear in this table:

Constant	Format
ATOM	Y-m-d\TH:i:sP
COOKIE	l, d-M-Y H:i:s T
ISO8601	Y-m-d\TH:i:sO
RFC822	D, d M y H:i:s O
RFC850	l, d-M-y H:i:s T
RFC1036	D, d M y H:i:s O
RFC1123	D, d M Y H:i:s O
RFC2822	D, d M Y H:i:s O
RFC3339	Y-m-d\TH:i:sP
RSS	D, d M Y H:i:s O
W3C	Y-m-d\TH:i:sP

These are string constants and contain date and time formatting codes. The formatting codes are replaced with a value by the DateTime class. For example the symbol "Y" is replaced with the 4 digit year of the date being stored.

Obviously the point of declaring the constant is so that you don't have to memorize the strings, so don't worry about studying the formats. I included the formatting strings because they are a good indication of the commonly used ones.

Date and time formatting codes are *case-sensitive*. For example "y" is a two digit year and "Y" is a four digit year.

Characters in the formatting string that are not recognized formatting characters will be placed into the output unchanged. So the string "Y-m-d" would include the hyphens between the year, month, and day when output – like this "2015-12-25".

You can find a list of the PHP date and time formatting codes on the manual page[75], but here are the ones that are in the table above:

[74] https://secure.php.net/manual/en/class.datetime.php
[75] https://secure.php.net/manual/en/function.date.php

Code	Replaced with	Example(s)
Y	A full 4 digit year	1999
m	Two digit month, with leading zeroes	06
d	Day of the month, 2 digits with leading zeros	14
D	A three letter textual day (e.g.:	Mon, Tue, Wed
H	24 hour format hour with leading zero	00, 09, 12, 23
i	Two digit minute, with leading zeroes	05,15,25,45
s	Two digit seconds, with leading zeroes	05,15,25,45
P	Difference to Greenwich time (GMT) with colon between hours and minutes (PHP 5.1.3+)	+02:00
O	Difference to Greenwich time (GMT) in hours	+0200
T	Timezone abbreviation	EST, CET

Date calculations

The most simple calculations can be performed using the DateTime class method `modify()`. For example to find the date and time that is one month in the future you can do the following:

```
<?php
$dateTime = new DateTime();
$dateTime->modify('+1 month');
echo $dateTime->format(DateTime::COOKIE) . PHP_EOL;
```

PHP offers a much more flexible way to work with date calculations, however.

The DateInterval class[76] is used to store either a fixed amount of time (in years, months, days, hours etc) or a relative time string in the format that DateTime's constructor supports.

The DateTime class allows you to `add()` or `sub()` a DateInterval from a DateTime. It will handle leap years and other time adjustments while doing so.

To specify a fixed amount of time when creating a DateInterval object we pass its constructor a string. The string always starts with P and then lists the number of each individual date unit in descending order. Optionally the letter T appears and then the time units are included.

This makes a lot more sense with some examples:

String	Description
P14D	14 days
P2W	2 weeks
P2W5D	This is invalid – you may not specify weeks and days
P2WT5H	2 weeks and five hours
P1Y2M3DT4H5M	1 Year, 2 months, 3 days, 4 hours, 5 minutes

Note that:

1. Every string begins with P,
2. The number of units precedes the letter indicating the unit,
3. Time units are split from the date units by the letter T,
4. Units are sorted in descending order

[76] https://secure.php.net/manual/en/class.dateinterval.php

Here is an example in code:

```
<?php
$dateTime = DateTime::createFromFormat('d-m-Y H:i:s', '06-03-2013 13:14:15');
$dateInterval = new DateInterval('P1M2DT3H4M5S');
$dateTime->add($dateInterval);
echo $dateTime->format(DateTime::COOKIE) . PHP_EOL;
```

This code outputs the date and time that is 1 month, 2 days, 3 hours, 4 minutes, and 5 seconds after 6 March 2013 13:14:15.

Comparing dates

The DateTime `diff` method allows you to compare the difference between two DateTime objects. It returns a DateInterval that contains the period of time between the two dates being represented.

Note that the DateTime class handles timezone and daylight savings time conversions for you.

Lets try and find out how long it is to Christmas.

```
<?php
$now = new DateTime();
$christmas = new DateTime('25 december');
if ($now > $christmas) {
    $christmas = new DateTime('25 december next year');
}
$interval = $christmas->diff($now);
echo  $interval->days . ' days until Christmas' . PHP_EOL;
```

Notice the following in this snippet:

1. Passing no parameter to the construct uses the current date and time
2. We can use mathematical operators like >, <, and == to compare DateTime objects
3. We can use fairly flexible language when creating a DateTime, such as “25 december next year” for the case where the current date is between Christmas and New Year
4. The `diff()` method returns a DateInterval
5. The DateInterval object has a number of public properties that can be accessed to measure years, months, and in this case days.

Input-Output (I/O)

PHP 4.3 introduced streams as a way of generalizing file, network, data compression, and other operations that share a set of common set of functions and uses.

Files

There are two main groups of functions to deal with files those that work with file *resources*, and those that work with a file *name*.

Remember that a resource is a type of variable that can't be stored directly in PHP. A file resource is an operating system file handle. All of the functions that deal with file resources begin with a single "f" letter and then have a verb describing their function. For example `fopen()` opens a file resource.
Functions that work with the string name of a file all start with the word "file" and are followed by a verb descriptive of what they do. For example `file_get_contents()` takes a string file name and returns the contents of that file.

Opening Files

The function `fopen()` is used to open files. It returns a resource variable which is a handle to the file.

You must pass two parameters to `fopen():`

1. The file name
2. The file mode

File Modes

Files can be opened in different modes. File modes describe how we will be interacting with the file.

File modes relate to operating system file privileges. For example if the PHP user only has read access to a file then an attempt to open it in write mode will be denied by the operating system. If we try with a lesser privilege (such as read only) then the operating system will create a file handle for us.

We communicate two pieces of information about how we intend to use a file when we specify a mode:

1. Whether we are reading, writing, or both
2. Whether we want to place the file pointer at the beginning or ending of the file

The file pointer is like an iterator cursor. It stores the file position that will be returned on the next read. I find it easier to remember the file modes by thinking about what their intention is.

Mode	Intention	Read / Write	Pointer	Extra
r	Read	Read	Start	
r+	Read	Read / Write	Start	
w	Overwrite	Write	Start	Truncate
w+	Overwrite	Write / Read	Start	Truncate
a	Append	Write	End	
a+	Append	Write / Read	End	
x	Create	Write	n/a	Fails if file exists
x+	Create	Write / Read	n/a	Fails if file exists
c	Create	Write	Start	Create if not exists
c+	Create	Write /Read	Start	Create if not exists

You'll notice that adding a + symbol to a file mode has the effect of indicating that you also want to perform the opposite of the default mode. So when we're overwriting a file if we add a + symbol then we indicate that we also want to read the file.

When using the "w" modes to overwrite a file PHP will truncate the file to zero bytes.

The "c" mode will create a file if it exists or open an existing file. The pointer will be set to the start of the file for existing files. This contrasts with the "x" mode where if the file exists the function will fail.

There are two flags that you can specify by adding them to the end of the mode string. The default flag is defined by your SAPI and version of PHP that you're using, so for compatibility purposes you should specify them.

You can specify a "b" flag to specify that you're working with binary files. This means that no characters will be translated. This is necessary when you're working with images or other binary files.

On a Windows server you can specify a "t" flag to translate "\n" to "\r\n".

In order to keep your code portable you should use the "b" flag and make sure that your code uses the correct line endings.

Reading

You can read from a file resource using the `fread()` function.

```
<?php
$handle = fopen('info.txt', 'r');
while (!feof($handle)) {
    echo fread($handle,1024);
}
```

In this example we're using the file function `feof()` which is a function that returns TRUE when the file pointer is at the end of the file and FALSE otherwise. Using it in a while loop as above has the effect of continuing the loop until we reach the end of the file.

The `fread()` function takes two parameters. The first is the variable holding the file resource, and the second is the number of bytes to read. If it reaches the end of the file then `fread()` will stop reading.

Here are three more PHP functions that make it easier to read files.

Function	Used to
fgetcsv()	Read a line from file pointer and parse for CSV fields
file_get_contents()	Take a string filename and read the results into a string
readfile()	Read a string filename and writes the contents to the output buffer
file()	Reads an entire file into an array

Writing

Writing to a file is done with the binary-safe `fwrite()` function. `fputs()` is an alias to this function.

The `fwrite()` function takes two parameters – the file resource to write to, and the string to write to the file.

There is a writing counterpart for the `freadcsv()` function, namely `fputcsv()` which formats an array as CSV and writes the line to a file. In addition to parameters for the file resource and array it takes optional parameters to define the CSV format.

If you want to write formatted strings to a file then you should use `fprintf()` which works like the `printf()` command.

If you want to dump the contents of a file to a connected client you can use `fpassthru()`. This function will start at the current file position and write the rest of the file to the output buffer.

Finally there is a convenient function to quickly write a string to a file. The function `file_put_contents()` doesn't require you to provide a file resource and just requires the file name and the string you want to write.

File System Functions

PHP has an extensive list of functions that connect you to the file system. We'll deal with a few of them in this chapter, but as I so often do I'm going to refer you to the PHP manual[77] for the exhaustive list.

Directories

This group of functions let you traverse, create, and delete directories.

Function	Use
chdir()	Change PHP's current directory
chroot()	Change root directory of the running process to the specified directory and sets PHP's working directory to "/"
rmdir()	Deletes a directory
readdir()	Returns the name of the next entry in the directory handle passed as a parameter. The entries are returned in the order in which they are stored by the filesystem.
scandir()	Reads the directory specified by the string parameter and returns a list of the files and directories it contains

The difference between `scandir()` and `readdir()` is the parameter that they take. Where `readdir()` uses a directory handle, `scandir()` accepts the name of the directory as a string. This is possibly confusing because it seems that the naming convention of file functions (f* versus file*) doesn't apply to directories.

File information

We referred to these functions in the security chapter but there are other use cases where you need to obtain information about a file.

PHP provides the `finfo_open()` function which returns a new instance of a fileinfo resource. You provide it with two parameters – a predefined option constant, and the string location to a magic database file.

The magic database file is a format used to describe file types and is also used by the Unix standard command "file". If you don't supply a path to the magic database then PHP will use the one that it comes bundled with.

[77] https://secure.php.net/manual/en/ref.filesystem.php

Once PHP knows how to identify files you can use the `finfo_file()` function to obtain information about the file. It takes at least two parameters – the fileinfo resource you just created and a string name of the file you want to check.

Here is an example from the PHP manual:

```
<?php
$finfo = finfo_open(FILEINFO_MIME_TYPE);
foreach (glob("*") as $filename) {
    echo finfo_file($finfo, $filename) . "\n";
}
finfo_close($finfo);
```

Both of these functions have object orientated styles of use, as in this example from the PHP manual:

```
<?php
// finfo will return the mime type
$finfo = new finfo(FILEINFO_MIME, "/usr/share/misc/magic");

/* get mime-type for a specific file */
$filename = "/usr/local/something.txt";
echo $finfo->file($filename);
```

Managing Files

You can use PHP to manage files. Some of the common functions are listed in this table.

Function	Purpose
copy	Copies a file
unlink	Deletes a file
rename	Renames a file. You can use this to move a file between directories.
chmod	Set file permissions
chgrp	Change the group of the file
chown	Change the owner of the file (superuser only)
umask	Change the current umask

Determining the Type of a Filesystem Object

It is good programming practice to verify that files and directories exist and that you have the proper permissions to use them in the way you intend.

PHP provides a number of functions that return Boolean values if the object matching the string you pass as the parameter meets the test.

All of these functions take a string parameter that is the name of a file or directory. In the table below the check is against the object found that matches the name given in the parameter.

Function	Checks
is_dir	Is a directory
is_file	Is a file
is_readable	Is a file or directory, and can be read
is_writeable	Is a file or directory, and can be written to
is_executable	Is a file or directory, and can be executed
is_link	Is a symlink
Is_uploaded_file	Was uploaded by a POST request

All of the functions will return FALSE if no filesystem object was found matching the name given in the parameter.

Magic File Constants

PHP has a number of magic constants that you can use in relation to the file currently executing.

Constant	Refers To
__LINE__	The line of the file currently executing
__FILE__	The full path and filename of the file
__FUNCTION__	The current function name
__CLASS__	The name of the class in scope
__METHOD__	The name of the method being executed

These constants are very useful when writing debug logs. For example I typically start all of my Log messages with the `__METHOD__` tag so that it's immediately clear which class and method the log message is generated in.

Streams

A *stream* is almost like a conveyer belt of things that come to you one by one. In PHP you can also skip along the conveyer belt and seek to a particular position.

Streams are referenced in a format that you might recognize:

```
scheme://target
```

For example http://www.php.net specifies the http scheme and the target as the URL of the PHP website.

Stream Wrappers

Wrappers are code objects which translate the stream into a particular encoding or protocol. The PHP manual[78] has a list of the wrappers that are implemented within the language, and the `stream_wrapper_register()` function lets you define your own.

[78] https://secure.php.net/manual/en/wrappers.php

Protocol	Use
file://	Accessing the local file system
http://	Accessing HTTP(s) URLs
ftp://	Accessing FTP(s) URLs
php://	Accessing various I/O streams
compress.zlib://	Compression streams
data://	Data (RFC 2397)
glob://	Find pathnames matching pattern
phar://	PHP Archive
ssh2://	Secure Shell 2
rar://	RAR
ogg://	Audio streams
expect://	Process Interaction Streams

The PHP streams that you can access are php://stdin, stdout, stderr, input, output, fd, memory, temp, filter.

As an example of reading a stream, lets look at how to read the body of a PUT request. At some time in your career you will be coding a REST API and will need to read and parse the body of PUT requests that clients are making to your server. There is no superglobal for this request type as there is for GET and POST, so how is it done? The answer is in the php://input stream!

```
<?php
// reads the PUT body
$input = file_get_contents('php://input');
// parses the input into an array
parse_str($input, $params);
print_r($params);
```

Filters

Stream filters can be applied to streams and perform transformation operations on data leaving the stream.

Filter	Function
string.rot13	Encodes the data with ROT13
string.toupper	Converts the string to uppercase
string.tolower	Converts the string to lowercase
string.strip_tags	Strips XML tags from the string
convert.*	Convert data according to an algorithm, for example convert.base64-encode will encode the data to base64
mcrypt.*	Provides symmetric encryption using libmcrypt
mdecrypt.*	The decryption filter using libmcrypt
zlib.*	Uses the zlib library to compress and uncompress data

These filters are attached to a stream using the `stream_filter_append()` function. You can apply the filter to the read and write directions of the stream independently.

```
<?php
$handle = fopen("files.php", 'a+');
stream_filter_append($handle, 'string.rot13');
while (!feof($handle)) {
    echo fread($handle,1024);
}
```

You can provide a third parameter to `stream_filter_append()` to attach it to reading or writing the stream. The parameter is one of the predefined constants STREAM_FILTER_READ, STREAM_FILTER_WRITE, or STREAM_FILTER_ALL. By default the filter is attached to reads and writes.

The example above will output something like this:

```
<?cuc
$unaqyr = sbcra("svyrf.cuc", 'n+');
fgernz_svygre_nccraq($unaqyr, 'fgevat.ebg13');
juvyr (!srbs($unaqyr)) {
    rpub sernq($unaqyr,1024);
}
```

Stream Contexts

Stream contexts are a wrapper for a set of options that can modify a stream's behaviour.

You create a context with the `stream_context_create()` function. You pass it two optional parameters, both of which are associative arrays. The first parameter is the options, and the second is an array of context parameters.

Each type of stream has its own set of context options. The PHP manual[79] has the exhaustive list of them.

The only parameter available at the moment is a callable that will be called when an event occurs on a stream. The events are all predefined STREAM_NOTIFY_* constants.

The prototype for the callback function is in the PHP Manual, along with an example of notify events for the HTTP stream.

As an example if you are downloading a file you could set up your callback function to respond to the STREAM_NOTIFY_FILE_SIZE_IS event and abort the download if it is too big. This example prevents us from downloading the homepage of www.example.com if it is larger than a kilobyte.

[79] https://secure.php.net/manual/en/context.php

```php
<?php
function callback($notification_code,
    $severity,
    $message,
    $message_code,
    $bytes_transferred,
    $bytes_max)
{
    if ($notification_code == STREAM_NOTIFY_FILE_SIZE_IS) {
        if ($bytes_max > 1024) {
            die("Download too big!");
        }
    }
}

$context = stream_context_create();

stream_context_set_params($context,
    ["notification" => "callback"]);

$handle = fopen('http://www.example.com', 'r', false, $context);

fpassthru($handle);
```

You can change the options and parameters with the `stream_context_set_params()` function while the `stream_context_get_params()` will return the current parameters for the stream.

Web Features

Sessions

HTTP is a stateless protocol which means that the connection between the client and the server is lost once the transaction ends. However in just about any application, the webserver needs to be able to distinguish between and keep track of visitors.

In order to do so the server creates a session for the client. The client will send the session identifier to the server with every request and this allows the server to associate the request with a particular session.

Websites that don't need to remember who a user is don't need to use sessions. An example of such a site would be one that just serves static content that is the same for all visitors.

PHP supports sessions by default but they can be disabled through a configuration setting in php.ini

Starting a Session

A session in PHP is started when you call the function `session_start()` or automatically if your php.ini configuration specifies `session.auto_start = 1`.

If you are using `session_start()` then you must make sure that you call this function before any output is sent to the client.

When the session starts the user is assigned a random unique session identifier called the "session id". The session id is either stored in a cookie on the client or passed through the URL if you enable `session.use_trans_sid` configuration setting.

Accepting sessions from the URL can be risky and it is better to configure PHP to only use cookies with the `session.use_only_cookies` setting. The security section of this manual has more information about this.

Session Identifier and Session Variables

The session extension makes available the `SID` predefined constant that holds the session identifier. You can also use the `session_id()` function to get it, or set it.

You can use the function `session_regenerate_id()` to make a new session identifier for a client. You should call this immediately after calling `session_start()` to help protect against session fixation.

Once a session has started, the superglobal $_SESSION is available as an associative array containing the session variables.

Logging a User Out

In order to log a person out completely you should:

1. set the $_SESSION array to an empty array,
2. set the session cookie expiry time to the past,
3. and then call the function `session_destroy()`[80]

[80] https://secure.php.net/manual/en/function.session-destroy.php

The effect of step 2 is to unset the session identifier on the client side.

Session Handlers

PHP supports creating your own session handler, but by default PHP sessions are stored on disk and use the `serialize()` and `unserialize()` commands to encode and decode the data.

In addition to disk based sessions PHP also ships with a memcache session handler that can be configured in php.ini

If you want to write your own session handler you should implement the `SessionHandler` interface[81] (PHP 5.4+).

This will let you use alternative ways of storing your sessions, and also customize how you encode and decode

> Prior to PHP 5.4 you would need to make multiple calls to the function `session_set_save_handler()` to set each of the methods in the interface class.

the session data.

GET and POST data

HTTP has several different methods that requests can be made[82]. GET and POST are two such methods.

GET requests are used to request a representation of the specified resource. In other words a GET request is trying to retrieve information.

In contrast a POST request is used to send information to the server that you want it to store.

From a technical perspective the main difference between the two lies in how data is transferred. A GET request encodes it data as part of the URL while a POST request encodes the data payload in its body.

Encoding data into URLs

For an example of how variables are encoded into a URL, consider this example:

```
http://example.com/index.php?name=foo&email=test@test.com
```

The variables begin with a question mark and are delimited by ampersand symbols. Each variable is a key value pair with the equals sign denoting the value.

If we visit a page, then all of the variables in the URL are automatically placed into the $_GET associative array.

You can encode arrays into the URL using syntax like this:

```
http://example.com/users.php?sort[col]=name&sort[order]=desc
```

You would be able to access these variables like this:

- echo $_GET[‘sort’][‘col’];
- echo $_GET[‘sort’][‘order’];

[81] https://secure.php.net/manual/en/class.sessionhandlerinterface.php
[82] https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol#Request_methods

File Uploads

We'll focus on how file uploads work and the PHP syntax associated with them in this section. Make sure that you study the section on file uploads in the security chapter in conjunction with this section.

Forms allow files to be uploaded by means of a "multi-part" HTTP POST transaction.

You can specify that you want to encode your POST using multi-part form data in your HTML by declaring a form something like this:

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

Note that I've left the "action" attribute blank. By default an HTML form will submit to the URI that it is served from.

Limiting the size of uploads

You do not want people to be able to upload massive files that fill up your disk. In order to manage the size of files that people can upload you can limit the size in the browser and on the server.

To tell the client to limit the size of the upload you can add an input to your form like this:

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

Limiting the size in the browser should be seen as being just to improve user experience. It is very simple for a user to disable or change the limit in order to bypass the limit.

You should rather configure PHP to limit the size of a POST operation. The `post_max_size` limits the maximum of data any POST may contain. The `upload_max_filesize` is applied to limit the size of files that can be uploaded.

Temporary Files

PHP stores the uploaded file in a temporary location and makes it available to the script that the form POST'ed to.

You can process the file in the temporary location and then optionally move it to a permanent location. PHP will automatically delete the temporary file when the script finishes running so if you want to keep it you have to move it.

In addition to creating the temporary file PHP will populate the $_FILES superglobal array. Each file that was uploaded in the form will have an entry in the array.

You need to be aware that the information in the $_FILES array can quite easily be spoofed so you should manually validate every piece of information.

Each file be represented by an array in the $_FILES superglobal and will keys for the name, type, size, temporary filename, and error code.

Key	Description
name	The original name of the file stored on the client
type	The MIME type provided by the client
size	The size in bytes of the file
tmp_name	The name of the file in its temporary location
error	An error code, or UPLOAD_ERR_OK if the upload was successful

The security section has more information on dealing with file uploads.

Passing variables by POST

When you use a POST request the variables you want to send are sent in the body of the request.

There are different ways that the variables are encoded and you won't need to be able to create the encoding yourself. The browser will create the POST body for you when submit the form. As a simple example a POST request could look something like this:

```
POST / HTTP/1.1
Host: example.com
Accept: application/xml
Cache-Control: no-cache
Content-Type: application/x-www-form-urlencoded

name=Alice&email=alice%40example.com
```

There are three advantages to sending variables with POST:

Firstly, POST data is able to be encoded in a particular character set, this isn't the case with GET.

Secondly, because your variables are being sent in the message body you're not limited as to how much data you can send by the length of the URL.

Lastly, POST allows you to upload files but GET does not.

There is no difference in security between the two methods.

> There is no limit in the HTTP protocol on the length of the URL but there are limits on browsers and other clients. As a general rule don't create a URL longer than 2000 characters.

Forms

Forms allow users to submit data to your PHP script.

When declaring a form in HTML you specify the method it uses to send information to the server. Although you can choose either GET or POST you should make sure that you choose a request method that matches what you intend to do.

PHP automatically makes form data available to your script in one of two superglobals $_GET, $_POST, depending on which method the form used to make the request.

The Request Superglobal

The $_REQUEST superglobal is an associative array that by default contains the contents of $_GET, $_POST and $_COOKIE[83].

> You should know that the argv and argc entries which contain the arguments when using the command line prompt are contained in the $_SERVER array.

The php.ini setting `variables_order` determines which of the GET, POST, and COOKIE variables are present in the `$_REQUEST` array as well as the order. If the same variable is in multiple request types it will take on the value of the last one in the sequence of this settings value.

So, for example, lets imagine the configuration is set to "EGPCS" indicating that POST comes after GET. Then if both $_GET['action'] *and* $_POST['action'] are set then $_REQUEST['action'] will contain the value of $_POST['action'].

Because you won't be certain of exactly where the data in $_REQUEST is coming from you should use this array with caution. Introducing uncertainty in your code complicates your testing.

Form elements

These superglobals can easily be edited by the client and so should always be filtered carefully and not trusted.

Dots and space in form field names are converted to underscores, so the field

```
<input name="email.address" type="text">
```

will be placed into either $_GET['email_address'] or $_POST['email_address'] depending on the forms method.

Arrays in HTML forms

Form data can be turned into an array using syntax like this in HTML:

```
<form action="formhandler.php" method="POST">
    <input type="text" name="name[first]">
    <input type="text" name="name[last]">
    <input type="submit">
</form>
```

This will result in $_POST or $_GET being an array that looks like this:

```
array(
 'name' => array(
     'first' => '',
     'last' => ''
    )
)
```

One of the most useful ways that arrays help is in grouping inputs together.

[83] https://secure.php.net/manual/en/reserved.variables.request.php

Consider a checkbox that can have multiple values:

```
<h1>What pets do you want in your home?</h1>
<form action="formhandler.php" method="POST">
    <input type="checkbox" name="pets[]" value="cats" id="lotsacats">
    <label for="lotsacats">Lots of Cats</label>

    <input type="checkbox" name="pets[]" value="dog" id="adog">
    <label for="adog">Just a dog</label>

    <input type="submit">
</form>
```

Assuming that the person checked both boxes before submitting the form then the $_GET or $_POST array will contain:

```
array(
    'pets' => array('cats', 'dog')
)
```

This makes checkboxes a lot neater and easier to use. You can read more about this in the PHP manual[84].

Selecting Multiple Items From a List

Lastly, you will need to use an array if you want the user to be able to select multiple items from a select list:

```
<select name="var[]" multiple="yes">
```

Note that the name of the select is an array, so each value that the user selects will be added to the "var" array in your superglobal array.

Cookies

Cookies let you store a small (4kb to 6kb) amount of data on the client device. The client will read them and send them with each request.

PHP can store its session identifier in the cookie. The session information is stored on the server and matched to the client through the identifier in the cookie. This is done by PHP for you. By default PHP session cookies are valid until the person closes their browser.

You cannot control cookies on the client device. They can be edited or deleted at any time by the client. This means that you should neither trust the information sent with them nor rely on them to exist. You should also not store sensitive information in cookies.

If you want to delete a cookie you can set an expiry date that is in the past. This will let the client know that the cookie is no longer needed and can be deleted. You have no guarantee that the client will respect this.

[84] https://secure.php.net/manual/en/faq.html.php#faq.html.arrays

A server will set a cookie using the `Set-Cookie` response header. The client will include it with future requests using the `Cookie` request header.

Setting Cookies

The `setcookie()` function is used to set a cookie. The parameters are explained in the PHP manual[85] and are given in the order of this table:

Parameter	Used for
value	Storing a scalar value in the cookie.
expire	Unix epoch timestamp when the cookie expires. You can't rely on the cookie existing until it expires as it is common for people to delete their cookies.
path	The base path on the domain that the cookie will be available on. If you set it to '/' then it will be available on all paths, otherwise it will be available on the path and all sub-paths from it.
domain	The cookie will be available on this and all sub-domains under it. You can only set a cookie that matches your the domain the cookie is being served from.
secure	Tells the client that it should only send the cookie if it is being sent over an HTTPS encrypted connection
httpdonly	Tells the client that it should only send the cookie using HTTP and not make it available to scripting languages like Javascript. To a limited degree this can help reduce XSS and session fixation attacks on clients that support it.

Cookies can only store scalar values. You can, however, use syntax like the following example:

```
<?php
setcookie("user[name]", "Alice");
setcookie("user[email]", "alice@example.com");
```

The next time the person makes a request to the site the $_COOKIE variable will contain something like this:

```
Array
(
    [PHPSESSID] => j1m5od9ngqi3krmu6fkjcebcb4
    [user] => Array
        (
            [name] => Alice
            [email] => alice@example.com
        )

)
```

Note that "user" is an array and that the cookie value also contains the PHP session identifier.

[85] https://secure.php.net/manual/en/function.setcookie.php

Retrieving Cookies

You can access the cookie information using the `$_COOKIE` superglobal.

Remember that this array is populated with information from the cookie sent by the client. This means that if you use `setcookie()` to create or change a cookie the `$_COOKIE` array will only contain the new information when the client makes a new request.

HTTP Headers

HTTP headers are sent with the request from the client and with the response from the server. They are used to convey information about the HTTP message such as what sort of information is being provided and what will be accepted in return.

HTTP headers take the form of a name-value pair in a clear text string. A carriage return and line feed character follows each header. There is no limit in the standard but most servers and clients impose limits on the length of a header and the total number of headers that may be sent in one request/response.

PHP will automatically emit valid headers for you, but there are several cases where you may want to send your own header.

Sending headers

The PHP function `header()` lets you send a header to the client. You may only send headers before any normal content has been sent to the client. One of the reasons that it is common to omit the closing `?>` tag in included PHP files is to avoid having a new line character occur after the tag. This character would be sent as HTML content and would prevent you from being able to send headers.

The parameters sent to `header()` are as follows:

Parameter	Description
Header string	String containing the header to set. For example: "Cache-Control: no-cache, must-revalidate"
Replace	Boolean to indicate whether this header must replace a previously sent header with the same name.
Response code	The HTTP response code to send with the header

There are two special cases for headers.

The first is for headers that begin with the string "HTTP/". These can be used to explicitly set the HTTP response code, as in this example from the PHP manual:

```
<?php
header("HTTP/1.0 404 Not Found");
```

The second special case is for using the "Location" header. This header indicates to the client that the document they are looking for is in the location you specify. PHP will automatically set a 302 HTTP status code if you use this header, unless you've already set a 2xx or 3xx header. Here's an example:

```
<?php
header("Location: http://www.example.com/");
exit;
```

In this example the server will respond with status code 302 and the client will be redirected to the example domain.

Note the usage of the `exit` language construct after sending the redirect header.
Your code continues to run after sending the header unless you stop it.

It is up to the client to respect your redirect header. If they decide not respect it then your code will continue to output and they will see whatever output it generates.

Tracking headers

The `headers_list()` function will return an array of headers that are ready to be sent or have already been sent to the client. You can determine if the headers have been sent by calling `headers_sent()`.

If you want to prevent a particular header from being sent you can use the `headers_remove()` function to unset a header from the list to be sent.

HTTP Authentication

PHP can send a header to the client that causes it to pop up an "Authentication required" dialog box. When the user fills in the dialog with a user and password the URL of the PHP script is called again.

On the second call PHP will have three predefined variables available in the $_SERVER array. These are `PHP_AUTH_USER`, `PHP_AUTH_PW`, and `AUTH_TYPE` and are set to the user name, password and authentication type respectively.

You should then authenticate the user using whatever method you see fit, such as checking the user and password against a database.

Examples of HTTP authentication are given on the PHP manual page[86]:

```
<?php
if (!isset($_SERVER['PHP_AUTH_USER'])) {
    header('WWW-Authenticate: Basic realm="My Realm"');
    header('HTTP/1.0 401 Unauthorized');
    echo 'Text to send if user hits Cancel button';
    exit;
} else {
    echo "<p>Hello {$_SERVER['PHP_AUTH_USER']}.</p>";
    echo "<p>You entered {$_SERVER['PHP_AUTH_PW']} as your password.</p>";
}
```

In this example we just output the contents of the variables in the $_SERVER array, but in real life we would perform some form of authentication.

[86] https://secure.php.net/manual/en/features.http-auth.php

The password sent by the client is base64encoded to standardize the character set but there is no hashing or encryption performed.

HTTP Status Codes

HTTP status codes are sent with responses and follow standards set by the Internet Engineering Task Force as well as de facto standards used within the industry.
The most important ones are the common ones:

Code	Status
200	OK, the request was successful
201	CREATED, the request resulted in a new resource being created
301	Moved permanently, the resource will always be found at the location specified
400	Bad Request – there was something in the request that was malformed or otherwise prevented its execution
401	Unauthorized – the client has not been authenticated as being allowed to make this request
403	Forbidden – The (authenticated) client is not allowed to make this request
418	I'm a teapot. The client is attempting to send coffee making protocols to the server, which is in fact a teapot.[87]
500	Internal Server error. The server was not able to complete the request and can't respond more appropriately. Commonly associated with a crash or misconfiguration.

When using API's the HTTP status code is very important. If you're coding an API you should make sure that you send the correct status code for the error.

For example, if the request failed and you send an error message in the body you should make sure that the HTTP status code is 400 and not 200.

As you work with PHP you'll become more familiar with the status codes but if in doubt you should look up a list and make sure that you're sending an appropriate response[88].

[87] No, this is not a joke and is recognized as a de facto status code
[88] http://httpstatusdogs.com/

Databases and SQL

Databases are the fastest way to persist data that you intend to refer to regularly and need to keep in the long time.

PHP uses extensions to interact with a range of databases. For example to interact with the MySQL database you can use the functions provided by the mysqli extension. Note the "i" at the end of mysqli. This is the replacement for the now deprecated "mysql" extension. Features like "Prepared Statements" are only available with the new extension.

PHP also offers abstraction layers that provide an application layer between your code and the database. We'll be looking at PDO (PHP Data Objects) in this book.

We'll be focusing on relational databases in this book, but it is worth mentioning in passing an alternative to relational databases. MongoDB is a very popular NoSQL database and they have contributed a driver for PHP that allows you to connect to their database. We'll be focusing on the native relational databases and the MongoDB driver is not likely to be included in your Zend examination.

You will be expected to know basic SQL for your Zend examination. The assumed environment will be MySQL if the question does not specify otherwise.

Introduction to Databases

Lets begin by making sure that some of the concepts of relational databases are clear.

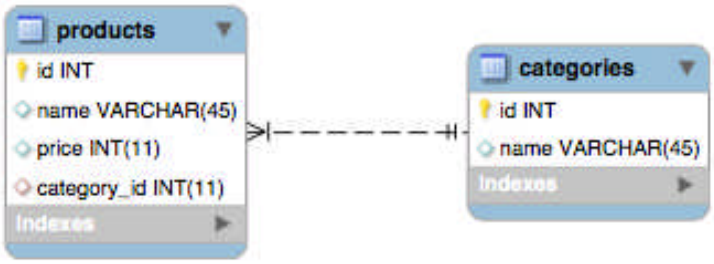
Keys

Keys impose constraints, such as PRIMARY and UNIQUE.

A primary key can be defined on either a single or multiple columns. It guarantees that each row in the database will have a unique value combination for the columns in the key. A row may not have a null value for its primary key.

A table may have only one primary key.

A foreign key can also be defined on either a single or multiple columns. It references the primary key on another table. This is a unique reference, so only one row in the referenced table will be linked to the table containing the foreign key.



In the above diagram we refer to the category that a product belongs to with the foreign key "category_id". Both tables have got primary keys named "id".

Indexes

Indexes are data structures that are needed to implement the key constraint.

Indexes make retrieving records faster. The database engine will create a structure on disk or in memory that contains the data from the indexed columns. This structure is optimized for lookups and helps the database find the row in the table faster.

Whenever you insert a row into a table the indexes need to be updated. This adds an overhead to writing.

You cannot have a key without an index, but it is possible to index columns that are not keyed. You would do this in cases where you don't want to enforce uniqueness but do want to speed up SELECT statements that include these columns in their WHERE clauses.

The binding between keys and indexes is very tight and in MySQL they are considered synonymous.

Relationships

Relationships are a core feature of relational databases. By declaring how tables are related you are able to enforce referential integrity and minimize dirty data.

There are several types of relationships.

Relationship	Description
one-to-one	One row in the parent table can reference exactly one row in the child table
one-to-many	One row in the parent table can be referenced by many rows in the child table
many-to-many	Any amount of rows in the parent table can be referenced by any amount of rows in the child table

By having relationships between tables you are able to store data that is logically related together in a table distinct from other data.

For example we can have a "products" table that stores information about what we sell. Products belong to categories. A single category can have lots of different products in it. This means that one row in the category table can be referred to by multiple rows in the products table.

SQL data types

Columns in a SQL database table have a data type assigned to them. Just as with PHP variable types SQL types can each store different formats of data.

Each database manager will implement the SQL data types slightly differently and will have different optimizations between its types.

We will discuss a few of the common data types and avoid focusing on any particular implementation of SQL.

Numeric types

The types of integers vary in the amount of bytes that they take to store their value.

Integer Type	Bytes	Values
BIGINT	8	-2^63 to +2^63
INTEGER	4	-2^31 to +2^31
SMALLINT	2	-2^15 to +2^15

MySQL allows you to specify a parameter for integers, which is actually a display value and doesn't affect the underlying storage. It's a fairly common misconception that the parameter is for precision.

Non-integer types can be stored in either NUMERIC or DECIMAL values. The SQL-92 standard specifies that a NUMERIC type must have the *exact* precision stipulated, while the DECIMAL type must be *at least* as precise. The implementation of these data types does differ between vendors.

They both take the same parameters:

```
NUMERIC(21,3)
```

The first parameter specifies the total number of digits of precision, and the second parameter specifies how many digits of decimal precision must be stored.

In the example we are going to store a number that has 21 digits in total, of which 3 appear after the decimal point.

Character Types

SQL allows for character to be stored either in fixed or variable length strings.

A fixed length string is always allocated the same number of bytes on disk. This can help speeding up read performance in some database implementations. The trade-off is that if a string being stored in a fixed length data store is shorter than the number of characters allocated, then you are storing more characters than you have to.

Variable length strings can swell up to the limiting size given to them. The database engine allocates storage according to the length of the string. Database implementations will store the length of the string being stored. This will be at least one character to indicate the end of the string, but in some engines each variable string will incur heavier storage overhead.

In general when storing a string that you know is always going to be of a particular length, such as a hash for example, then you should store it in a fixed length character field. This will improve performance and you won't incur storage waste.

Working with SQL

We won't focus on any specific implementation of SQL and will rather try to use generic statements. The Zend examination will not be testing your knowledge of a particular database engine, but will be expecting you to know basic SQL syntax.

Creating a Database and Table

The CREATE statement can be used to create databases and tables. Creating a database is simple, you just specify the name of the database:

```
CREATE DATABASE mydatabase;
```

When creating a table you can specify a list of the columns you want to store in it. For each column you specify the name, data type, and attributes.

```
CREATE TABLE IF NOT EXISTS users (
  id int unsigned NOT NULL AUTO_INCREMENT,
  name varchar(255) NOT NULL,
  email varchar(255) NOT NULL,
  password varchar(60) NOT NULL,
  PRIMARY KEY (id),
  UNIQUE KEY users_email_unique (email)
);
```

Dropping Database and Tables

The inverse of CREATE is the DROP statement.

```
DROP TABLE category;
DROP DATABASE mydatabase;
```

If you have specified foreign keys the database will not let you drop a table if this will violate one of the constraints.

As an example, refer back to our example with products and categories. If we try to drop the category table and there are still products referencing it the database engine should not allow the operation.

Retrieving Data

The SELECT statement is used to retrieve data. The syntax for SELECT can be very complicated and is one of the statements that differs the most between vendors. For your Zend certification you will need to understand basic usage and joins.

In this simple pseudo-code example of a query we retrieve the names of products from our table that cost more than 100 currency units. We specify that we want the results to be returned in descending order of price.

```
SELECT name
    FROM products
    WHERE price > 100
    ORDER BY price DESC
```

You can specify multiple column names separated by commas or use the wildcard * to receive all columns.

The format of the data that PHP receives back is dependent on the driver and function that you use to call the query. You'll generally receive back an object or an array that has keys/properties corresponding to the columns.

Inserting new data

The INSERT statement is used to create new rows in the database. You will need to provide a list of the columns and the values to insert to them. Columns that are marked NOT NULL are mandatory and must have a value specified when you create the row.

```
INSERT INTO products
    (name, price, category_id) VALUES
    ('cheeseburger', 100, 3)
```

If you don't specify the names of the columns SQL will assume that you're providing values in the order that the columns appear in the table. This can be a drawback if ever you change the structure of your table.

Otherwise, as in the example above, you specify the names of the columns, and then the values. The values are assigned to the columns in order. So in our example the name of the product is set to 'cheeseburger', it's price is 100, and it is placed into the category that has an id value of 3 (whatever that may be).

Updating data

The UPDATE statement accepts a list of values similar to the INSERT statement, as well as an optional WHERE clause similar to the SELECT statement.

You must specify what values to update the existing data to, and the criteria for the rows that must be updated.

```
UPDATE products
    SET price = price + 100
    WHERE category_id = 3;
```

Aggregating data

You can use the database to perform calculations and send you the result.

Statement	Returns
AVG	Average value of the data values
SUM	Total of all the data values found
COUNT	How many records were found
DISTINCT COUNT	How many unique records were found
MIN	The lowest value in the data set
MAX	The highest value in the data set

Using these statements is as follows:

```
SELECT AVG(price) FROM products;
```

Grouping data

You can tell SQL to group data by a column or combination of columns before returning it to you. This is often useful in conjunction with the aggregating functions.

Lets take an example where we want to find out the total amount of sales that each of our customers has purchased.

```
SELECT email, SUM( sales_value )
FROM  `transactions`
GROUP BY email
```

In this example we group transactions that have the same email address. The SQL database engine will apply the SUM statement by adding up the sales values in each group and then returning that.

I included the email address in the SELECT statement so that the output will have the email address of the customer, and the sum of all the sales values of transactions with their email address.

Joins

Joins are used to connect tables based on supplied criteria. This lets you retrieve information from related tables.

In our products and categories database you can retrieve the category name of products by joining the category table to the product table:

```
SELECT *
FROM  products
JOIN categories ON categories.id = products.category_id
```

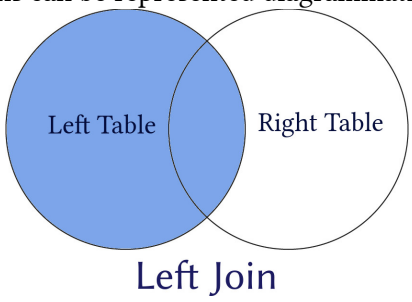
We are joining the category table to the products table and give instructions to SQL on how to match rows. A row from the categories table will be included if its "id" column matches the "category_id" column in the products table.

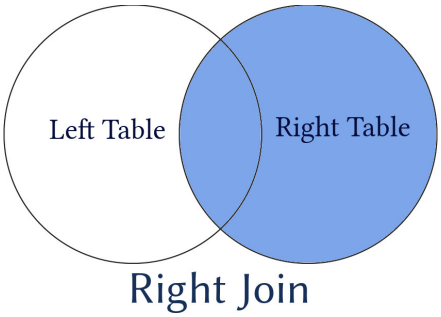
Join Types

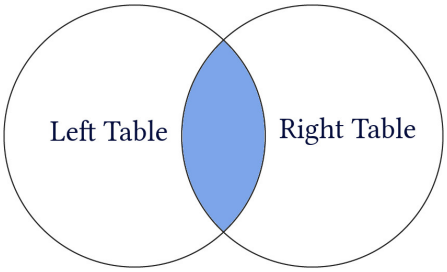
There are several ways to join tables.

Join Type	Effect
INNER JOIN	Selects records that have matching values in both tables, as in the example above
LEFT OUTER JOIN	Select tables from the left table that have matching right table records
RIGHT OUTER JOIN	Select records from the right table that have matching left table records
FULL OUTER JOIN	Select all records that match either left or right table records

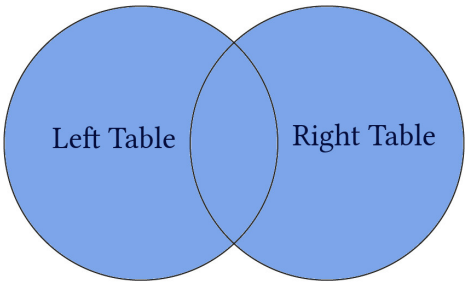
These joins can be represented diagrammatically as follows:







Inner Join



Full Join

Prepared Statements

When you issue a command to a SQL engine it has to parse the command in order to execute it. After the statement has been executed SQL will discard the compiled code with the result that repeated calls with the same SQL command will need to be parsed individually. Obviously this results in duplicated effort.

You can save SQL from having to repeat its efforts by using prepared statements that become parsed code templates that SQL stores for multiple reuse.

Prepared statements also offer significant security advantages. Parameters are bound to the prepared statement, and are not included as part of the code string. This means that it is not possible for your parameters to intrude on the code, which means that you no longer need to worry about escaping code to prevent SQL injection. Just keep in mind the possibility of stored XSS attacks before you stop worrying about escaping data coming into or out of your database.

```
<?php
// prepare and bind
$stmt = $conn->prepare("INSERT INTO users (username, password) VALUES (?, ?)");
$stmt->bind_param("ss", $username, $password);

// set parameters and execute
$username = "bob";
$password = password_hash("password", PASSWORD_BCRYPT);
$stmt->execute();
```

Transactions

A transaction is a set of SQL statements that will either all succeed or else have no effect. After a transaction finishes the database must not have any table constraints invalidated and must be in a state where all the changes have been persisted. A database must have some way of ensuring that transactions can run at the same time and not interfere with each other, for example by incrementing a primary key that another transaction is depending on.

In summary, a transaction is a set of SQL statements that must complete successfully in an “all or nothing” manner. After it runs the database must be in a consistent state and must be recoverable from error.

The syntax for transactions varies between vendors, but there are three important statements.

One statement will mark the beginning of the transaction block. The SQL statements following this will be considered to be part of the transaction.

There are two statements that can end a transaction. One of them will tell SQL to go ahead and make all of the changes that the transaction is making.

The other will tell SQL that for whatever reason you want to abandon the transaction and rather revert back to the state the database was in when the transaction started.

I'll tabulate the three most common vendors' statements:

MySQL	MS-SQL	ORACLE
START TRANSACTION	BEGIN TRANSACTION	START TRANSACTION
COMMIT	COMMIT TRANSACTION	COMMIT
ROLLBACK	ROLLBACK WORK	ROLLBACK

PHP Data Object (PDO)

The PDO is a data abstraction layer that offers a single interface for you to interact with multiple data sources. While using the PDO you can use the same functions to interact with your database no matter the vendor.

It's important to understand that PDO is an access abstraction layer and does not abstract SQL or data types. The SQL that you pass to the `PDO::query()` or prepared statements must be valid for the vendor you are connecting to.

PDO uses database adapters to be able to connect to the database. These adapter classes implement PDO interfaces and expose vendor specific functions as regular extension functions.

PDO is configured in the PHP configuration file. At runtime you can change options with the `PDO::setAttribute()` function.

The PDO extension makes available a number of predefined constants. You won't need to remember them all for the Zend examination, but take a look through the PHP manual[89] and familiarize yourself with them.

The PDO will emulate prepared statements for databases that don't support them, but will otherwise use the native prepared statement functionality of the database.

Connecting to PDO

In order to connect to the database with PDO you create an instance of the PDO class. The constructor accepts parameters for the database source (DSN) and the username/password if these are required.

[89] https://secure.php.net/manual/en/pdo.constants.php

```php
<?php
try {
    $dbh = new PDO('mysql:host=localhost;dbname=test', $user, $pass);
} catch (PDOException $e) {
    echo "Error connecting to database: " . $e->getMessage();
}
```

If there are errors connecting to the database then a `PDOException` will be thrown. It is very important to note that the stack trace of the exception will probably contain the full database connection details. Make sure that you catch it and don't let it be displayed "raw".

To close a connection when you're done with it you can set the pdo variable to null.

```php
$dbh = null;
```

Database connections are automatically closed at the end of your running script unless you make them persistent. Persistent database connections are not closed but are instead cached for another instance of the script to use. This reduces the overhead of needing to connect to the database every time your web application runs.

Transactions in PDO

PDO offers transaction commands too, but does not emulate proper transaction handling. This means that you can only use the PDO transaction functions on databases that natively support transactions. The functions are `PDO::beginTransaction()`, `PDO::commit()`, and `PDO::rollBack()`.

```php
<?php
$dsn = 'mysql:host=localhost;dbname=example';
$pdo = new PDO($dsn, 'dbuser', 'dbpass');
$pdo->setAttribute(PDO::ATTR_EMULATE_PREPARES, TRUE);
$pdo->setAttribute(PDO::ATTR_ERRMODE,
    PDO::ERRMODE_EXCEPTION);
$password = password_hash("password", PASSWORD_BCRYPT);
try {
    $pdo->beginTransaction();
    $pdo->exec("
          INSERT INTO users
              (username, password)
          VALUES
              ('bob', '{$password}'");
    // some more update or insert statements
    $pdo->commit();
} catch (PDOException $e) {
    $pdo->rollBack();
    echo 'Rolled back because: ' . $e->getMessage();
}
```

In the example we make a connection to the database with PDO and start a transaction.

We wrap all of the PDO transaction functions in a try...catch block. If a PDO statement fails to run it will throw a PDOException. We use the catch block to roll back the transaction.

Fetching PDO Results

We use the `PDO::fetch()` method to retrieve data from a PDO result. PDO will maintain a cursor to traverse the result set and uses this to determine which element to return to you.

PDO will return the data to you in a format that you specify in the first parameter to `fetch()`.

Fetch style	Returns
PDO::FETCH_ASSOC	Returns an associative array with your database columns as keys
PDO::FETCH_NUM	Returns an array indexed by column number as returned by your result set
PDO::FETCH_BOTH	Returns an array with both the indexes of ASSOC and NUM style fetches.
PDO::FETCH_BOUND	Returns true and assigns the values of the columns in your result set to the PHP variables to which they were bound with the PDOStatement::bindColumn() method
PDO::FETCH_CLASS	Returns a new instance of the requested class mapping the columns of the result set to named properties in the class
PDO::FETCH_INTO	Updates an existing instance of the requested class, mapping as for FETCH_CLASS
PDO::FETCH_OBJ	Returns an anonymous object with property names that correspond to the column names from your result set
PDO::FETCH_LAZY	Combines PDO::FETCH_BOTH and PDO::FETCH_OBJ and creates the object variable names as they are accessed
PDO::FETCH_NAMED	As for PDO::FETCH_ASSOC returns an associative array. If there are multiple columns with the same name the value referred to by that key will be an array of all the values in the row that had that column name.

Prepared Statements in PDO

Not all database engines support prepared statements and this is the only feature that PDO will emulate for adapters that don't.

The syntax for a prepared statement in PDO is very similar to using a native function.

```
<?php
$stmt = $dbh->prepare("INSERT INTO users (name, email) VALUES (:name,
:value)");
$stmt->bindParam(':name', $name);
$stmt->bindValue(':email','alice@example.com');

// insert one row
$name = 'one';
$stmt->execute();
```

Walking through the example we see that the `prepare()` method is used to create the statement object.

We're using two different forms of binding parameters as a means to demonstrate the different.

In the first, `bindParam()`, we're binding a variable to the statement parameter. When the statement executes the parameter will take the value of the variable at *execution* time.

The second way to bind variables, `bindValue()`, binds a literal to the statement parameter. If you used a variable name in `bindValue()` then the value of the variable at *bind* time is used. Changes to the variable before the statement executes will not affect the parameter value.

> Only values can be bound in a SQL statement, not entities like table names or columns. You can only bind scalar values, not composite variables like arrays or objects.

Repeated Calls to PDO Prepared Statements

We have seen that the `bindParam()` method inserts the value of a variable at the time the statement is executed into the statement parameter. You can see that using `bindParam()` allows you to repeatedly call the prepared statement, using different values for the parameters on each call.

The method `closeCursor()` is used to clear the database cursor and return the statement to a state where it can be executed again. Some databases have problems executing a prepared statement when a previously executed statement still has unfetched rows.