

## Table of Contents

S#	Session	Page #
1.	Session 1: Introduction to PHP	
	• <a href="#">Overview of Web Concepts and Technologies</a>	5
	• <a href="#">PHP HTTP Headers</a>	9
2.	Session 2: Installing and Configuring PHP 7	
	• <a href="#">Install PHP 7 on CentOS, RHEL, or Fedora</a>	13
	• <a href="#">PHP 7.0 as Software Collection</a>	15
	• <a href="#">Adding PHP 7 to your Linux Server</a>	16
	• <a href="#">Your First Page - Hello World!</a>	19
3.	Session 3: New Features of PHP 7	
	• <a href="#">What's New in PHP 7 New Features</a>	21
	• <a href="#">Three-way Comparison</a>	28
	• <a href="#">Generator</a>	31
	• <a href="#">PHP: Stream a File Line-by-line Using a Generator</a>	33
4.	Session 4: Form Handling in PHP	
	• <a href="#">A Forms Primer</a>	34
	• <a href="#">PHP Form Handling</a>	38
5.	Session 5: Using Variables and Expressions in PHP	
	• <a href="#">Variables</a>	45
6.	Session 7: Scalar Type Declarations	
	• <a href="#">Data Types</a>	49
	• <a href="#">PHP Data types</a>	50
	• <a href="#">What's New in PHP 7 New Features: Anonymous Classes</a>	51

S#	Session	Page #
7.	Session 8: PHP Operators	
	• <a href="#">Comparing</a>	53
8.	Session 10: Conditional Statements in PHP	
	• <a href="#">Selecting</a>	56
9.	Session 12: Flow Control in PHP	
	• <a href="#">Loops and Iteration</a>	62
10.	Session 14: Functions in PHP	
	• <a href="#">Functions</a>	69
	• <a href="#">Path and Directory Function in PHP</a>	74
11.	Session 16: Working with Arrays	
	• <a href="#">How to Create Arrays in PHP?</a>	77
12.	Session 18: Handling Databases with PHP	
	• <a href="#">Database Access</a>	82
	• <a href="#">More SQL</a>	88
13.	Session 19: Working with Cookies	
	• <a href="#">Bake Cookies Like a Chef</a>	93
14.	Session 21: Session Management in PHP	
	• <a href="#">Sessions</a>	99
15.	Session 22: Handling E-mail with PHP	
	• <a href="#">Sending Mail Attachment(s) with PHP Mail Function: Rules</a>	102
16.	Session 24: OOP Concepts	
	• <a href="#">Magic Methods in PHP</a>	104

S#	Session	Page #
17.	Session 26: Generator Delegation and Throwable Interfac	
	<ul style="list-style-type: none"><li>• <a href="#">PHP Errors and Exceptions</a></li></ul>	108
	<ul style="list-style-type: none"><li>• <a href="#">Exception Handling Syntax</a></li></ul>	110
	<ul style="list-style-type: none"><li>• <a href="#">Why Throwing Exceptions is Better than Returning Error Codes?</a></li></ul>	112
	<ul style="list-style-type: none"><li>• <a href="#">Ubuntu Tips: How Do you Display PHP Error Messages?</a></li></ul>	124

## Session 1: Introduction to PHP

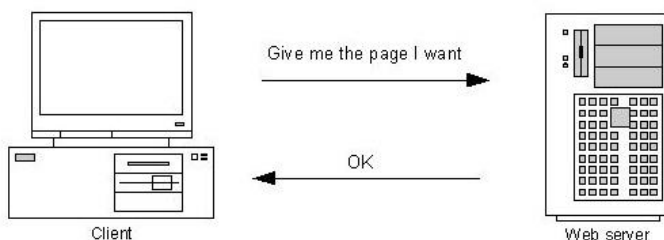
### Overview of Web Concepts and Technologies

<b>Source</b>	<a href="http://interoperating.info/courses/introphp/intro/overview-web-concepts-and-technologies">http://interoperating.info/courses/introphp/intro/overview-web-concepts-and-technologies</a>
<b>Date of Retrieval</b>	03/09/2013

#### Web servers and clients

Because the focus of this class is on using PHP for web programming, it will be useful to briefly go over some very basic concepts about how the web works prior to diving into PHP itself. PHP is just one element in a complex ecology of technologies and protocols that make up the web environment, and it works the way it does in large part because of the niche that it occupies and the other technologies that it has to interact with.

The web wasn't always so complex, of course. Back in the very early days, you could model the typical web transaction something like this:



A client (typically a web browser running on a desktop computer) sends a request for a particular file to a server. The request is formatted in a particular way that allows the server to identify where the file lives in its filesystem. The server returns the file, along with a brief header indicating the status of the request (successful, not successful) and a few other things that might help the client software process the response.

Here's an example of the kind of request a browser might send out to a server. You may have seen something like this before.

<http://www.server.org/greetings/hello.html>

- protocol
- server address
- directory
- file name

To be a bit more precise, this isn't a request, it is of course a URL, which is a way of representing an http request for inclusion in html documents. If the URL is hyperlinked, the browser translates the URL into an http request when the hyperlink is clicked.

You can see that the URL can be chunked into a number of definable parts. The first part specifies the protocol that is to be used to handle the request (the hypertext transfer protocol, or http, the fundamental protocol of the web); the second is the address of the server to which the request is being sent; the third is where the file lives on the server; and the fourth is the name of the file that the client wants to retrieve.

An important thing to note here is that the file being sent back to the client (typically an html document, but possibly a binary file like an image) is a static entity. That works reasonably well for certain kinds of information (reports, essays, articles, pictures of cats) but falls down when it comes to other kinds of data. For example, what if you had a list of people and a list of organizations, and you wanted to show the people sorted alphabetically, and also sorted by organization? Well, you could maintain the data in two separate files. And then someone comes along and asks for the same thing, but this time sorted by email address. Now we're up to 3 separate files containing the same data, and we have the beginnings of a maintenance problem.

So early on it was seen that simply serving up static files was pretty limiting, and some way was needed to produce a more dynamic response to user input. Of course, this first requires that the user be able to send a wider range of input than is available just by clicking on a hyperlink.

One very common way to gather user input is to have the user fill in some fields on a web form, which in turn affixes that input to the end of the http request, in the form of name/value pairs called parameters.

For example, here's a simple web form:

**Please Select a Greeting**

Hello  
Goodbye  
Nice day if it don't rain

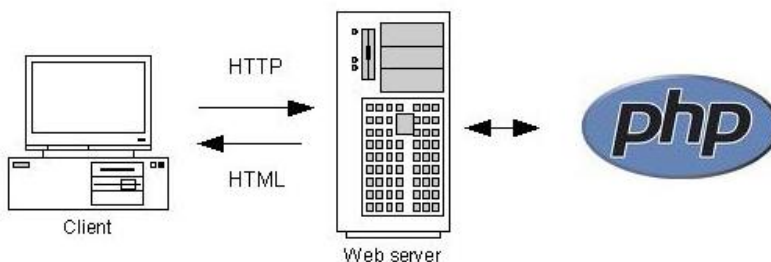
Optional: please tell us your name:

And here's what the http request generated by the form might look like:

<http://www.server.org/greetings/greeting.php?type=hello&name=Bob>

In this new example, the client is still sending a request for a file that lives in a particular location on the web server, but the file is no longer a simple static document. The client is now sending a request for a php script, which will presumably do something clever with the two parameters that now accompany the request.

So we can now visualize the transaction as being more like this:

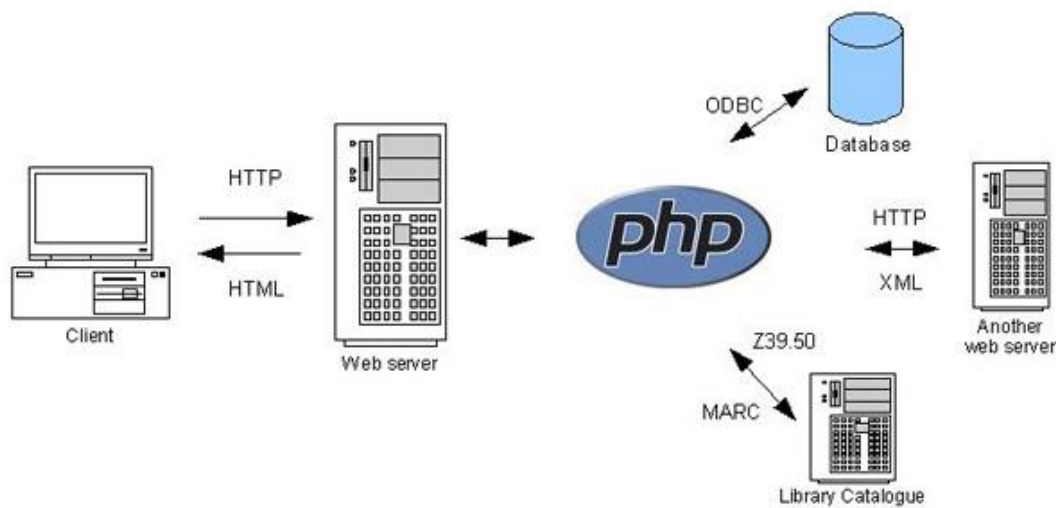


The web server still receives the request, but can no longer process the whole request by itself. Instead, it sees that the request is for a PHP script, so it passes off the processing of the script to the PHP interpreter, which is just another piece of software. The PHP interpreter processes the script (which in turn processes any parameters that have accompanied the request), and returns some output (generally html) to the web server, which in turn passes it back to the client.

Of course, this diagram might be a bit confusing because PHP (the interpreter) typically lives on the same hardware as the web server software. What you need to keep in mind here is that all of these components are really just software, and software can live anywhere, even though I've followed a diagrammatic convention of using hardware icons to

represent two of the 3 components. As you'll soon see, all three components (browser, web server, and php interpreter) can even live together on the same piece of hardware, if you want them to.

That's about as far as we're going to take you in this course, but we'd be remiss if we didn't at least mention that there's yet another level of complexity, wherein PHP in turn interacts with other services and applications before returning output to the web server. That's where things really get interesting ...



~~~ End of Article ~~~





# PHP HTTP Headers

|                          |                                                                                                     |
|--------------------------|-----------------------------------------------------------------------------------------------------|
| <b>Source</b>            | <a href="http://www.techflirt.com/php-http-headers/">http://www.techflirt.com/php-http-headers/</a> |
| <b>Date of Retrieval</b> | 03/09/2013                                                                                          |

If you are a programmer and programming in PHP then you will be familiar with **header()** function of PHP. At least you will be aware with the following type of code `header("Location: b.php")`. to redirect your page to b.php. Basically through the header function you can do a lots of thing very easy and with peace of mind. In my post "php http headers" I will try to explain all aspect of the http headers and php header function . To start with header function implementation first I would like to introduce you with HTTP HEADER concept. Do you know what HTTP Header is? if yes you can skip below section otherwise first please read the concept of HTTP Header written below

## What HTTP HEADER Is?

HTTP header is noting but a set of rule written in textual form. You can say it is the command packet for internet protocol. HTTP HEADER contain information data of your request on server point of view and instruction for browser point of view. Let us take an example. Whenever you typed any URL(website address) in your browser and request any page then following things happens in the background before you get requested page displayed:

- Your request goes with some information like, ip from which you are requesting the page, browser information etc(due to this reason we can access referar or client IP in php through \$\_SERVER array)
- Your above request information goes in the form of HTTP header
- Your request HTTP header is first verified by browser and then return HTML with HTTP HEADER response.

In all above process one thing is common which involve at all place is HTTP Header, which goes while user request the page and comes when response comes from HTTP server. So we can categories HTTP header into two part first is Request HTTP header and another is Response Http Header.

Here is an example of request http header

```
GET /index.html HTTP/1.1
```

```
Host: www.yoursite.com
```

```
From: admin@st.com
User-Agent: Mozilla/4.0 (compatible; MSIE 5.5; Windows NT 4.0)
When webserver send the page it comes with HTTP header response. Following is
an example of response http header
Date: Fri, 27 Jun 2008 00:22:19 GMT
Expires: -1
Cache-Control: private, must-revalidate, max-age=0
Content-Type: text/html; charset=UTF-8
Last-Modified: Fri, 27 Jun 2008 00:22:18 GMT
Etag: 9398183226707537952
Set-Cookie: IGTP=LI=1:LM=1214526139; expires=Sun, 27-Jun-2010 00:22:19 GMT;
path=/ig;
domain=www.google.com
```

So http header is noting but some data and command type information which act as bridge between webserver and web browser. If we submit any html form then data of the form goes through the HTTP header. If you want to redirect the page you submit the location of the page through the HTTP header and page get redirected.

### Back To PHP HTTP HEADER Beauty

Through the php script you can handle all request and response of HTTP header very intelligently and in very easy way. As you have already implemented some of the feature of HTTP header like Location, Cache, content type, there are some other part of story still alive. Let us cover it out:

Aha I forget one thing to inform you: Send HTTP Header always before sending content to your browser otherwise you will get an error of header already sent output started at line xxx. In this case normally programmers take help of ob\_start function and ignore fixing the bug of output. In most of the cases it arise due to space left between php tag or echo within the page. So please take care of that.

Now let us see following example

```
header("Location: page1.php");
header("Location: page2.php");
```

The output will be your browser will display page2.php because first it send HTTP header to the browser that Location: page1.php and then it again override the value of location with

page2.php.

Now look at the following example:

```
header("Location page1.php");  
  
echo "test";  
  
header("Location: page2.php");
```

Now think for the output of above code. It will simply redirect the page to page1.php because on second line of code output started so you can not send http header.

In some of the cases you need to open the pdf or any other file type then the http header for this will be Content-type: mimetype

To open that type of file in most of the cases programmer prefer to open download dialog box. To open download dialog box you just need to do few line

```
<?php  
header('Content-type: image/jpg'); // just specify that page will open jpg  
image file  
header('Content-Disposition: attachment; filename="photo_downloaded.jpg"');  
//for download dialog box  
readfile('original.pdf'); //just read the content and throw it to the browser  
?>
```

So from above code you can even open dialog box for image file also.

These are some simple beauty of http header. Now let us go for some best condition.

Suppose you are creating CMS pages and for search engine compatibility you have made some rewriting so that all request goes to your content.php.. Now suppose user typed any page which are not exists so what you will do. Either you will handle exception and show message that page not displayed. But with the help of http header you can specify any error page and even change your page status with 401 url not found. it is very very simple thing in which you only need to write following code

```
<?php  
header("HTTP/1.0 404 Not Found");  
  
?>
```

In php 5 and above you can setup your own custom http header data and its value Like  
`header("custome-header-name: header value");`

so you can pass header values between different header post back in php. is not it really cool. 😊

As you can set the custom value of http header you can remove the http header value by **header\_remove()** function.

As you can remove and add http header value you can list all header value which is in request or reponse by function **get\_header()** function.

get\_header function return an array of all header keys and value.

As I have previously discuss that if output started then header will not be sent it means that before first output line last header line sent. so you can check this by header\_sent() function. header\_sent() return true if there is any html output started. so before sending any header you can check for header status by header\_sent function. For example you want to send header to redirect the page and also if header is sent then redirect it through the javascript. Following is a code example

```
<?php
function redirect($filename) {
if (!headers_sent())
header('Location: '.$filename);
else {
echo '<script type="text/javascript">';
echo 'window.location.href="'.$filename.'";';
echo '</script>';
echo '<noscript>';
echo '<meta http-equiv="refresh" content="0;url='.$filename.'" />';
echo '</noscript>';
}
}

redirect('http://www.google.com');
?>
```

~~~ End of Article ~~~



## Session 2: Installing and Configuring PHP 7

### Install PHP 7 on CentOS, RHEL, or Fedora

|                   |   |
|-------------------|---|
| Source            | <a href="http://blog.remirepo.net/post/2016/02/14/Install-PHP-7-on-CentOS-RHEL-Fedora">http://blog.remirepo.net/post/2016/02/14/Install-PHP-7-on-CentOS-RHEL-Fedora</a> |
| Date of Retrieval | July 10 2016  |

Here is a quick **howto** upgrade default PHP version provided on Fedora, RHEL or CentOS with latest version **7.0**.

Repositories configuration:

On **Fedora**, standards repositories are enough, on **Enterprise Linux** (RHEL, CentOS) the **Extra Packages for Enterprise Linux** (EPEL) repository must be configured, and on RHEL the **optional** channel must be enabled.

#### Fedora 23

```
wget http://rpms.remirepo.net/fedora/remi-release-23.rpm
dnf install remi-release-23.rpm
```

#### RHEL version 7.2

```
wget https://dl.fedoraproject.org/pub/epel/epel-release-latest-7.noarch.rpm
wget http://rpms.remirepo.net/enterprise/remi-release-7.rpm
rpm -Uvh remi-release-7.rpm epel-release-latest-7.noarch.rpm
subscription-manager repos --enable=rhel-7-server-optional-rpms
```

#### RHEL version 6.7

```
wget https://dl.fedoraproject.org/pub/epel/epel-release-latest-6.noarch.rpm
wget http://rpms.remirepo.net/enterprise/remi-release-6.rpm
rpm -Uvh remi-release-6.rpm epel-release-latest-6.noarch.rpm
rhn-channel --add --channel=rhel-$(uname -i)-server-optional-6
```

#### CentOS version 7.2

```
wget https://dl.fedoraproject.org/pub/epel/epel-release-latest-7.noarch.rpm
wget http://rpms.remirepo.net/enterprise/remi-release-7.rpm
rpm -Uvh remi-release-7.rpm epel-release-latest-7.noarch.rpm
```

#### CentOS version 6.7

```
wget https://dl.fedoraproject.org/pub/epel/epel-release-latest-6.noarch.rpm
wget http://rpms.remirepo.net/enterprise/remi-release-6.rpm
rpm -Uvh remi-release-6.rpm epel-release-latest-6.noarch.rpm
```

#### remi-php70 repository activation

Needed packages are in the **remi-safe** (enabled by default) and **remi-php70** repositories, the latest is not enabled by default (administrator choice according to the desired PHP version).

#### RHEL et CentOS

```
yum-config-manager --enable remi-php70
```

## Fedora

```
dnf config-manager --set-enabled remi-php70
```

## PHP upgrade

By choice, the packages have the same name than in the distribution, so a simple update is enough:

```
yum update
```

That's all :)

```
$ php -v
PHP 7.0.3 (cli) (built: Feb  3 2016 10:09:48) ( NTS )
Copyright (c) 1997-2016 The PHP Group
Zend Engine v3.0.0, Copyright (c) 1998-2016 Zend Technologies
    with Xdebug v2.4.0RC4, Copyright (c) 2002-2016, by Derick Rethans
```



## PHP 7.0 as Software Collection

|                   |   |
|-------------------|---|
| Source            | <a href="http://blog.remirepo.net/post/2015/03/25/PHP-7.0-as-Software-Collection">http://blog.remirepo.net/post/2015/03/25/PHP-7.0-as-Software-Collection</a> |
| Date of Retrieval | July 10 2016  |

PM of upcoming major version of **PHP 7.0**, are available in **remi** repository for **Fedora** 20, 21, 22 and **Enterprise Linux** 6, 7 (RHEL, CentOS, ...) in a fresh new Software Collection (php70) allowing its installation beside the system version.

As I strongly believe in SCL potential to provide a simple way to allow installation of various versions simultaneously, and as I think it is useful to offer this feature to allow developers to test their applications, to allow sysadmin to prepare a migration or simply to use this version for some specific application, I decide to create this new SCL.

### Installation :

```
yum install php70
```

To be noticed:

- the SCL is independant from the system, and doesn't alter it
- this SCL is available in **remi-safe** repository
- installation is under the **/opt/remi** tree
- the **Apache** module, php70-php, is available, but of course, only one *mod\_php* can be used (so you have to disable or uninstall any other, the one provided by the default "php" package still have priority)
- the **FPM** service (php70-php-fpm) is available, it listens on default port 9000, so you have to change the configuration if you want to use various FPM services simultaneously.
- the **php70** command give a simple access to this new version, however the **scl** command is still the recommended way (or the **module** command).
- for now, the collection provides **7.0.0-dev**, but alpha/beta version should be released soon
- more PECL extensions will be progressively also available
- only **x86\_64**, no plan for other arch.

```
$ scl enable php70 'php -v'
PHP 7.0.0-dev (cli) (built: Mar 25 2015 14:40:01)
Copyright (c) 1997-2015 The PHP Group
Zend Engine v3.0.0-dev, Copyright (c) 1998-2015 Zend Technologies
    with Zend OPcache v7.0.4-dev, Copyright (c) 1999-2015, by Zend
Technologies
```



## Adding PHP 7 to your Linux Server

|                          |   |
|--------------------------|---|
| <b>Source</b>            | <a href="https://waltereibert.com/blog/adding-php7-to-your-linux-server/">https://waltereibert.com/blog/adding-php7-to-your-linux-server/</a> |
| <b>Date of Retrieval</b> | 10/07/2016  |

PHP 7 has been released for some time now and it seems pretty solid. Besides new features like return type hints, it is up to 3x faster than PHP 5.6. Luckily, for most Linux distributions there are PHP 7 repositories available.

### Ubuntu

On Ubuntu 14.04 and later you have to ensure you can add a Personal Package Archives (PPA):

```
sudo apt-get install software-properties-common
```

**[update]** The PPA has changed. You need both `ppa:ondrej/php` and `ppa:ondrej/php-qa`.

Add the repositories and install PHP7:

```
sudo add-apt-repository ppa:ondrej/php
sudo add-apt-repository ppa:ondrej/php-qa
sudo apt-get update
sudo apt-get install libapache2-mod-php7.0 php7.0-fpm php7.0-common php7.0-
cli php-pear php7.0-curl php7.0-gd php7.0-gmp php7.0-intl php7.0-imap php7.0-
json php7.0-ldap php7.0-mbstring php7.0-mcrypt php7.0-mysql php7.0-ps php7.0-
readline php7.0-tidy php7.0-xmlrpc php7.0-xsl
sudo apt-get --purge autoremove
```

On older Ubuntu versions, you will need to install `python-software-properties`:

```
sudo apt-get install python-software-properties
```

### Debian

Add the Dotdeb repository to `/etc/apt/sources.list`:

```
deb http://packages.dotdeb.org jessie all
deb-src http://packages.dotdeb.org jessie all
```



Fetch and install the GnuPG key:

```
wget https://www.dotdeb.org/dotdeb.gpg
sudo apt-key add dotdeb.gpg
```

Install the PHP7 packages:

```
sudo apt-get update
sudo apt-get install libapache2-mod-php7.0 php7.0-fpm php7.0-common php7.0-
cli php-pear php7.0-curl php7.0-gd php7.0-gmp php7.0-intl php7.0-imap php7.0-
json php7.0-ldap php7.0-mbstring php7.0-mcrypt php7.0-mysql php7.0-ps php7.0-
readline php7.0-tidy php7.0-xmlrpc php7.0-xsl
sudo apt-get --purge autoremove
```

## CentOS

Add the PHP 7 repository (on CentOS 7):

```
sudo yum -y install epel-release
wget http://rpms.famillecollet.com/enterprise/remi-release-7.rpm
sudo rpm -Uvh remi-release-7*.rpm
```

Then install PHP 7:

```
sudo yum -y --enablerepo=yum --enablerepo=remi,remi-php70 install php-fpm
php-common php-cli php-pear php-pdo php-mysqlnd php-gd php-mbstring php-
mcrypt php-xml php-iconv php-soap php-opcache php-tidy php-json php-openssl
php-bcmath
scl enable php70 'php -v'
```

## openSUSE

Add the PHP 7 repository (on version 42.1):

```
sudo zypper ar
http://download.opensuse.org/repositories/devel:/languages:/php:/php7/openSUS
E_Leap_42.1/devel:/languages:/php:/php7.repo
```

Then install PHP 7:

```
sudo zypper refresh
```

```
sudo zypper install php7 php7-bcmath php7-bz2 php7-calendar php7-ctype php7-curl php7-devel php7-dom php7-enchant php7-exif php7-fileinfo php7-fpm php7-ftp php7-gd php7-gettext php7-gmp php7-iconv php7-imap php7-intl php7-json php7-ldap php7-mbstring php7-mcrypt php7-mysql php7-opcache php7-openssl php7-pdo php7-pear php7-phar php7-posix php7-pspell php7-readline php7-soap php7-tidy php7-tokenizer php7-wddx php7-xmlreader php7-xmlrpc php7-xmlwriter php7-xsl php7-zip php7-zlib
```

## Docker

Start Docker:

```
docker run -p 80:80 -it --rm --name my-apache-php-app -v "$PWD":/var/www/html php:7-apache
```

## Compile

Of course you can always build from source. You will have to install the required build tools. On Ubuntu this would be:

```
sudo apt-get install build-essential automake autoconf gcc libtool binutils libxml2-dev libcurl4-openssl-dev libfreetype6-dev libjpeg-dev libpng12-dev libbz2-dev libmcrypt-dev libmm-dev libxslt1-dev
```

Then compile PHP:

```
./configure \
--enable-fpm \
--with-gettext --enable-mbstring \
--with-mcrypt \
--enable-soap --with-xsl \
--with-bz2 --enable-zip --with-curl --with-openssl \
--with-gd --with-jpeg-dir=/usr --with-png-dir=/usr --with-zlib-dir=/usr --
with-freetype-dir=/usr --enable-exif \
--with-mysqli=mysqli --with-pdo-mysql=mysqli \
--with-mm=/usr
make
sudo make install
```

~~~End of Article~~~



# Your First Page - Hello World!

|                          |                                                                                                                                                                                   |
|--------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Source</b>            | <a href="https://courses.p2pu.org/en/groups/learn-php/content/your-first-page-hello-world/">https://courses.p2pu.org/en/groups/learn-php/content/your-first-page-hello-world/</a> |
| <b>Date of Retrieval</b> | 10/07/2016                                                                                                                                                                        |

Now that you know the PHP syntax, let's look at actually using it. Before we go on, it's important to know that PHP can be used in any part of any page. Before any HTML is processed, after your `</body>` tag, and everything in between.

Let's make our first page; Hello World.

```
<html>
<body>
<?
    # This will display your text
    echo "Hello World!";
?>
</body>
</html>
```

## Looks like:

Hello World!

Now to break this down:

- Open HTML and BODY tags
- Open syntax - allows us to use PHP
- When you call on the "#", it creates a comment. Comments are not useful for coding, but they are great for leaving notes for yourself or others, and debugging.
- "echo" is the command (referred to as a function), and will display the text between the quotes. Open quotation, text, close quotation, semicolon. The semicolon means that's the end of that statement - finished and PHP will look for the next command to execute.
- Close syntax
- Close HTML and BODY tags

## Comments:

There are 3 types of comments, but they all do the exact same thing.

- # - an entire line for a comment
- // - last part of the line comment
- /\* comment \*/ - block comment

Examples of these look like this:

```
<?
# this will cover the entire line
echo "Test"; // this is typically used after something important
/* and this will
cover everything
in this section */
?>
```

### Semicolins:

After your line of code, or statement (Except for comments), always use a semicolon. NEVER forget them!

### echo function:

the echo function is the same as print in many other languages. I recommend using echo because it's one character shorter. 20% faster to load, 20% less typing.

Anytime you want to display some sort of text, you use the echo function. Your text can be enclosed by double quotations or single. If it started with double, for example, it has to end with double.

example:

```
<?
echo "1. this will show up <br />"; // echo also accepts html
echo '2. this will show up <br />'; // single quotes work
echo "3. this will NOT work <br />"; /* starting with a double quote, ending
with a single is erroneous */
echo '4. this won't work either, can you figure out why?';
?>
```

The first two lines work flawlessly.

The last two will not work. Line #3 starts with a " but ends with a ' - which will cause a "Parse error:"

**Parse error:** syntax error, unexpected \$end in **/home/client/public\_html/directory/echo.php** on line **6**



## Session 3: New Features of PHP 7

### What's New in PHP 7 New Features

|                          |                                                                                                                                                             |
|--------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Source</b>            | <a href="http://www.codedoodle.com/2016/01/whats-new-in-php-7-new-features.html">http://www.codedoodle.com/2016/01/whats-new-in-php-7-new-features.html</a> |
| <b>Date of Retrieval</b> | 10/07/2016                                                                                                                                                  |

For years PHP has evolved with lot's of improvements, fixes and new features. PHP 7 has been released with many improvements and new features. With the improved new PHP you can write better and clean code that runs faster. Here are some of the new features that has been released with PHP 7.

#### Scalar Typehints

Prior to PHP 7 scalar type-hints are not supported. Using scalar type hints you can enforce functions and methods to accept only values of that particular types, by this way we know for sure that the variable will be of the specified type and lot's of code can be avoided.

#### Syntax

```
function name (type $variable, type $variable) {...}
```

#### Example: php7-scalar-typehints.php

```
interface PHP7InterfaceTest {  
  
}  
  
class PHP7ClassTest implements PHP7InterfaceTest {  
  
}  
  
$arr = ['one', 'two', 'three'];  
  
function php7_typehint_test(  
    int $integer,  
    bool $boolean,  
    string $string,  
    float $float,  
    array $arr,  
    PHP7InterfaceTest $interface,  
    PHP7ClassTest $class) {
```

```
// Inside here you can be sure all the values passed will
// be of it's respective types
var_dump(func_get_args());
}

// this will pass

php7_typehint_test(3, true, 'A string', 3.333, $arr, new PHP7ClassTest, new
PHP7ClassTest);
```

**Correct output**

```
array(7) {
    [0]=>
    int(3)
    [1]=>
    bool(true)
    [2]=>
    string(8) "A string"
    [3]=>
    float(3.333)
    [4]=>
    array(3) {
        [0]=>
        string(3) "one"
        [1]=>
        string(3) "two"
        [2]=>
        string(5) "three"
    }
    [5]=>
    object(PHP7ClassTest)#1 (0) {
}
```

```
[6]=>

object(PHP7ClassTest)#2 (0) {

}

}
```

### Integer TypeHint Error

```
// integer typehint fail check

php7_typehint_test('thisWillFail', true, 'A string', 3.333, new
PHP7ClassTest, new PHP7ClassTest);
```

### Error Output

PHP Fatal error: Uncaught TypeError: Argument 1 passed to php7\_typehint\_test() must be of the type integer, string given, called in php7-scalar-typehints on line 26 and defined in php7-scalar-typehints:9

Stack trace:

```
#0 php7-scalar-typehints(26): php7_typehint_test('thisWillFail', true, 'A
string', 3.333, Object(PHP7ClassTest), Object(PHP7ClassTest))
```

```
#1 {main}
```

thrown in php7-scalar-typehints on line 9

### Interface implementation errors

```
// interface typehint check

php7_typehint_test(3, true, 'A string', 3.333, new stdClass, new
PHP7ClassTest);
```

### Error Output

PHP Fatal error: Uncaught TypeError: Argument 5 passed to php7\_typehint\_test() must implement interface PHP7InterfaceTest, instance of stdClass given, called in php7-scalar-typehints on line 28 and defined in php7-scalar-typehints:9

Stack trace:

```
#0 php7-scalar-typehints(28): php7_typehint_test(3, true, 'A string', 3.333,
Object(stdClass), Object(PHP7ClassTest))
```

```
#1 {main}
```

## Return Type Declarations

PHP 7 supports the return type declarations with which we can specify the type of value that will be returned from the function. Using return type hint's it is easy to identify the return type and we can be sure that the particular type specified will be returned as a result of that function or method.

### Syntax

```
function name (type $variable, type $variable): return_type
{
    ...
}
```

### Example: php7-return-type-declaration.php

```
// modified the function in the previous example
// .....

    PHP7ClassTest $class

): array { // specify the return type of the function

    // Inside here you can be sure all the values passed will
    // be of it's respective types

    return func_get_args();
}

// .....

```

If the function does not return the specified type, then an `Uncaught TypeError` will be thrown.

## Null coalescing operator (??)

The Null coalescing operator (??) is a syntactic sugar for the ternary operator (?:). If the first operand exists it returns it, if not, it returns the second operand.

### Syntax

```
$resulting_value = $operand1 ?? $operand2;
```

### Example: php7-null-coalescing.php

```
$null_value = null;
```



```
$not_a_null_value = 'Hi, i am returned, since i am not a null value';

// instead of using this

// echo $null_value ? $null_value : 'Hi, i am the default value. The variable
passed is null' . PHP_EOL;

// the following can be used.

echo $null_value ?? 'Hi, i am the default value. The variable passed is null'
. PHP_EOL;

echo $not_a_null_value ?? 'Hi, i am the default value. The variable passed is
null';
```

### Output

```
Hi, i am the default value. The variable passed is null

Hi, i am returned, since i am not a null
```

## Spaceship operator (<=>)

The spaceship operator is used to compare two expressions. It returns 0, -1, 1 when the first expression is equal, less than or greater than the second expression respectively.

### Syntax

```
expression_one <=> expression_two
```

### Example: php7-spaceship-operator.php

```
$value = 1;

$compare = 1;

// When both expression of equal.

var_dump($value <=> $compare);

$compare = 2;

// When first expression is lesser.

var_dump($value <=> $compare);

$compare = 0;

// When first expression is greater.

var_dump($value <=> $compare);

// Examples of other types and expressions
```

```
var_dump(1+2 <=> 2*3); // Will output int(-1)

var_dump('testString' <=> 'test'); // Will output int(1)

var_dump(new stdClass() <=> new stdClass()); // Will output int(0)
```

### Output

```
int(0)

int(-1)

int(1)

int(-1)

int(1)

int(0)
```

### Namespace use grouping

This is one of my favourite feature in PHP 7. You can now avoid multiple namespace use statements that looks very awful when there are lot's of classes to be included from a single namespace.

With namespace grouping, multiple classes from a single namespace can be grouped and included.

### Syntax

```
use some/namespace/{class1, class2, class3 as C3};
```

### Example

```
namespace Codedoodle\Classes;

interface Class1 {

}

interface Class2 {

}

interface Class3 {

}

namespace Codedoodle\Posts;

use Codedoodle\Classes\{Class1, Class2, Class3 as C3};

class App implements Class1, Class2, C3 {

}
```

Above example shows how to group classes from the `Codedoodle\Classes` namespace into the `Codedoodle\Posts` namespace.

*~~~ End of Article ~~~*



## Three-way Comparison

|                   |                                                                                                                     |
|-------------------|---------------------------------------------------------------------------------------------------------------------|
| Source            | <a href="https://en.wikipedia.org/wiki/Three-way_comparison">https://en.wikipedia.org/wiki/Three-way_comparison</a> |
| Date of Retrieval | 10/07/2016                                                                                                          |

In computer science, a three-way comparison takes two values A and B belonging to a type with a total order and determines whether  $A < B$ ,  $A = B$ , or  $A > B$  in a single operation, in accordance with the mathematical law of trichotomy.

### Machine-level computation

Many processors have instruction sets that support such an operation on primitive types. Some machines have signed integers based on a sign-and-magnitude or one's complement representation (see signed number representations), both of which allow a differentiated positive and negative zero. This does not violate trichotomy as long as a consistent total order is adopted: either  $-0 = +0$  or  $-0 < +0$  is valid. Common floating point types, however, have an exception to trichotomy: there is a special value "NaN" (Not a Number) such that  $x < \text{NaN}$ ,  $x > \text{NaN}$ , and  $x = \text{NaN}$  are all false for all floating-point values x (including NaN itself).

### High-level languages

In C, the functions `strcmp` and `memcmp` perform a three-way comparison between strings and memory buffers, respectively. They return a negative number when the first argument is lexicographically smaller than the second, zero when the arguments are equal, and a positive number otherwise. This convention of returning the "sign of the difference" is extended to arbitrary comparison functions by the standard sorting function `qsort`, which takes a comparison function as an argument and requires it to abide by it.

In Perl (for numeric comparisons only), PHP (since version 7), Ruby, and Groovy, the spaceship operator `<=>` returns the values  $-1$ ,  $0$ , or  $1$  depending on whether  $A < B$ ,  $A = B$ , or  $A > B$ , respectively. In Python 2.x (removed in 3.x), the `cmp` function computes the same thing. In OCaml, the `compare` function computes the same thing. In the Haskell standard library, the three-way comparison function `compare` is defined for all types in the `Ord` class; it returns type `Ordering`, whose values are `LT` (less than), `EQ` (equal), and `GT` (greater than):<sup>[1]</sup>

```
data Ordering = LT | EQ | GT
```

Many object-oriented languages have a three-way comparison method, which performs a three-way comparison between the object and another given object. For example, in Java, any class that implements the `Comparable` interface has a `compareTo` method which returns a negative integer, zero, or a positive integer. Similarly, in the .NET Framework, any class that implements the `IComparable` interface has such a `CompareTo` method.

Since Java version 1.5, the same can be computed using the `Math.signum` static method if the difference can be known without computational problems such as arithmetic overflow mentioned below. Many computer languages allow the definition of functions so a `compare(A,B)` could be devised appropriately, but the question is whether or not its internal definition can employ some sort of three-way syntax or else must fall back on repeated tests.

When implementing a three-way comparison where a three-way comparison operator or method is not already available, it is common to combine two comparisons, such as  $A = B$  and  $A < B$ , or  $A < B$  and  $A > B$ . In principle, a compiler might deduce that these two expressions could be replaced by only one comparison followed by multiple tests of the result, but this is not a common optimization.[vague]

In some cases, three-way comparison can be simulated by subtracting  $A$  and  $B$  and examining the sign of the result, exploiting special instructions for examining the sign of a number. However, this requires the type of  $A$  and  $B$  to have a well-defined difference. Fixed-width signed integers may overflow when they are subtracted, floating-point numbers have the value NaN with undefined sign, and character strings have no difference function corresponding to their total order. At the machine level, overflow is typically tracked and can be used to determine order after subtraction, but this information is not usually available to higher-level languages.

In one case of a three-way conditional provided by the programming language, Fortran's now-deprecated three-way arithmetic IF statement considers the sign of an arithmetic expression and offers three labels to jump to according to the sign of the result:

IF (expression) negative,zero,positive

The common library function `strcmp` in C and related languages is a three-way lexicographic comparison of strings; however, these languages lack a general three-way comparison of other data types.

## Composite data types

Three-way comparisons have the property of being easy to compose and build lexicographic comparisons of non-primitive data types, unlike two-way comparisons.

Here is a composition example in Perl.

```
sub compare($$) {
    my ($a, $b) = @_;
    return $a->{unit} cmp $b->{unit}
        || $a->{rank} <=> $b->{rank}
        || $a->{name} cmp $b->{name};
}
```

Note that `cmp`, in Perl, is for strings, since `<=>` is for numbers. Two-way equivalents tend to be less compact but not necessarily less legible. The above takes advantage of short-circuit evaluation of the `||` operator, and the fact that 0 is considered false in Perl. As a result, if the first comparison is equal (thus evaluates to 0), it will "fall through" to the second comparison, and so on, until it finds one that is non-zero, or until it reaches the end.

In some languages, including Python, Ruby, Haskell, etc., comparison of lists are done lexicographically, which means that it is possible to build a chain of comparisons like the above example by putting the values into lists in the order desired; for example, in Ruby:

```
[a.unit, a.rank, a.name] <=> [b.unit, b.rank, b.name]
```

## Trivia

The three-way comparison operator for numbers is spelled `<=>` in Perl, Ruby, Groovy and PHP, and is called the spaceship operator because it reminded Randal L. Schwartz of the spaceship in an HP BASIC Star Trek game.

*~~~ End of Article ~~~*



# Generator

|                          |                                                                                                                                             |
|--------------------------|---------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Source</b>            | <a href="https://en.wikipedia.org/wiki/Generator_(computer_programming)">https://en.wikipedia.org/wiki/Generator_(computer_programming)</a> |
| <b>Date of Retrieval</b> | 10/07/2016                                                                                                                                  |

In computer science, a generator is a special routine that can be used to control the iteration behaviour of a loop. In fact, all generators are iterators. A generator is very similar to a function that returns an array, in that a generator has parameters, can be called, and generates a sequence of values. However, instead of building an array containing all the values and returning them all at once, a generator yields the values one at a time, which requires less memory and allows the caller to get started processing the first few values immediately. In short, a generator looks like a function but behaves like an iterator.

Generators can be implemented in terms of more expressive control flow constructs, such as coroutines or first-class continuations. Generators, also known as semicoroutines, are a special case of (and weaker than) coroutines, in that they always yield control back to the caller (when passing a value back), rather than specifying a coroutine to jump to; see comparison of coroutines with generators.

## Uses

Generators are usually invoked inside loops. The first time that a generator invocation is reached in a loop, an iterator object is created that encapsulates the state of the generator routine at its beginning, with arguments bound to the corresponding parameters. The generator's body is then executed in the context of that iterator until a special yield action is encountered; at that time, the value provided with the yield action is used as the value of the invocation expression. The next time the same generator invocation is reached in a subsequent iteration, the execution of the generator's body is resumed after the yield action, until yet another yield action is encountered. In addition to the yield action, execution of the generator body can also be terminated by a finish action, at which time the innermost loop enclosing the generator invocation is terminated. In more complicated situations, a generator may be used manually outside of a loop to create an iterator, which can then be used in various ways.

Because generators compute their yielded values only on demand, they are useful for representing streams, such as sequences that would be expensive or impossible to compute at once. These include e.g. infinite sequences and live data streams.

When eager evaluation is desirable (primarily when the sequence is finite, as otherwise evaluation will never terminate), one can either convert to a list, or use a parallel construction that creates a list instead of a generator. For example, in Python a generator `g` can be evaluated to a list `l` via `l = list(g)`, while in F# the sequence expression `seq { ... }` evaluates lazily (a generator or sequence) but `[ ... ]` evaluates eagerly (a list).

In the presence of generators, loop constructs of a language – such as `for` and `while` – can be reduced into a single loop `... end loop` construct; all the usual loop constructs can then be comfortably simulated by using suitable generators in the right way. For example, a ranged loop like `for x = 1 to 10` can be implemented as iteration through a generator, as in Python's `for x in xrange(10, 1)`. Further, `break` can be implemented as sending `finish` to the generator and then using `continue` in the loop.

## PHP

The community of PHP implemented generators in PHP 5.5. Details can be found in the original RFC about Generator.

```
function fibonacci() {  
    $last = 0;  
    $current = 1;  
    yield 1;  
    while (true) {  
        $current = $last + $current;  
        $last = $current - $last;  
        yield $current;  
    }  
}  
  
foreach (fibonacci() as $number) {  
    echo $number, "\n";  
}
```

*~~~ End of Article ~~~*



## PHP: Stream a File Line-by-line Using a Generator

|               |                                                                                                                                                                                           |
|---------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Source</b> | <a href="http://www.geekality.net/2016/02/28/php-stream-a-file-line-by-line-using-a-generator/">http://www.geekality.net/2016/02/28/php-stream-a-file-line-by-line-using-a-generator/</a> |
|---------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

This function will “stream” a file line-by-line, as a Generator.

Can be very useful if for example you need to process a big file, don't want to read the whole thing into memory, but only process each line by itself.

```
function read($file)
{
    $fp = fopen($file, 'rb');

    while(($line = fgets($fp)) !== false)
        yield rtrim($line, "\r\n");

    fclose($fp);
}

// Usage
foreach(read('http://example.com') as $line)
{
    var_dump($line);
}
```

~~~ End of Article ~~~



## Session 4: Form Handling in PHP

### A Forms Primer

|                   |   |
|-------------------|---|
| Source            | <a href="http://www.dcu.ie/~costelle/?q=node/25">http://www.dcu.ie/~costelle/?q=node/25</a> |
| Date of Retrieval | 03/09/2013  |

An HTML form allows users to enter information into a web page. For this reason several of the HTML elements in an HTML form are called *input* elements. Each input element is distinguished by its *type* attribute. An important HTML element of most forms is the button to submit the form, which is an *input* element with *type* value of *submit*, and a *value* attribute that sets the text that appears on the button to the user:

```
<input type="submit" value="submit this form!">
```

All form elements (sometimes called controls) are contained in a parent form element. The form element contains two important attributes: *action* and *method*. The *action* attribute specifies a program that the information in the form will be sent to. For instance in our case here this will be the url to a PHP page. The *method* can have a value of either *get* or *post* and specifies how the data should be sent to the receiving url which was specified in *action*. In most cases (and in the examples here) you will be using *post* as the value of this attribute.

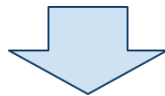
Here is an example of a basic form:

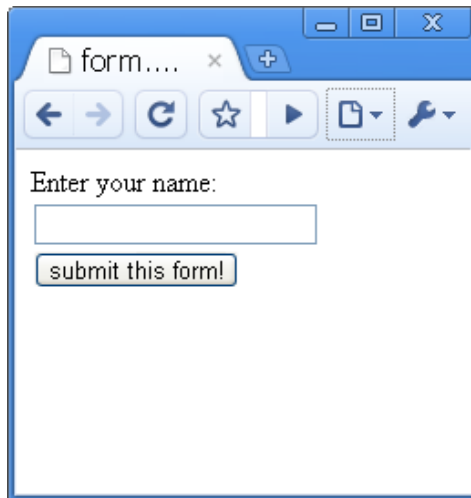
```
<form action="submitForm.php" method="post">

    Enter your name: <input type="text" name="userName"></input>

    <input type="submit" value="submit this form!">

</form>
```





When the user submits the form a POST request will be made for **submitForm.php** and this will contain whatever the user typed in the input text box as the named variable **userName**.

We have introduced a new form control here. Again, it is an input element, like the submit button, but this time it has a *type="text"* attribute (instead of *"type=submit"*). It has another important attribute called name (*name="userName"*). We give each element that accepts user input a name in a form. This way we can get easily retrieve all information entered in the form later on.

Some other useful form elements are:

- textarea - a large textbox
- radio input - User must click one option from a list
- select - User selects an option from a drop-down list (or multiple selectable options)
- password input - the same as text input except that input characters are masked

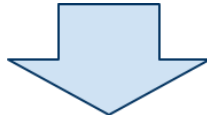
Below is an example of each of these. Note that many (but not all) form elements use the same **input** tag but with different **type** attributes:

```
<form action="submitForm.php" method="post">
Title:      <select name="userTitle">
<option name="Mr">Mr</option>
<option name="Mr">Ms</option>
<option name="Mr">Dr</option>
</select><br>

Name: <input type="text" name="userName"></input> <br>
Password: <input type="password" name="userPassword"></input> <br>

Your story: <br><textarea name="userStory"></textarea><br>
```

```
I have read the 250 pages of terms and conditions<br>
<input type="radio" name="termsAgree" value="yes"> yes
<input type="radio" name="termsAgree" value="no"> no
<br>
    <input type="submit" value="submit this form!">
</form>
```



Note that each radio input must have the same **name** attribute so that they are grouped together. Also note that we give everything a **name** and where a user does not input text a value also. This allows us to retrieve any submitted information by name later. Note the very different forms of the radio input and the select element when in fact they do almost the same thing functionally; also note that textarea is not an **input** tag with a type of **textarea** as you might expect but a tag named **textarea**. This is one of the annoying things about HTML, it is messier than it could be. But, forms are still a pretty easy way to create menu-driven UIs, and powerful when we pass form information to a server-side program such as PHP.

There is much more to learn about forms. To name but a few important things:

- The label element
- The checkbox input
- The file upload input
- The button input
- The **checked** or **selected** attribute of some form controls

Moreover, there are many new exciting form elements on the way in HTML 5.

*~~~ End of Article ~~~*



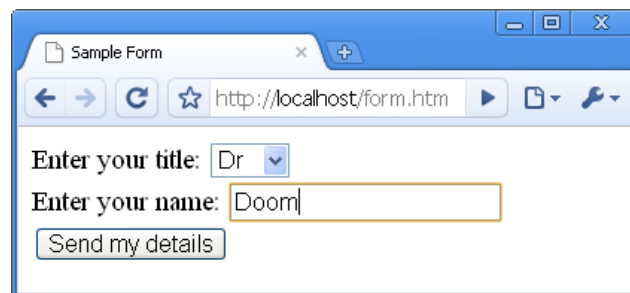
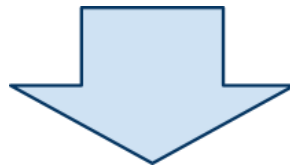
## PHP Form Handling

|                   |   |
|-------------------|---|
| Source            | <a href="http://www.dcu.ie/~costelle/?q=node/15">http://www.dcu.ie/~costelle/?q=node/15</a> |
| Date of Retrieval | 03/09/2013  |

Using GET and appending values to the url is an easy way of passing information into a PHP page. Another way to do this is by using POST instead of GET. With POST information is not sent in the url query string, but instead from an html form in a web page:

```
<html>

    <head>
        <title>Sample Form</title>
    </head>
<body>
    <form method="post" action="submit.php">
        Enter your title:
        <select name="title"></input>
            <option name="Mr">Mr</option>
            <option name="Mrs">Mrs</option>
            <option name="Ms">Ms</option>
            <option name="Dr">Dr</option>
        </select>
        <br>
        Enter your name:
        <input name="name" type="text"></input><br>
        <input type="submit" value="Send my details"></input>
    </form>
</body>
</html>
```

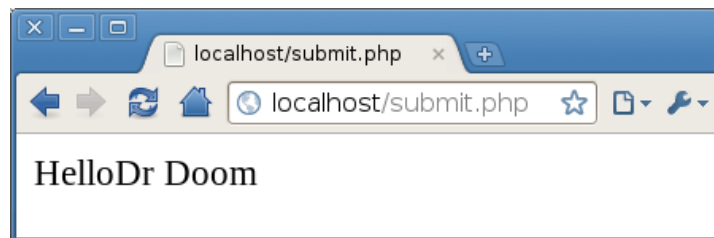
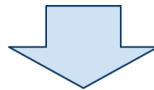






```
<?php

    echo "Hello" , $_POST['title'], " ", $_POST['name'];

?>
```



Each element in the form is passed to **submit.php** and goes into the **\$\_POST** array. The name attribute of the form element becomes the key to access that value in the **\$\_POST** array:

|  |   |                               |
|--|---|-------------------------------|
| <code>&lt;select name="title"&gt;</code>           |  | <code>\$_POST["title"]</code> |
| <code>&lt;input name="name" type="text"&gt;</code> |  | <code>\$_POST["name"]</code>  |

We specified two important things in the HTML form - that we wanted to create a http **post** request, and that we wanted to call the **submit.php** page:

```
<form method="post" action="submit.php">
```

In this example we had two pages, a html page containing a form which calls a php page when submitted. We could do this all with one php page instead. We can put the form in the **submit.php** page itself and have the form call the same page when it is submitted. In this case we just need to check if the form has been submitted, and if it has we display the submitted information:

```
<html>
<head>
<title>Sample Form</title>
</head>
<body>
<?php
    //Check if the form has been submitted

    if ($_SERVER['REQUEST_METHOD'] == 'POST'){
```

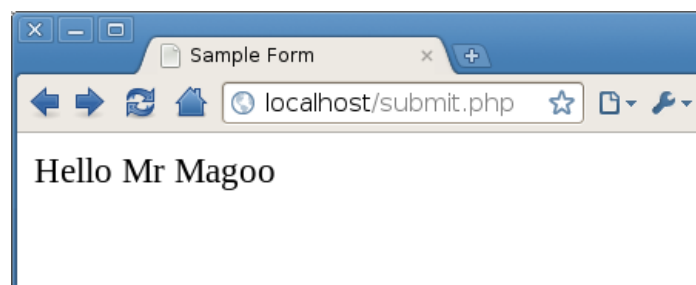
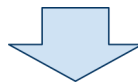
```
//display the information that was submitted in the form
echo "Hello" , $_POST['title'], " ", $_POST['name'];
}else{

    //the form has not been submitted so display the form instead.
?>
<form method="post" action="submit.php">

    Enter your title:
    <select name="title"></input>
        <option name="Mr">Mr</option>
        <option name="Mrs">Mrs</option>
        <option name="Ms">Ms</option>
        <option name="Dr">Dr</option>
    </select>
    <br>
    Enter your name:
    <input name="name" type="text"></input><br>
    <input type="submit" value="Send my details"></input>

</form>
<?php    }    ?>

</body>
</html>
```



If we have all the code in the one page we can use a nice piece of code for the value of the form's action attribute: `$_SERVER['PHP_SELF']`. This code retrieves the value of `'PHP_SELF'` from the `$_SERVER` array. This value is the name of the PHP page itself. So, if our page is called **submit.php** as above `$_SERVER['PHP_SELF']` will return the the string `'submit.php'`.



Now in our form, instead of:

```
<form method="post" action="submit.php">
```

we can use:

```
<form method="post" action="<?php echo $_SERVER['PHP_SELF'] ?>" >
```

This is now more generalizable. So, for instance we can use this line of code in any PHP page that calls itself through a form. Or, if we rename our **submit.php** file to something else we don't need to change this line of code in the form.

Let's look at a more detailed example of submitting a form using PHP. Let's create a very simple discussion forum. Posters must enter their name and email as well as their message into an HTML form. When the form is submitted we need to check that the user has entered something in the name, email and message fields (i.e. that none of these are empty). We will also do a (very basic) check to see if the user has entered a valid email. If they haven't filled out the form correctly we send an error message back to the user asking them to fill out the form again. Otherwise we display the user's post in the forum along with the current date:

```
<html>

<head>
<title>Simple Discussion Forum</title>
</head>
<body>
<h3>Simple Discussion Forum</h3>
<?php
    if ($_SERVER["REQUEST_METHOD"] == "POST"){
        //Check if the form has been submitted
        $email = $_POST["email"];
        $post = $_POST["post"] ;
        $name = $_POST["name"];
        $errorMessage = "";

        //check that the fields are not blank
        if ( empty ($post) ) {
            $errorMessage .= " Please enter a post. ";
        }

        if ( empty ($name) ) {
            $errorMessage .= " Please enter your name. ";
        }

        if ( empty ($email) ) {
            $errorMessage .= " Please enter an email address. ";
        }else{
            if (strpos($email, "@") < 1) {
                $errorMessage .= " Please enter a valid email address
(the one you entered did not include the @ symbol) ";
            }
        }
    }
}
```

```

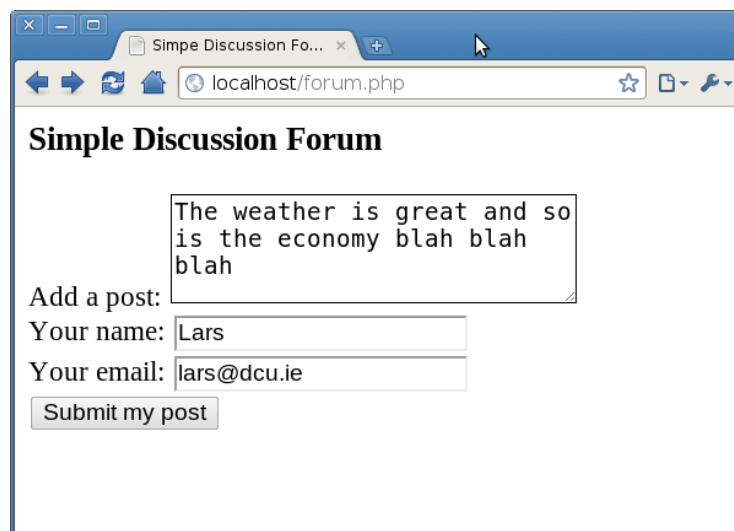
    }
}

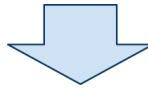
if ($errorMessage == ""){
    //there are no errors so we are ok to print the submitted
post
    $date = date("F j, Y, g:i a");
    echo "<div style='><a href='mailto:$email'>$name</a> on
$date said : <p>$post</p></div>";
}else{
    //there are errors so don't print the posting but show error
message instead
    echo "Oops, there appears to be an error.
<strong>$errorMessage</strong>";
}
}
?>

<form method="post" action="<?php echo $_SERVER['PHP_SELF'] ?>" >
    Add a post:
    <textarea name="post"></textarea><br>
    Your name:
    <input name="name" type="text"></input><br>
    Your email:
    <input name="email" type="text"></input><br>
    <input type="submit" value="Submit my post"></input>
</form>
</body>
</html>

```

If the form is filled in correctly something like the following should happen:





Simple Discussion Forum

Lars on August 8, 2010, 11:07 am said :

The weather is great and so is the economy blah blah blah

Add a post:

Your name:

Your email:

Submit my post

If the user doesn't fill in the form correctly the might see something like this:

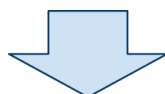
Simple Discussion Forum

Add a post:

Your name: Judy

Your email: judy.dcu.ie

Submit my post





Simple Discussion Forum

Oops, there appears to be an error. **Please enter a post.**  
**Please enter a valid email address (the one you entered did not include the @ symbol)**

Add a post:

Your name:

Your email:

~~~ End of Article ~~~



## Session 5: Using Variables and Expressions in PHP

### Variables

|                   |                                                                                           |
|-------------------|-------------------------------------------------------------------------------------------|
| Source            | <a href="http://www.dcu.ie/~costelle/?q=node/9">http://www.dcu.ie/~costelle/?q=node/9</a> |
| Date of Retrieval | 03/09/2013                                                                                |

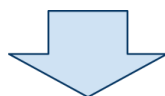
In programming a variable is used to store a value. A variable is like a drawer in a filing cabinet. Suppose we put something in the top drawer of the filing cabinet. Let's call this variable `topDrawer`. Once we have a name for our drawer we don't have to worry about what is in there constantly, we just open the drawer and have a look when we need to know. The contents (value) of the drawer can change. We can take out what is in there and replace it with something else. Again we don't have to bother remembering what the new value is, we just open `topDrawer` and have a look. That is why computers are so great, they allow us to be lazy and forgetful. If someone asks me what I have in the top drawer I say, "Don't ask me, just look in `topDrawer`". Laziness (along with impatience and hubris) is allegedly one of the psychological cornerstones of computer programming.

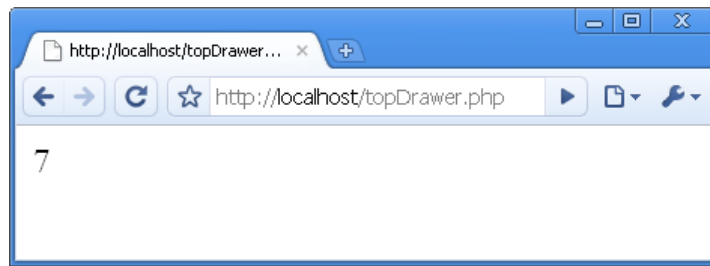
When we put something in `topDrawer` we say we *assign* it a value. We do this with the assignment operator (`=`). Now imagine the unlikely office scenario where I have a giant styrofoam number seven. I need to file this away safely so I put it in my top drawer:

```
$topDrawer = 7;
```

Here `topDrawer` has been assigned a value of `7`. You may have noticed that the variable name is preceded by a dollar sign (`$`). In the PHP language all variable names must start with a dollar sign. As soon as you see a dollar sign you know you are looking at a variable. Variable names can contain numbers, letters and underscores (as long as they start with a dollar sign followed immediately by a letter or underscore). Although the dollar sign makes variable names look a bit ugly in PHP compared to some other languages like Java, it does have advantages too as we will see. We can tell what value a variable has by simply passing the variable name as an argument to `echo`:

```
<?php
    $topDrawer = 7;
    echo $topDrawer;
?>
```





Something important to note here is that the assignment operator (`=`) is used to put a value into a variable. We use it when we want to shove something into a drawer. It *does not* mean equals (as it does in mathematics). This decision, taken in earlier days of computing, to use `=` for assignment has caused no end of confusion for novice programmers ever since. We will see the actual equals operator soon but for now just keep in mind that `=` means assignment and not equals. Confused? I don't blame you.

Lets look at another simple operator `++`, called the increment operator. This operator is nice because it doesn't do much. It only does one simple thing. The increment operator takes a number and adds one to it. Simple. Here is an example:

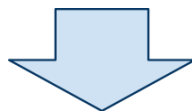
```
$foo = 2 ;  
$foo++ ; //results in 3
```

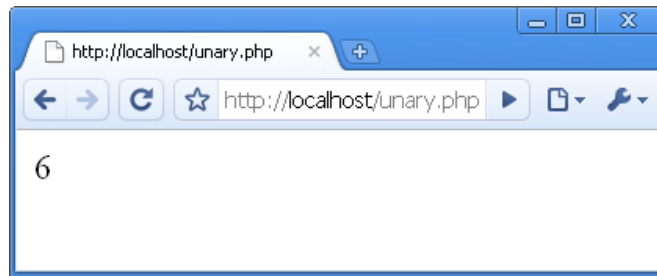
Another way of writing the above, without using the increment operator, would be:

```
$foo = 2 ;  
$foo = $foo + 1; //results in 3
```

Now lets look at another PHP code example using the increment operator and (its lovely sister) the decrement (`--`) operator which takes away a value of one:

```
<?php  
    $foo = 2 * 2;  
    $foo++;  
    $foo++;  
    $foo++;  
    $foo--;  
    echo $foo;  
?>
```





This is a pretty nonsensical program but it illustrates something important about programming. In the program we do several things to the `$foo` variable. We multiply 2 and 2 and assign the result to `$foo`. Then we increment `$foo` three times and decrement it once. Its hard to keep track of exactly what is in `$foo` at any given time. This is the dark side to putting something in a variable and forgetting about it. We may happily put a nice friendly 7 into `topDrawer` but come back later and pull out an unlucky thirteen which we didn't expect. If we ended up with an unexpected result in the last example we could go back try and debug the problem. Suppose we had hoped to end up with a 7 instead of a 6. One simple thing would be to go back and add an call to `echo` with `$foo` as an argument after each line e.g.

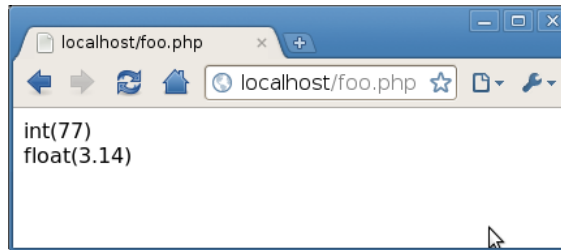
```
<?php
    $foo++;
    echo $foo;
    $foo++;
    echo $foo;
?>
```

Ok in this example this is pretty pointless, as its fairly easy to figure out what value should be in `$foo` just by looking at the code, but when a program gets complicated it often helps to stick in some calls to `echo` during development, so you can check what value is actually in a variable. It may not be the one you were expecting. If your program works but just doesn't do what you thought it would, then you have made a *conceptual error* (or logic error). This is harder to track down than a bug (syntax error) which will cause your program to crash (or at least complain loudly).

Let's add one more command to our toolkit for inspecting variables, the `var_dump` function. In programming a function is something that takes some data, does some work with it, and produces an output based on that work. This is similar to the concept of a function in mathematics. Or we may be familiar with the concept of a function in everyday language as something that has a use or does a job. The `var_dump` does a useful job of giving us information about variables. The `var` in `var_dump` stands for variable and `dump is a word in computing` that means to show information that is being stored by the computer. `var_dump` takes a variable name as an argument and tells us the value of that variable and its type. Unlike the `echo` command we need to put the arguments to a function, in parenthesis:

```
<?php
    //create variables with different types
    $foo = 77;
    $bar = 3.14;
```

```
//show information about variables  
var_dump($foo);  
echo "<br>";  
var_dump($bar);  
?>
```



We can see that `var_dump` is like `echo` in that it prints its result to the webpage. In the last example it tells us that the type of the `$foo` variable is int (integer) and its value is 77, while the type of the `$bar` variable is float (floating point) and its value is 3.14.

There are other handy PHP functions that you can use to debug code. Here are some you may want to learn more about:

- [var\\_export](#)
- [print\\_r](#)
- [error\\_log](#)

~~~ End of Article ~~~





## Session 7: Scalar Type Declarations

### Data Types

|                          |   |
|--------------------------|---|
| <b>Source</b>            | <a href="https://www.carnaghan.com/2012/06/php-language-basics/">https://www.carnaghan.com/2012/06/php-language-basics/</a> |
| <b>Date of Retrieval</b> | 10/07/2016  |

Variables in PHP are similar to define and use as with other programming languages. PHP uses loose typed variables, in other words they do not have to be defined specifically as a datatype upon initialization and can be changed to other data types using expressions.

To set a variable in PHP you only need to assign a value to it, for example `$my_variable = "hello"`; Variable names start with a dollar sign \$. The next character must be a letter or underscore and the remaining characters may be letters, names or underscores. PHP supports three different sets of data types:

- **Scalar Data Types:** integer, float, string, boolean
- **Compound Data Types:** array, object
- **Special Data Types:** resource, null

There are several functions for checking and setting the type of a variable. In some of the examples below I am using the echo command, which provides us with immediate output generated by the command. Echo can often be used to print certain things including variables, arrays, etc on screen.

To Check a variable's type:

```
echo gettype($variable);
```

Alternatively you can test to see if a variable is of a certain type (returned as either true or false) by using the `is_int`, `is_float`, `is_bool`, `is_array`, `is_object`, `isResource`, `is_null` functions.

To set a variable's type:

```
settype($variable, "type")
```

Alternatively casting can be used when you only want to temporarily use the variable as a different type:

```
echo (type) $variable;
```

~~~ End of Article ~~~



# PHP Data Types

|                   |                                                                                                         |
|-------------------|---------------------------------------------------------------------------------------------------------|
| Source            | <a href="https://en.wikipedia.org/wiki/PHP#Data_types">https://en.wikipedia.org/wiki/PHP#Data_types</a> |
| Date of Retrieval | 10/07/2016                                                                                              |

PHP stores integers in a platform-dependent range, either a 64-bit or 32-bit signed integer equivalent to the C-language long type. Unsigned integers are converted to signed values in certain situations; this behavior is different from that of other programming languages. `Integer` variables can be assigned using decimal (positive and negative), octal, hexadecimal, and binary notations.

Floating point numbers are also stored in a platform-specific range. They can be specified using floating point notation, or two forms of scientific notation. PHP has a native `Boolean` type that is similar to the native Boolean types in Java and C++. Using the Boolean type conversion rules, non-zero values are interpreted as true and zero as false, as in Perl and C++.

The `null` data type represents a variable that has no value; `NULL` is the only allowed value for this data type.

Variables of the "`resource`" type represent references to resources from external sources. These are typically created by functions from a particular extension, and can only be processed by functions from the same extension; examples include file, image, and database resources.

Arrays can contain elements of any type that PHP can handle, including resources, objects, and even other arrays. Order is preserved in lists of values and in hashes with both keys and values, and the two can be intermingled. PHP also supports `strings`, which can be used with single quotes, double quotes, nowdoc or heredoc syntax.

The Standard PHP Library (SPL) attempts to solve standard problems and implements efficient data access interfaces and classes.

~~~ End of Article ~~~



## What's New in PHP 7 New Features: Anonymous Classes

|                          |   |
|--------------------------|---|
| <b>Source</b>            | <a href="http://www.codedoodle.com/2016/01/whats-new-in-php-7-new-features.html">http://www.codedoodle.com/2016/01/whats-new-in-php-7-new-features.html</a> |
| <b>Date of Retrieval</b> | 10/07/2016  |

Anonymous classes are used for simple use and throw away purpose. An example would be implementing an simple Logging or Error implementations.

Anonymous class like other classes can accept constructor arguments, extends other classes and interfaces and also include traits.

### Syntax

```
(new class[, arguments] [extends] [implements] {  
    // include traits  
  
    // methods  
});
```

### Example: php7-anonymous-class.php

```
// Simple error interface  
interface ErrorInterface {  
    public abstract function showError($msg);  
}  
  
// Class containing users based operations  
class User {  
    protected $error;  
    public function setErrorInterface(ErrorInterface $error) {  
        $this->error = $error;  
    }  
  
    public function createUser($user) {
```

```
        if ($user) {

            echo 'User Cretaed ' . PHP_EOL;

        } else {

            $this->error->showError('Cannot create user..!!!');

        }

    }

}

$userObj = new User();

$userObj->setErrorInterface(

    // Create and pass the anonymous class.

    new class() implements ErrorInterface {

        public function showError($msg) {

            // Send some headers and set other info's

            echo 'Error: ' . $msg . PHP_EOL;

        }

    }

);

// Error will be showed

$userObj->createUser(null);

// This will display user created.

$userObj->createUser('Foo');
```

## Output

Error: Cannot create user..!!!

User Cretaed

In the above example, the `setErrorInterface` method is passed an anonymous class that implements the `ErrorInterface` interface and the `User` class uses that anonymous class as an instance and shows error using that object instance of the anonymous class.

~~~ End of Article ~~~

## Session 8: PHP Operators

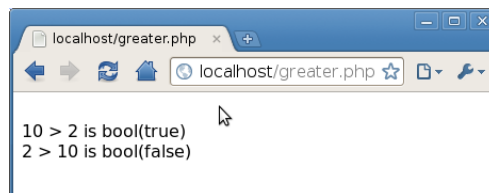
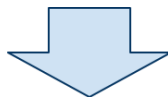
### Comparing

|                   |                                                                                             |
|-------------------|---------------------------------------------------------------------------------------------|
| Source            | <a href="http://www.dcu.ie/~costelle/?q=node/29">http://www.dcu.ie/~costelle/?q=node/29</a> |
| Date of Retrieval | 03/09/2013                                                                                  |

The increment (**++**) and decrement (**--**) are *unary operators*, that is they only operate on one piece of data or operand. The thing that an operator operates on is called an *operand*. We have already looked at the arithmetic operators, such as addition (**+**) which take two operands e.g. **1 + 1**. Because they operate on two operands they are known as *binary operators*. Now let's look at some more binary operators known as the comparison operators.

You are probably familiar with the greater than **>** and less than **<** operators from Mathematics. These are two of the [comparison operators](#) in PHP. These operators compare two operands and return a value:

```
<?php
//greater than
echo "<br>10 > 2 is ";
var_dump(10 > 2);
echo "<br>2 > 10 is ";
var_dump(2 > 10);
?>
```



The **>** operator returns a boolean (bool) value. The boolean type is the simplest of the types. It can have only one of two values: true or false. In the above example **10 > 2** evaluates to **true** (because 10 is greater than 2). Similarly **2 > 10** evaluates to **false** (because 2 is *not* greater than 10). If you want to test whether a value is greater than *or* equal to another value there is a comparison operator for just that:

```
2 >= 2; //true, because 2 is greater than or equal to 2
2 > 2; //false
```

The less than **<** and less than or equal to **<=** operators work similarly:

```
2 <= 2; //true, because 2 is less than or equal to 2
2 < 2; //false
```

One of the most important comparison operators is equals. We have already seen that the assignment operator uses the equals sign `=`. So, for equals in many programming languages, including PHP, we use `==`:

```
2 == 2;           //true because 2 equals 2
2 == 5;           //false
2 == 1 + 1;       //true
```

This last line is true because the addition operator `+` has greater precedence than the equals operator `==`. Addition does its work first adding 1 and 1 and then passing the result to the equality operation. So far we have looked at using comparing integers. The comparison operators can also take types other than numbers as arguments. Here is an example of using strings with the comparison operator:

```
//Assignment
$foo = "apples";
//Equality
$foo == "apples"; //true
$foo == "oranges"; //false
```

We can see from this example that the assignment and equality operations look similar but do very different things. We also have the *not equals* operator `!=` which returns true if its two operands are not equal:

```
//Not equal operator
"oranges" != "apples"; //true, "oranges" is not equal to "apples"
2 != 2;                //false, 2 is equal to 2
```

Because PHP is not a fussy language it allows us to do some things that don't make immediate sense. For instance it allows us to compare a string and a number:

```
"oranges" == 22; //false, "oranges" is not equal to 22 (whatever that means!)
```

Comparing different types can become tricky however:

```
"a" == 0;           //true
"0" == 0;           //true
"a" == "0";         //false
```

The string `"a"` being equal to `0` is not something we might expect. It is also odd how strings `"a"` and `"0"` can both equal the number `0` but are not equal to each other. What PHP tries to do when [comparing a string and a number](#) is convert the string to a number first. We also have a stricter equals operator that compares types as well as values, and will only return true if both types are equal. This equality operator is called identical `===`:

```
"0" === 0;          //false, different types
"0" === "0";        //true
12 === 12.0;         //false, different types
```

Use the type equality operator `===` instead of the less strict equality operator `==` when you can. It will be easier to tell what the outcome of your code will be and it should lead to less bugs.

*~~~ End of Article ~~~*



## Session 10: Conditional Statements in PHP

### Selecting

|                   |                                                                                             |
|-------------------|---------------------------------------------------------------------------------------------|
| Source            | <a href="http://www.dcu.ie/~costelle/?q=node/31">http://www.dcu.ie/~costelle/?q=node/31</a> |
| Date of Retrieval | 03/09/2013                                                                                  |

You can think of most programming as made up of three constructs called *sequence*, *selection* and *iteration*. The sequence construct is the most basic whereby commands are simply executed in sequence:

```
//Sequence
```

```
echo "step one"; //this happens
```

```
echo "step two"; //then this happens
```

Life is rarely a sequence of simple tasks. Most of the time we are planning for various things that *could* happen. In programming we use *selection* constructs to handle this. The simplest selection construct is where we have two possibilities. To do this in PHP we use the **if** statement which says: **if** something happens we will perform a task, otherwise we will do something **else**. Imagine we want to test a variable called **\$foo** to see if it contains a value greater than 1:

```
if ($foo > 1) {  
    echo "foo is greater than 1";  
}else{  
    echo "foo is NOT greater than 1";  
}
```

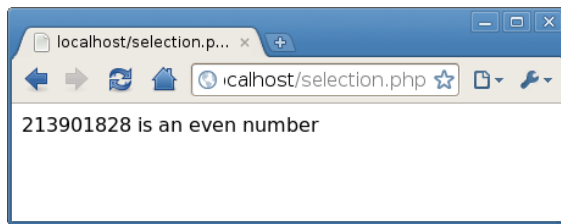
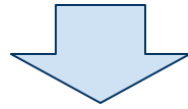
The if statement starts with the word **if**, followed by two brackets which contains a test or condition. This test in the brackets evaluates to true or false. If the test condition is true the next block of code in curly braces is executed. If the test condition evaluates to false the block of code after the else is executed instead. Here is an example that tests a variable to see if it contains an even or an odd number:

```
<?php
```

```
$num = rand();  
if ($num % 2 == 0){  
    echo "$num is an even number";  
}else{  
    echo "$num is an odd number";  
}
```

```
?>
```





There are a couple of things going on here. Firstly we are calling the **rand** function and assigning the result to the `$num` variable. The [rand function](#) creates a random integer. (Unlike the **var\_dump** function we don't have to pass an argument to **rand**, so the brackets following the function name are empty.)

In the test condition of the **if** statement we are using the modulus operator **%** with **\$num** and **2**:

```
if ($num % 2 == 0)
```

This divides **\$num** by **2** and returns the remainder. We know when we divide a number by 2 that the result will be either 1 or 0. If the result is 0 we know that the number is even and otherwise it is odd. The modulus operator is useful for testing to see whether a number is odd or even. The result of the modulus operation is then passed to the equality operator **==** which tests to see if it is equal to 0. The arithmetic operators have higher precedence than the comparison operators so we know the modulus operation will be performed before the equality operation. We could make this clearer by putting brackets around the modulus operation:

```
if (($num % 2) == 0)
```

This does the exact same thing as before but this time it is clearer to someone reading the code that the modulus operation will happen first and then the equality operation because things in brackets are always evaluated first.

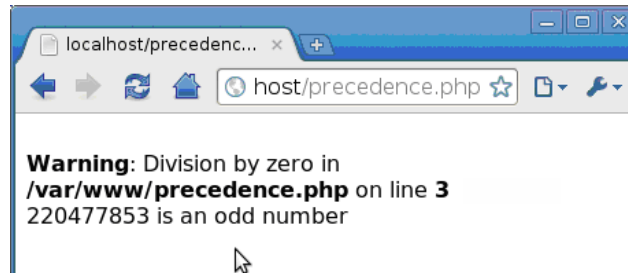
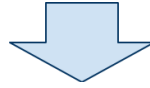
What would happen if the precedence were reversed and the equality operation happened before the modulus? We can simulate this by putting brackets around the equality operation. This will force it to execute first and pass its result to the modulus operation. For curiosity's sake let's see what happens if we do this:

```
<?php
```

```
    $num = rand();
```

```
    if ($num % (2 == 0)){ //just doing this for curiosity's sake
        echo "$num is an even number";
    }else{
        echo "$num is an odd number";
    }
}
```

?>



PHP gives us a warning that we are doing something bad. What has happened? Well the first part of the test condition to be executed is the equality operation because it is in brackets:

`(2 == 0)`

This returns false because 2 is not equal to 0. Then false is passed to the modulus operation. The modulus operator works with numbers so it expects integers as arguments. When it gets false as an argument it tries to convert it to a number and makes it 0. Now it tries to perform its operation by dividing 0 by 2 but this makes no sense so it gives us a warning.

We see here that PHP tries to convert types to ones that it expects. We have seen that false is converted to 0 when a boolean is converted to an integer. This also works in reverse and 0 gets converted to false when a boolean is expected. We can use this in our **if** statement test condition:

```
if ($num % 2)
```

The modulus operation returns an integer of either 1 (which becomes true) or 0 (which becomes false). So we can simplify our program:

```
<?php
$num = rand();
if ($num % 2){
    echo "$num is an even number";
}else{
    echo "$num is an odd number";
}

?>
```

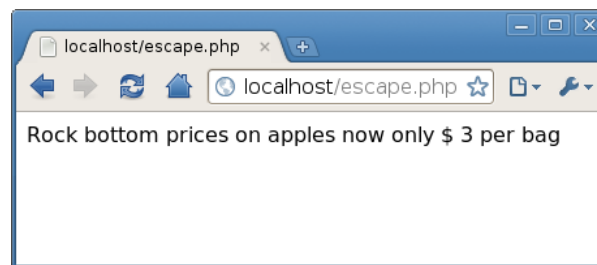
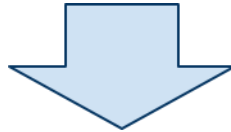
You may have noticed that the value of the \$num variable is being printed to the webpage. This is one of the advantages of PHP's variable naming rules. Remember that we must name variables starting with a dollar sign. Whenever PHP sees a dollar sign in a string it prints the value of the variable. This comes in very handy but suppose we actually want to use a dollar sign in a string. In this case we use the escape sequence `\$`:

```
<?php
$price = rand(1, 10);
```

```

if ($price < 5){
    echo "Rock bottom prices on apples now only \$ $price per
bag ";
}else{
    echo "Premium organic apples now only \$ $price per bag";
}
?>

```



In this example we are passing two variables to `rand` to tell it to create a random number between 1 and 10. We assign this value to a variable called `$price`. **if** `$price` is less than 5 we show some information on how cheap the apples are, or **else** we show some information on the quality of the apples. Not only are you learning programming here but also the complex secrets of marketing.

In this example we passed two arguments to **rand**. We have previously seen that **rand** can also be called with no arguments by leaving the parenthesis empty. What's going on here? Well what `rand` is really doing is supplying its own defaults if we give it no arguments. In the case that we don't pass it any values it takes the biggest possible range it can to pick a random number from, 0 to [a big number](#).

The **else** part can be left out of the **if** statement altogether:

```

if ($testCondition){
    //do something here
}

```

If there is no **else**, and the test condition in brackets is false, the **if** statement is simply ignored. It is also possible to write the **if** statement without using curly braces at all:

```

if ($friendly) echo "Hello!" ;

```

is the same as:

```

if ($friendly){
    echo "Hello!";
}

```

which is also the same as:

```

if ($friendly)

```

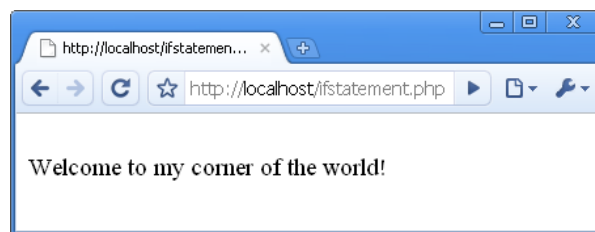
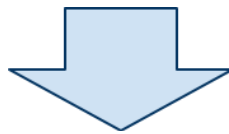
```
echo "Hello!" ;
```

It is important to note that indentation and white-space make no difference to PHP. It looks for curly braces or semi-colons in order to determine the end of an instruction. Consider the following:

```
<?php

$friendly = false;
if ($friendly)
    echo "Hello!" ;
    echo "<br>Welcome to my corner of the world!" ;

?>
```



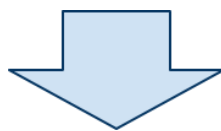
The indentation in the code of the above example may mislead someone into thinking that the two echo statements are controlled by if statement but actually only the first one is. We need to use curly braces when we want more than one instruction to execute as part of an **if** statement:

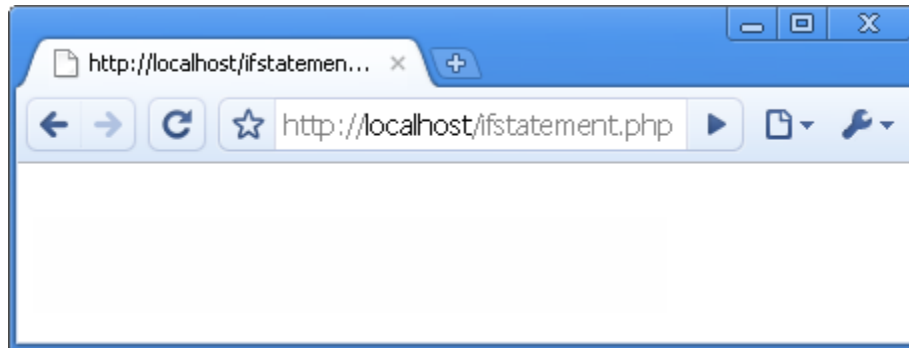
```
<?php

$friendly = false;

if ($friendly){
    echo "Hello!" ;
    echo "<br>Welcome to my corner of the world!" ;
}

?>
```





Sometimes we can achieve the same result in programming by using different code. For instance, we can write simple **if** statements with *or without* curly braces. When we want to execute more than one instruction as part of an **if** statement we must use the braces of course. Because of this it is a good idea to *always* use the curly braces. This makes our code more understandable and means that our code will be more consistent: whenever we use an **if** we need to use curly braces to enclose the condition after the test condition. This is what we call a *convention* in programming. It is different to a *rule* that is enforced by the processor. Our program may still run if we don't use our convention but conventions are vitally important in programming. A common set of conventions may be used in a language, or within a company, an organisation or any community of programmers.

~~~ End of Article ~~~



## Session 12: Flow Control in PHP

### Loops and Iteration

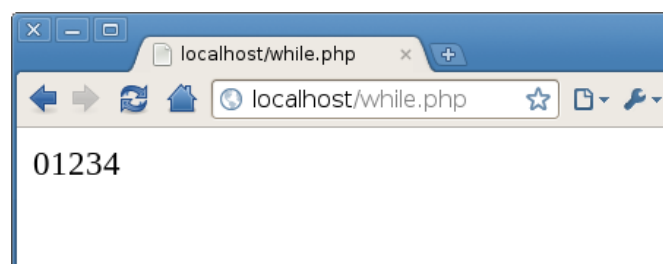
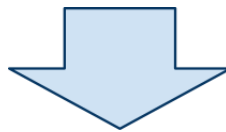
|                   |   |
|-------------------|---|
| Source            | <a href="http://www.dcu.ie/~costelle/?q=node/33">http://www.dcu.ie/~costelle/?q=node/33</a> |
| Date of Retrieval | 03/09/2013  |

One of the best things about programming is that it lets us do repetitive tasks. This is good because repetitive are boring and take a lot of time. And, because repetitive tasks are boring we could easily make a mistake when doing them. In computing we use loops to do this and we call it *iteration*. Each time we repeat a task during a loop we call this an *iteration*. We can do iteration with the **while** construct:

```
while ( $someExpression ){  
  
    //keep doing what is inside the curly braces until $someExpression is  
    false.  
  
}
```

**while** is a pretty simple construct. It checks to see if the expression in the brackets is true. If is true it executes any code inside the curly braces. Then it checks the expression inside the brackets again to see if it true and if so executes the code in the curly braces and so on. Here is an example that prints a list of numbers:

```
<?php  
  
    $i = 0;  
  
    while ($i < 5 ){  
  
        echo $i;  
  
        $i++;  
  
    }  
  
?>
```



This example shows a very simple example of iteration using **while**. The condition in brackets (**\$i < 5**) is tested and if it is true the code in the curly braces is executed. Each

time the code in the curly braces is executed we use the increment operator `++` to increment the `$i` variable. Eventually this variable becomes 5 and in then the loop stops executing (so 5 is never printed).

In this example we *know* we want to do something exactly five times. Let's look at a better example of using `while` where we *don't know* how many times we want to do something. Suppose we want to get even number at random. One way of doing this is to keep generating numbers with the `rand` function until we get an even one:

```
//get an even number
$num;
while ( $num % 2 ){

    $num = rand();
}
echo $num;
```

This programme keeps generating a random number until we get an even one. You can see that this `while` looks very like an `if` construct. Of course we could write this programme another way entirely:

```
$num = rand();
if ( $num % 2 ){
    $num--;
}

echo $num;
```

In this example we generate a random number, then if it is not an even number we subtract 1 from it. Now we have an even number.

In programming there are often many ways to write the same programme. So how do we know which is the *correct* one? The first example here is probably not as good as the second because it is less efficient - we could be stuck in the loop for a long time waiting for an odd number. Because there are many ways of doing the same thing in programming it's often not easy to know which is the best one. Sometimes the best way is the way that is most readable (though not necessarily with the least amount of code). Other times the best solution may be the one that executes fastest and uses least memory.

We could rewrite our first simple iteration example using a `for` loop instead of a `while`. `for` loops are useful when we know the exact number of iterations we need to make:

```
for ($i = 0; $i < 5; $i++){

    echo $i;
}
```

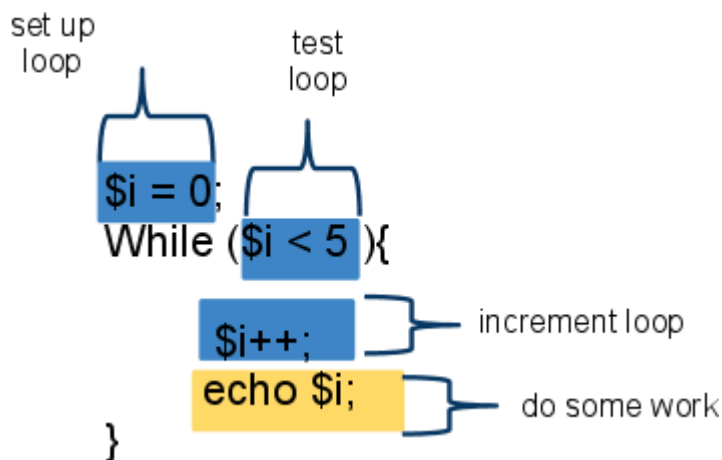
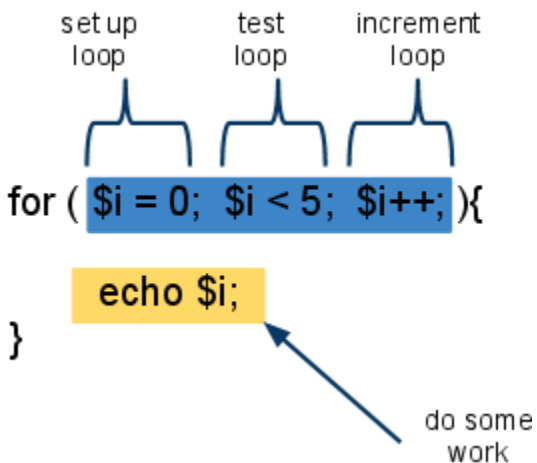
You can see that this is very similar to:

```
$i = 0;
while ($i < 5 ){

    echo $i;
```

```
$i++;  
}
```

What is better about the **for** over the **while** in this particular example here is that code to *describe the looping* is separate from code to *do the work* in the loop. Everything to *set up*, *run* and *stop* the loop happens inside the brackets of the **for** loop:



In both loops a variable `$i` is used as a counter. There are three things that control the loop: The first is initialising the counter:

```
$i = 0;
```

The second is testing the counter for some condition (in this case that the counter is less than 5):

```
$i < 5;
```

And then incrementing the counter (if the test condition is true):

```
$i++;
```



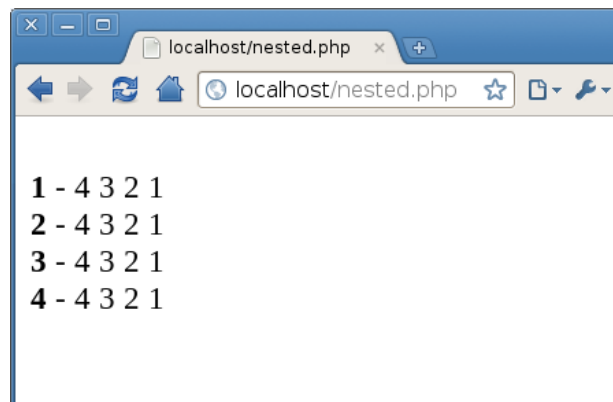
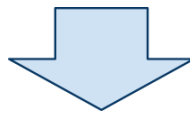
In the **for** loop these three things go in brackets after the word **for**, and are separated by semicolons. By convention the counter variable is normally named **i**.

We can add a loop inside another loop (called a nested loop) and in this case we use letters **j**, **k**, **l** and so on as counters:

```
<?php

//a nested loop
for ($i = 1; $i <= 4; $i++){ //outer loop uses i as counter name
    echo " <br><b> $i </b> - ";
    for ($j = 4; $j >= 1; $j--){ //inner loop uses j for its
counter
        echo " $j ";
    }
}

?>
```



In this last example the outer loop is going from 1 to 4 while the inner loop is going the opposite direction from 4 to 1. For each completion of the outer loop the inner loop completes 4 times.

Suppose we wanted to create a multiplication table ("two times two is four, three times two is six " and so on). We can do this using nested loops. Here is an example that uses an html table to format the output (loops are very useful for creating html tables):

```
<html>
<head>
    <title>Multiplication Table</title>
```

```
<style>
    table {padding: 10px;}
    td {text-align: center; padding: 5px}
    h1 {; text-align: center;}
</style>
</head>
<body>

<h1> Multiplication Table</h1>
<table border="1">

<?php
    // $i are the rows
    for ($i = 1; $i < 11; $i++){
        if ($i > 1 ) {
            echo " <tr><td>  $i  </td>";
        }
        else{
            echo " <tr><td>  &nbsp;  </td>";
        }
        // $j are the columns
        for ($j = 2; $j < 11; $j++){
            echo " <td> ", $j * $i , "</td> " ;
        }
        echo " </tr>";
    }
?>

</table>
</body>
</html>
```



|    |    |    |    |    |    |    |    |    |     |
|----|----|----|----|----|----|----|----|----|-----|
| 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10  |
| 2  | 4  | 6  | 8  | 10 | 12 | 14 | 16 | 18 | 20  |
| 3  | 6  | 9  | 12 | 15 | 18 | 21 | 24 | 27 | 30  |
| 4  | 8  | 12 | 16 | 20 | 24 | 28 | 32 | 36 | 40  |
| 5  | 10 | 15 | 20 | 25 | 30 | 35 | 40 | 45 | 50  |
| 6  | 12 | 18 | 24 | 30 | 36 | 42 | 48 | 54 | 60  |
| 7  | 14 | 21 | 28 | 35 | 42 | 49 | 56 | 63 | 70  |
| 8  | 16 | 24 | 32 | 40 | 48 | 56 | 64 | 72 | 80  |
| 9  | 18 | 27 | 36 | 45 | 54 | 63 | 72 | 81 | 90  |
| 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |

In this example we have included more html and css. The php that generates the rows and columns of our table is separate from the rest of the html page. We can include the `<?php ?>` tags anywhere in our php document, they don't need to be at the top and bottom. This allows us to include fragments of php embedded in our html page. It's always difficult when dealing with several languages in one space (e.g. html, css, javascript and php can all be mixed in the one file) and there aim should always be to keep related parts of your program together. In this case we want to keep the php part of the code separated from the rest of the static html as much as possible.

Something we might do to improve the output is remove the 1 from the top left cell of the table. A simple way to do this is put a line in our code saying only to print the `$i` variable if it is greater than 1. This makes our code a bit more complicated but makes our multiplication table neater. Something we can do to make the code better is make take the numbers that specify how many iterations each loop should run for out of the loop code altogether. Instead we create two variables one called columns and one called rows. Each of these is assigned the number of rows and columns we want in our table. It's easy now for someone who doesn't know what our code is doing to change the number of the columns or rows just by changing the value of these variables. It also makes our code easier to read and understand what it is doing. One other thing that we have changed in our modified program is that we are using "greater than or equal to" in our loop test condition:

```
$i <= $rows
```

This basically means we keep going with the loop until we reach 10 and 10 is included:

```
<html>

<head>

<title>Multiplication Table</title>

<style>

    table {padding: 10px;}

    td {text-align: center; padding: 5px}
```

```
        h1 {; text-align: center;}

    </style>
</head>
<body>
<h1> Multiplication Table</h1>
<table border="1">
<?php
    $columns = 10;
    $rows = 10;
    for ($i = 1; $i <= $columns; $i++){
        if ($i > 1 ) {
            echo " <tr><td>  $i  </td>";
        }
        else{
            echo " <tr><td>  &nbsp;  </td>";
        }
        for ($j = 2; $j <= $rows; $j++){
            echo " <td> ", $j * $i , "</td> " ;
        }
        echo " </tr>";
    }
?>
</table>
</body>
</html>
```

~~~ End of Article ~~~



## Session 14: Functions in PHP

### Functions

|                   |                                                                                                                               |
|-------------------|-------------------------------------------------------------------------------------------------------------------------------|
| Source            | <a href="https://p2pu.org/en/groups/learn-php/content/functions/">https://p2pu.org/en/groups/learn-php/content/functions/</a> |
| Date of Retrieval | 03/09/2013                                                                                                                    |

A *function* is just a section of code that can be executed whenever and where you want it to be. The great thing about *functions*, is that you can use them over and over! It's great because once you create a *function* that you use more than once, it saves you time on more coding.

You can create your own *function*, or use one of the hundreds built in *functions* that PHP offers. You should also know that *functions* can be confusing, and understanding them at first can be difficult for some people. If you're one of those people who ends up having a hard time understanding functions, just keep trying!

This class is long and has a lot of code in it - don't be intimidated by it, I try my best to keep it simple!

#### Creating your own function:

It's pretty simple to create a *function*. Here's the syntax:

```
<?

function functionName()
{
    // code you want to use
}

?>
```

There are a few basic rules we should go over first. Your PHP functions should be named after what it does. It's much easier to keep track of which function does what if you use this method. And like a variable, it must start with a letter or underscore.

Let's go over the *function* syntax real quick:

To define a function, you must start your block of code with the word "function" and this will tell PHP, "This is a function I'm creating". Then you name your function, followed by ().

Everything from your function will go inside the curly brackets. You can even call other functions inside a function!

**Note:** PHP's recursion is not very good, so if you try to make a function loop through itself ten thousand times.. There's a chance it won't.

### Output:

This is something more tutorials and people won't tell you about PHP functions. There are 2 kinds of output:

Output that is created inside the function, and

Output that has to be called from outside the function.

Let's create our first function that outputs a string from inside the function.

```
<?
```

```
function favCar()  
{  
    echo "Audi";  
}
```

```
?>
```

Ok, simple enough. Now let's see how we call it:

```
<?
```

```
echo "I really like ";  
favCar();
```

```
?>
```

### Displays:

I really like Audi

But what if we added favCar() into what we echo, like so:

```
<?="I really like " . favCar(); // Remember the shorthand way to echo? ?>
```

### Displays:

AudiI really Like

That's a little backwards.. The reason why our script put "Audi" in front, was because it is trying to be echo'd inside an echo. PHP saw that code like this:

```
<? echo "I really like " . echo "Audi"; ?>
```

To remedy this, we use the *return* command instead of echo inside of our function.

```
<?

function favCar()
{
    return "Audi";
}

?>
```

What the *return* command does is shoot out the answer, and then you can use it inline with a *echo* command. Think of *return* like the answer to a solution;  $1 + 1 = \text{return}$ . If this doesn't make any sense to you now, it probably will a little ways down the PHP Learning Road.

### Parameters:

A parameter is an extra piece of information that you can feed to your function. The syntax is very similar, only now we add variables inside our regular brackets.

```
<?

function favCar($name)
{
    echo "My favorite car is " . $name;
}

?>
```

**How to call it:**

```
<?

    favCar("Audi"); // Displays "My favorite car is Audi"

?>
```

`$name` in the function `favCar` is a variable, but when we call it, we used a string, which works as well. If we wanted to, we could use a variable rather than a string.

```
<?

$car = "Audi";

function favCar($name)
```

```
{
    echo "My favorite car is " . $name;
}

favCar($car); // Displays "My favorite car is Audi"; same result

?>
```

Functions can hold many parameters, but we're only going to use two in our next example.

```
<?

function add($a, $b)
{
    $answer = $a + $b;
    return $answer;

    /* Our we can use
    return $a + $b;
    as a shorter method, your choice! */
}

echo "10 + 5 = " . add(10, 5);

?>
```

All we did was add a comma after our first parameter, and added another parameter (that looks like a variable).

Our function called *add* will simply add the first number and the second number together. The *return* shoots out the *\$answer* for *\$a + \$b*, and we display it on our page using the *echo* command.

### Boolean:

Boolean logic is a form of algebra that returns TRUE or FALSE. (1 or 0). Don't worry, we don't do algebra in this course!

If something returns 1, it's TRUE. Or if it returns TRUE, it's 1. Same concept with FALSE. If something returns 0, it's FALSE. Or if it returns FALSE, it's 0.

Let's look at how we fit this in with functions:

```
<?
```



```
function overTen($a, $b)
{
    $total = $a + $b;
    if($total>=10)
    {
        return true;
    }
    else
    {
        return false;
    }
}

if(overTen(5, 5)==true)
{
    echo "Our addition function said our result is equal to or greater than
10";
}
else
{
    echo "Our function's result was below 10";
}

?>
```

~~~ End of Article ~~~



## Path and Directory Function in PHP

|                          |   |
|--------------------------|---|
| <b>Source</b>            | <a href="http://terriswallow.com/weblog/2007/path-and-directory-function-in-php/">http://terriswallow.com/weblog/2007/path-and-directory-function-in-php/</a> |
| <b>Date of Retrieval</b> | 03/09/2013  |

### Directory Name – `dirname()`

This is an easy one, it returns the name of the directory the specified file is in. Here's two examples for you

```
<?php
$path = "/html/index";
$dir = dirname($path); // $dir is set to "/html"

$path = "/home/httpd/domains/example.org/html/index.php";
$dir = dirname($path); // $dir is set to
"/home/httpd/domains/example.org/html"
?>
```

Though be warned if there is not file then the last directory is not listed and is instead recognized as the file

```
<?php
$path = "/html/";
$dir = dirname($path); // $dir is set to "/"

$path = "/home/httpd/domains/";
$dir = dirname($path); // $dir is set to "/home/httpd"
?>
```

### File Name – `basename()`

Basename pulls out what `dirname` misses, the actual file being called

```
<?php
$path = "/home/httpd/domains/example.org/html/index.php";
$file = basename($path); // $file is set to "index.php"

$path = "/html/index";
$file = basename($path); // $file is set to "index"
?>
```

This function also has an optional parameter to remove a specified suffix, like so:

```
<?php
$path = "/home/httpd/domains/example.org/html/index.php";
$file = basename($path, '.php');          // $file is set to "index"
?>
```

### Path Information – pathinfo()

Path Info will give you any information you need about a specified path. It returns an array of information or if you pass it the option of what specific information you want in the second (optional) parameter it returns a string.

```
<?php
$path_parts = pathinfo('/home/httpd/domains/example.org/html/index.php');

$dir = $path_parts['dirname']; // $dir is set to
"/home/httpd/domains/example.org/html/" -- same as dirname($path_parts)
$basename = $path_parts['basename']; // $dir is set to "index.php" -- same as
basename($path_parts)
$ext = $path_parts['extension']; // $ext is set to "php"
$file = $path_parts['filename']; // $file is set to "index"
?>
```

My **favorite thing** about pathinfo() is the 'filename' key of the array. If I pass it a path that has a period in the filename it detects it correctly.

```
<?php
$path_parts = pathinfo('/home/httpd/domains/example.org/html/index.344.php');

$basename = $path_parts['basename']; // $dir is set to "index.344.php" -- same
as basename($path_parts)
$ext = $path_parts['extension']; // $ext is set to "php"
$file = $path_parts['filename']; // $file is set to "index.344"
?>
```

### Real Path – realpath()

Realpath is a great function that gives you the full absolute path to the directory of your choice in relation to the directory the file is in.

If my file is located in the absolute path of ``/home/httpd/domains/example.org/html/`` then the following would happen using the realpath function.

```
<?php
$new_path = realpath('./');      //$new_path is set to
"/home/httpd/domains/example.org/html/"

$new_path = realpath('../../subdomain.example.net/html/includes');
//$new_path is set to
"/home/httpd/domains/subdomain.example.net/html/includes"
?>
```

~~~ End of Article ~~~



## Session 16: Working with Arrays

### How to Create Arrays in PHP?

|                   |                                                                                                                                                     |
|-------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------|
| Source            | <a href="http://www.latestcode.net/2009/11/how-to-create-arrays-in-php.html">http://www.latestcode.net/2009/11/how-to-create-arrays-in-php.html</a> |
| Date of Retrieval | 03/09/2013                                                                                                                                          |

#### What is an array in php?

An Array is a collection of data values. It can also define as a collection of key/value pairs. An array can store multiple values in as a single variable. because array store values under a single variable name. Each array element has index which is used to access the array element. An array can also store different type of data values. That means you can store integer and string values in the same array. PHP has lot of [array functions](#) to deal with arrays. You can store integers, strings, floats or objects in arrays. Arrays are very important in any programming language. You can get complete idea of arrays by reading this tutorial.

#### How to define an array in PHP?

```
$numbers = array(1,2,3,4,5,6,7,8,9,0);
```

This will create an array of 10 elements. array is the PHP keyword that use to define a array. Not like other programming languages array size is not a fixed value. You can expand it at any time. That means you can store any value in the array you create. PHP will ready to adjust the size of the array.

#### Why arrays are important?

- Store similar data in a structured manner.
- Easy to access and change values.
- You can define any number of values (No need to define multiple variables).
- Save time and resources.
- To store temporary data.

In PHP there are two types of arrays.

1. Numeric/Indexed Arrays
2. Associative Arrays

#### Numeric Arrays

An indexed or a numeric array stores its all elements using numerical indexes. Normally indexed array begins with 0 index. When adding elements to the array index will be automatically increased by one. So always maximum index may be less than one to the size of the array.

Following example shows you how to declare numeric arrays in PHP

e.g. `$programming_languages = array('PHP','Perl','C#');`

Also you can define indexed/numeric arrays as follows.

```
$programming_languages[0]='PHP';  
$programming_languages[1]='Perl';  
$programming_languages[2]='C#';
```

Actually you don't need to specify the index, it will be automatically created. So you can also define it as bellow.

```
$programming_languages[]='PHP';  
$programming_languages[]='Perl';  
$programming_languages[]='C#';
```

Above array has three elements. The value 'PHP' has the index of 0 while 'C#' has index of 2. You can add new elements to this array. Then index will be increased one by one. That means the next element index will be 3. You can use that integer index to access array index as follows

```
echo $programming_languages[2]; //This will be print 'C#'  
or  
echo $programming_languages{2}; //You can also use curly brackets
```

If you access an element which is not exists in the numeric array you will get an error message like *Notice: Undefined offset: 4 in D:\Projects\array.php on line 7.*

## Associative Arrays

Associative arrays have keys and values. It is similar to two-column table. In associative arrays, key is use to access the its value. Every key in an associative array unique to that array. That means you can't have more than one value with the same key. Associative array is human readable than the indexed arrays. Because you can define a key for the specific element. You can use single or double quotes to define the array key. Following example explains you how to declare associative arrays in PHP

Method 1:

```
$user_info = array('name'=>'Nimal','age'=>20,'gender'=>'Male','status'=>'single');
```

Method

2:

```
$user_info['name']='Nimal';  
$user_info['age']='20';  
$user_info['gender']='Male';  
$user_info['status']='Single';
```

Above array has four key pair values. That means array has 4 elements. Each element has a specific key. We use that key to access array element in associative arrays. You can't duplicate the array key. If there a duplicate key, the last element will be used.

```
echo $user_info['name'];//This will print Nimal

or
echo $user_info{'name'};
```

If you access an element which is not exists in the associative array you will get an error message like *Notice: Undefined index: country in D:\Projects\array.php on line 11.*

## Multidimensional Arrays

Multidimensional arrays are arrays of arrays. That means one element of the array contains another array. Like wise you can define very complex arrays in PHP Suppose you have to store marks of the students according to their subjects in a class room using an array, You can use a multidimensional array to perform the task. See bellow example.

### Syntax:

```
$marks=array(
    'nimal'=>array('English'=>67, 'Science'=>80, 'Maths'=>70),
    'sanka'=>array('English'=>74, 'Science'=>56, 'Maths'=>60),
    'dinesh'=>array('English'=>46, 'Science'=>65, 'Maths'=>52)
);

echo $marks['sanka']['English'];//This will print Sanka's marks for English
```

You can use **array\_keys()** method to get all keys of an array. This will returns an array of keys.

```
print_r(array_keys($marks));
```

```
//This will output Array ( [0] => nimal [1] => sanko [2] => dinesh )
```

Note: you can use student number and subject number instead of using student name and subject name.

### 1. How to loop/iterate through an array in PHP?

if there are large number of elements in an array, It is not practical to access elements by manually. So you have to use loops for it. You can use foreach and for loops for access array elements. I think foreach loop is the most convenient method to access array elements. because you don't need to know the size of the array. In this tutorial you can learn iterate through arrays.

```
<?PHP
//Using foreach loop
```

```
$days = array('Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday');
```

```
foreach($days as $day) {
```

```

echo $day.'<br/>';

}

//Using for loop

$programmingLanguages = array('PHP','Python','Perl','Java','C#','VB.Net','C++');

for($x=0;$x<count($programmingLanguages);$x++){
echo $programmingLanguages[$x]. '<br/>';

}

//Also you can print all array elements using print_r() method. This is
useful to determine array structure.

print_r($countries);

//This will out put Array ( [0] => SriLanka [1] => India [2] => America [3]
=> UK [4] => Japan [5] => China )

? >

```

## 2. How to get the size of an array using pHP?

Sometimes you may need to get how many elements exists in the array or the size of the array. You can use pHP in-built methods to find the size of a any array. See bellow examples. Also you can count all elements in a multidimensional array using count() function.

```

<?pHP
$countries = array('SriLanka','India','America','UK','Japan','China');

echo count($countries); //using count() method

echo '<br/>';
echo sizeof($countries); // using sizeof() method

//Get multidimensional array size

$exchange_rates=array(
'USD'=>array('buying'=>125.00,'selling'=>129.00),
'EUR'=>array('buying'=>165.42,'selling'=>172.81),
'GBP'=>array('buying'=>197.33,'selling'=>205.11)
);

echo count($exchange_rates,true); //          this          will          out          put          9
//or
echo count($exchange_rates,COUNT_RECURSIVE);

```



```
echo count($exchange_rates,COUNT_NORMAL); // this will out put 3  
?>
```

*~~~ End of Article ~~~*



## Session 18: Handling Databases with PHP

### Database Access

|                          |                                                                                                                   |
|--------------------------|-------------------------------------------------------------------------------------------------------------------|
| <b>Source</b>            | <a href="http://www.toves.org/books/php/ch07-db/index.html">http://www.toves.org/books/php/ch07-db/index.html</a> |
| <b>Date of Retrieval</b> | 03/09/2013                                                                                                        |

One of the most common applications of PHP is to provide a Web interface for accessing a **database**. The database may hold information about user postings for a forum, account and order information for a vendor, or raw data for a statistics site.

Because databases are so common, there are a special category of programs written specifically for managing databases, called **database management systems** (more often referenced by their abbreviation **DBMS**). Typical users don't often know that they're dealing with DBMSs, so the names of specific products aren't widely known. Some of the more popular commercial DBMS packages are Oracle and Microsoft SQL Server, but there are also two prominent open-source DBMS packages, MySQL and PostgreSQL.

We'll use MySQL here, since it is easily available and used quite frequently for Web pages in conjunction with PHP. In fact, PHP's interfaces for interacting with the different DBMSs are all quite similar, so if you wanted to use another, it wouldn't take much to learn.

Most DBMSs represent a database as a set of **tables**, each of which has a relatively fixed set of **columns**, and a more dynamic set of **rows**. A row represents a single entity, and a column represents an attribute of an entity. The simplest thing is to look at an example: Suppose we are managing a Web forum using a database. One thing we might want would be a table listing all participants in the forum. Each participant would have a distinctive user ID, a password, a real name, an e-mail address, and the year the participant joined the forum; each of these attributes will be a column in our table.

**Users table**

| <b>userid</b> | <b>passwd</b> | <b>name</b>     | <b>email</b>      | <b>year_joined</b> |
|---------------|---------------|-----------------|-------------------|--------------------|
| sherlock      | pipe          | Sherlock Holmes | 221b@bakerst.uk   | 1878               |
| marple        | knitting      | Miss Marple     | marple@trenton.uk | 1923               |
| nancy         | magnifier     | Nancy Drew      | nancy@trenton.us  | 1958               |

Of course, the forum database will contain other tables, too. One table it will certainly have would list all posts made in the forum. The following table describes the columns that we will have in this table.

### Columns in the `Posts` table

`poster` the ID of the person posting the message, corresponding to the `userid` field of `Users`

`postdate` the date and time that the message was posted

`subject` the subject of the message, as entered by the poster

`body` the body of the message, as entered by the poster

## 7.2. Simple SQL retrieval

We will want to ask the DBMS to retrieve information for us. There is a language designed specifically for this, called **SQL** (an acronym for Structured Query Language). SQL is so widely popular that it is closely identified with high-quality DBMSs — which is why so many of the popular DBMSs, including MySQL, incorporate the name SQL into their name.

SQL is a language for composing **queries** for a database. It is not a full-fledged programming language, but it is powerful enough to represent all of the typical operations on a database. PHP scripts send requests to the database using SQL, so we need to learn some of its fundamentals.

For now, we'll look only at the simplest version of the most important SQL command: the **SELECT query**. A simple **SELECT** query is composed of three separate clauses: a **SELECT** clause listing the names of the columns whose values we want to see, a **FROM** clause saying which table we want to access, and a **WHERE** clause indicating which rows we want to retrieve from that table.

For example, suppose we wanted to look up Sherlock Holmes' password and e-mail address. The SQL query we would want to send to the DBMS is the following.

```
SELECT passwd, email
FROM Users
WHERE name = 'Sherlock Holmes'
```

Note that the **WHERE** clause is a condition. In this case, we want to compare the `name` of each row to the string Sherlock Holmes (in SQL, strings are enclosed in single quotes). When we find a row that matches, we want the DBMS to send back the columns named in the **SELECT** clause.

The **WHERE** clause can include several conditions joined by **AND** and/or **OR**.

```
SELECT name
FROM Users
WHERE year_joined > 1900 AND year_joined < 1970
```

If the table held the three people listed above, this SQL query would result in two values: Miss Marple and Nancy Drew.

### 7.3. PHP database functions

PHP provides access to MySQL databases via a number of functions. We'll find six of them particularly useful. They are listed below in the order they will typically be used.

```
$db = mysql_connect($dbms_location)
```

Connects to the DBMS whose address is identified by the parameter, which should be a string. The exact string will depend on where exactly the DBMS is located; an example string is localhost:/export/mysql/mysql.sock. The function returns a value that can later be used to refer to the DBMS connection.

```
mysql_select_db($db_name, $db)
```

Selects which database managed by the DBMS that this PHP connection will reference. The DBMS will have a name for each database, and the name should be passed as a string for the first parameter. The second parameter should be a reference to the DBMS connection as returned by `mysql_connect`.

```
$result = mysql_query($sql, $db)
```

Sends a query to the DBMS. The SQL code should be located in the string passed as the first parameter; the second parameter should be a reference to the DBMS connection as returned by `mysql_connect`. The function `mysql_query` returns a value that allows access to whatever the results were of the query; if the query failed — as for example would happen if there were an error in the given SQL — this return value would be `FALSE`.

```
mysql_error()
```

If the last executed query failed for some reason, `mysql_error` returns a string that describes this error. I recommend that you always make sure your PHP code somehow echoes the result of `mysql_error` when a query fails, so that you have a way to debug your script.

```
mysql_num_rows($result)
```

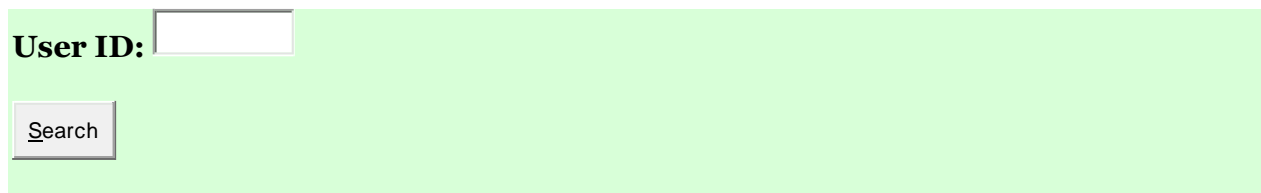
Returns the number of rows found by the SQL query whose results are encapsulated in the parameter. Of course, this could be 0 if there were no rows that resulted from the query.

```
mysql_result($result, $row, $col)
```

Returns a string holding the information found in row `$row` and column `$col` in the results that are encapsulated by the `$result` parameter. Both rows and columns are numbered starting from 0: That is, the first row is numbered 0, the second row 1, and so on, and similarly the first column is numbered 0.

#### 7.4. Example database access

Let's look at an example where these functions turn out to be useful. Suppose we want to write a script that allows a user of our Web site to view the information about a particular participant. The Web site user would see a form such as the following.



This form can be generated by the following HTML code.

```
<form method="post" action="user.php">
<p><b>User ID:</b> <input type="text" name="userid" /></p>
<p><input type="submit" value="Search" /></p>
</form>
```

The file `user.php` would be the following.

```
<?php
import_request_variables("pg", "form_");

$db = mysql_connect("localhost:/export/mysql/mysql.sock");
mysql_select_db("forum", $db);
$sql = "SELECT name, email"
      . " FROM Users"
      . " WHERE userid = '$form_userid'";
$rows = mysql_query($sql, $db);
if(!$rows) {
```

```

        $error = "SQL error: " . mysql_error();
    } elseif(mysql_num_rows($rows) == 0) {
        $error = "No such user name found";
    } else {
        $error = FALSE;
        $user_name = mysql_result($rows, 0, 0);
        $user_email = mysql_result($rows, 0, 1);
    }
?>
<html>
<head>
<title>User information</title>
</head>

<body>
<?php
    if($error) {
        echo "<h1>Error accessing user information</h1>\n";
        echo "<p>$error</p>\n";
    } else {
        echo "<h1>Information about $form_userid</h1>\n";
        echo "<p>User ID: <tt>$form_userid</tt></p>\n";
        echo "<p>Real name: $user_name</p>\n";
        echo "<p>E-mail address: <tt>$user_email</tt></p>\n";
    }
?>
</body>
</html>

```

Notice in particular the SQL generation within the PHP code. This program first constructs a string containing the SQL code, referenced by a variable `$sql`. Because this string is fairly long, and because it is easier to read with each clause on a separate line, the PHP splits the generation of this string across three different lines using the catenation operator. (Notice how there are not semicolons at the end of the first two lines: This is what leads PHP to consider all three lines as part of the same statement, as a semicolon marks the end of the current statement.) Whatever name the user typed in the text field is inserted into the SQL string in the appropriate place.

After executing the query via `mysql_query`, this program first checks whether there is an error, then it verifies that a result was found using `mysql_num_rows`; and if there is a

result, then it dumps the result into some variables. The PHP closes the connection and then proceeds to echo information about the result back to the user.

Notice that `mysql_result` in our example retrieves row 0 and column 0 from the result to retrieve the first column of the first row. This numbering scheme isn't the intuitive approach, but it's the way that PHP tends to number things.

*~~~ End of Article ~~~*



## More SQL

|                   |                                                                                                                             |
|-------------------|-----------------------------------------------------------------------------------------------------------------------------|
| Source            | <a href="http://www.toves.org/books/php/ch09-moresql/index.html">http://www.toves.org/books/php/ch09-moresql/index.html</a> |
| Date of Retrieval | 03/09/2013                                                                                                                  |

We now return to SQL to get a more thorough spectrum of the types of queries that one can use to access a database. We won't see any PHP concepts in this chapter, though we'll get more practice with what we know. Our primary goal is to extend our SQL knowledge beyond the basic **SELECT** queries that we saw before.

The first extension we'll make is to add one more clause to our **SELECT** query: After the **SELECT**, **FROM**, and the optional **WHERE** clause we can also have an **ORDER BY** clause to specify in what order we want the returned rows to be.

In the previous chapter, we saw how we could list all of the posts in our forum; but MySQL would have the right send them in no particular order, which isn't the behavior we would want. To amend this, we would add an **ORDER BY** clause to our query. (This clause needs to come after the **WHERE** clause; in the case of a query like this without a **WHERE** clause, it belongs after where the **WHERE** clause would be.)

```
SELECT subject, body
FROM Posts
ORDER BY postdate
```

The posts will now be retrieved in order, starting with the earliest post. If we wanted to report the most recent post first, we can do this by adding **DESC** afterwards.

```
SELECT subject, body
FROM Posts
ORDER BY postdate DESC
```

You may list several values in the **ORDER BY** clause, separated by commas. The results will be ordered by the first-mentioned value, then the second-mentioned value would break ties; then the subsequent values.

### 9.2. Joins

A more sophisticated type of **SELECT** query is the *join query*, where we list multiple tables in its **FROM** clause. We can then draw out information from the various tables.



```
SELECT subject, body, name
FROM Posts, Users
ORDER BY postdate
```

This won't work as you probably expect, however: A simple SQL join will join *every* row from the first table with *every* row from the second table. Thus a post by thesherlock user will be joined with each of sherlock, marple, and nancy, each yielding a separate row. If there are 5 posts in the database, and our Web page used the above query, it would list 15 posts, once for each combination of a post with a user. This is certainly not what we wanted.

To select only those results where the post and user rows correspond, we need to add that requirement into the **WHERE** clause.

```
SELECT subject, body, name
FROM Posts, Users
WHERE poster = userid
ORDER BY postdate
```

This is the query we'll use in our updated version of the page that we saw in the previous chapter for listing all current posts. This page has also been modified to use the **for** statement that we saw at the end of the previous chapter.

```
<?php import_request_variables("pg", "form_"); ?>
<html>
<head>
<title>Current Posts</title>
</head>

<body>
<h1>Current Posts</h1>

<?php
    $db = mysql_connect("localhost:/export/mysql/mysql.sock");
    mysql_select_db("forum", $db);
    $sql = "SELECT subject, body, name"
        . " FROM Posts, Users"
        . " WHERE poster = userid"
        . " ORDER BY postdate";
    $rows = mysql_query($sql, $db);
    if(!$rows) {
        echo "<p>SQL error: " . mysql_error() . "</p>\n";
    } elseif(mysql_num_rows($rows) == 0) {
```

```

        echo "<p>There are not yet any posts.</p>\n";
    } else {
        for($row = 0; $row < mysql_num_rows($rows); $rows++) {
            $post_subject = mysql_result($rows, $row, 0);
            $post_body = mysql_result($rows, $row, 1);
            $post_name = mysql_result($rows, $row, 2);

            echo "<h2>$post_subject</h2>\n";
            echo "<p>By: $post_name</p>\n";
            echo "<p>$post_body</p>\n";
        }
    }
}
?>
</body>
</html>

```

### 9.3. Inserts

Another common thing one would want to do with a database via the Web is to insert new items into a table. This is done via a different type of SQL query, called an **INSERT** query. As an example of an SQL query, suppose we want a PHP script that adds a new post into the database. The relevant SQL query would be the following.

```

INSERT INTO Posts (poster, postdate, subject, body)
VALUES ('sherlock', '2007-07-04 03:23', 'Case closed', 'The butler did it.')

```

An **INSERT** query has two clauses, the **INSERT INTO** clause and the **VALUES** clause. The **INSERT INTO** clause starts with the name of the table into which we wish to insert (**Posts** here), followed by a set of parentheses enclosing a list of the names of columns whose values we will specify for this new row. The **VALUES** clause consists of a set of parentheses enclosing the values for the columns named in the **INSERT INTO** clause, in the same order. Incidentally, you are allowed to omit most columns from the **INSERT** query, in which case the DBMS will choose some default value for the newly created row.

Unlike a **SELECT** query, an **INSERT** query doesn't really have any results: The information is going into the database, not coming out. As a result, you'd have no reason in your PHP script to use the `mysql_result` function after executing an **INSERT** query.

Let us now look at a PHP script that will execute an **INSERT** query. First, we need to specify the form that the user will complete.

```

<form method="post" action="post.php">
<p>Name: <input type="text" name="user" />

```

```

<br />Password: <input type="password" name="passwd" />
<br />Subject: <input type="text" name="subj" />
<br />Body: <input type="text" name="body" />
<br /><input type="submit" value="Post" />
</p></form>

```

Note that we have the user enter a password. Our PHP script will check the password first using a **SELECT** query; if that query finds a row with the relevant user/password combination, then it will proceed to add the post using a **INSERT** query.

```

<?php
    import_request_variables("pg", "form_");

    $db = mysql_connect("localhost:/export/mysql/mysql.sock");
    mysql_select_db("forum", $db);
    $sql = "SELECT userid"
        . " FROM Users"
        . " WHERE userid = '$form_user' AND passwd = '$form_passwd'";
    $rows = mysql_query($sql, $db);
    if(!$rows) {
        $message = "Password lookup error: " . mysql_error();
    } elseif(mysql_num_rows($rows) == 0) {
        $message = "Password not accepted.";
    } else {
        $sql = "INSERT INTO Posts (poster, postdate, subject, body)"
            . " VALUES ('$form_user', NOW(), "
            . "          '$form_subj', '$form_body')";
        $rows = mysql_query($sql, $db);
        if(!$rows) {
            $message = "Insert error: " . mysql_error();
        } else {
            $message = "The post was successfully inserted.";
        }
    }
}

?>
<html>
<head>
<title>Post requested</title>
</head>
<body>
<h1>Post requested</h1>

```

```
<p><?php echo $message; ?></p>

<p>[<a href="view.php">List Posts</a>]</p>
</body>
</html>
```

## 9.4. Other SQL

SQL provides many other types of queries, but they occur less often in PHP scripts than **SELECT** and **INSERT** queries. One that you may find useful, though, is the **UPDATE** query, which says to alter existing rows in a table. Let us look at an example that changes the password for a user.

```
UPDATE Users
SET passwd = 'sillyhat'
WHERE userid = 'sherlock'
```

The **UPDATE** clause specifies which table contains the row we wish to modify; the **SET** clause says how we want to change that row; and the **WHERE** clause provides a condition for identifying the row to modify. In fact, we are permitted to use a **WHERE** clause that matches multiple rows in the table: MySQL will simply apply the **SET** clause for all rows for which the **WHERE** clause holds.

Another thing you occasionally might want to do via PHP is to delete rows from a table. This is done, appropriately enough, with a **DELETE** query, which consists of a **DELETE FROM** clause specifying the table from which we wish to drop rows, and a **WHERE** clause specifying a condition for identifying which rows to drop. The following example will delete all posts from before 2006.

```
DELETE FROM Posts
WHERE postdate < '2006-01-01'
```

Of course, you would want to be careful with a **DELETE** query, because if you mess up your **WHERE** clause you could end up clobbering the entire table. Generally speaking, PHP scripts won't delete information: The database will only accumulate information. SQL provides many other query types, but they are targeted toward managing a database, such as creating tables, or renaming a column, or adding new columns to an existing table.

~~~ End of Article ~~~



## Session 19: Working with Cookies

### Bake Cookies Like a Chef

|                   |   |
|-------------------|---|
| Source            | <a href="http://webadvent.org/2011/bake-cookies-like-a-chef-by-michael-nitschinger">http://webadvent.org/2011/bake-cookies-like-a-chef-by-michael-nitschinger</a> |
| Date of Retrieval | 03/09/2013  |

It's Christmas time again. While you write the first lines of your shiny new authentication library, the smell of tasty cookies is slowly sneaking into your room. The few minutes until they are finished is a good time to lean back and think about cookies — not the ones in the kitchen, but the cookies in your browser.

Let's explore how to securely store data on the client side, and how to detect unwanted modifications.

#### ***Ingredients***

Nearly every web app needs some kind of authentication, and most use cookies to help with identity. Cookies are a very convenient, portable, and scalable method to identify users after they have been authenticated. Unfortunately, you have to store those cookies on the client. The phrase "all user input is evil" is often used when you have to deal with data from the client, so it is important to realize that cookies are under the client's control.

The first thing to remember is that cookies can provide more information to the user that you might think. For example, if your cookie contains a **PHPSESSID** key, an attacker doesn't need fancy attack tools to immediately recognize that your app probably uses PHP. Every bit of information is important for an attacker, even if it doesn't *seem* important.

An example from the Microsoft world is if you see **ISAWPLB** in a cookie, chances are that the target app uses the ISA Server as a reverse proxy or load balancer. Just look at the cookies of apps that you use regularly, and Google for some of the keys. Attackers can use this information in ways you would never consider. If you don't believe me, take a look at the presentation entitled [How I Met Your Girlfriend](#), by Samy Kamkar, which was presented at DEFCON 18. He shows you how to reduce the entropy of PHP sessions from 160 bits to 20 bits, which is much easier to brute force.

Attackers may also use cookies to bypass your authentication mechanisms. For example, if you implement a "remember me" feature by storing the user ID in the cookie (and no

additional checks are performed on the server side), then the attacker may randomly change the ID in the cookie until he finds a valid one. It's not very hard to guess that admin users have low IDs, right? If you want to learn more about attack vectors like this, I recommend you to start with the [OWASP Top Ten Project](#) where you can read about the top ten security risks in web apps, and how you can prevent them.

Does all of this mean that you should avoid cookies altogether? Of course not. Let's take a look at two ways to encrypt your cookies and detect possible changes made to them. Note that the following code snippets are taken from the [Lithium](#) core. (Make sure to check it out if you haven't already!) Additionally, I've modified the code snippets a bit to make them easier to understand. Lithium handles the following techniques transparently, so in practice, you only add and read data and don't need to care about the implementation details.

### Directions

A secure way to ensure that no one has tampered with your cookies is called [HMAC](#) (Hash-based Message Authentication Code). In a nutshell, your cookie gets signed with a hash that is generated from your data and a secret password. As a result, if the data changes, the hash will change, too.

PHP provides a handy `hash_hmac()` function that does the actual work for you. Here's the code snippet from Lithium:

```
public static function signature($data, $secret = null) {
    unset($data['__signature']);
    $secret = ($secret) ?: static::$_secret;
    return hash_hmac('sha1', serialize($data), $secret);
}
```

It all boils down to calling `hash_hmac()` with an algorithm (in this case `sha1`), the actual data, and a secret password. Because you can only hash a string, you may need to serialize a non-scalar payload first. The signature will be appended to the payload, so you need to unset it first, so that the hash actually represents just the payload. It looks similar to this (let's pretend that the static `signature` method is wrapped in a `Cookie` class and is public):

```
// password and payload
$secret = 'phpadvent';
$data = array('christmas' => 'fun');

// create the signature
$signature = Cookie::signature($data, $secret);
```

```
// store the cookie
$data += array('__signature' => $signature);
setcookie('mycookie', serialize($data));
```

To verify the cookie, remove the payload and generate the hash again. If the two hashes don't match, Lithium raises a **RuntimeException**. This isn't strictly necessary, but it makes sense in certain environments (you can then write exception handlers that log attack events, send emails, or block further requests).

```
public function read($currentData, array $options = array()) {
    // ....

    $currentSignature = $currentData['__signature'];
    $signature = static::signature($currentData);

    if ($signature !== $currentSignature) {
        $message = "Possible data tampering: HMAC signature does not match
data.";
        throw new RuntimeException($message);
    }
    return $data;
}
```

I recommend you read both the `hash_hmac()` [documentation](#) and the actual [implementation](#) in Lithium.

Message digests are a good way to detect data tampering, but in a lot of cases, it is an even better idea to encrypt all of your data. In order to do this, make sure to have the **mcrypt** extension installed. I recommend you use the [AES](#) algorithm in [CBC](#) mode (don't use ECB, as it is considered [far less secure](#)).

The code for encryption is a bit more complicated, but it's definitely worth the trouble.

```
protected static $_vector = null;
public function __construct($secret) {
    $this->_config = array(
        'secret' => $secret,
        'cipher' => MCRYPT_RIJNDAEL_256,
```

```
        'mode' => MCRYPT_MODE_CBC,
        'vector' => static::_vector(MCRYPT_RIJNDAEL_256,
MCRYPT_MODE_CBC)
    );
}

protected static function _vector($cipher, $mode) {
    if (static::$_vector) {
        return static::$_vector;
    }

    $size = static::_vectorSize($cipher, $mode);
    return static::$_vector = mcrypt_create_iv($size, MCRYPT_DEV_URANDOM);
}

protected static function _vectorSize($cipher, $mode) {
    return mcrypt_get_iv_size($cipher, $mode);
}

public function encrypt($decrypted = array()) {
    $cipher = $this->_config['cipher'];
    $secret = $this->_config['secret'];
    $mode    = $this->_config['mode'];
    $vector  = $this->_config['vector'];

    $encrypted = mcrypt_encrypt($cipher, $secret, serialize($decrypted),
    $mode, $vector);

    $data = base64_encode($encrypted) . base64_encode($vector);

    return $data;
}

public function decrypt($encrypted) {
    $cipher = $this->_config['cipher'];
    $secret = $this->_config['secret'];
    $mode    = $this->_config['mode'];
```



```

        $vector = $this->_config['vector'];

        $vectorSize = strlen(base64_encode(str_repeat(" ",
static::_vectorSize($cipher, $mode))));

        $vector = base64_decode(substr($encrypted, -$vectorSize));
        $data = base64_decode(substr($encrypted, 0, -$vectorSize));

        $decrypted = mcrypt_decrypt($cipher, $secret, $data, $mode, $vector);
        $data = unserialize(trim($decrypted));

        return $data;
}

```

The actual work is done by the `mcrypt_*` methods. (Lithium provides internal wrappers.) The constructor (not shown here) creates a vector (by calling `_vector()`) and sets the cipher (`MCRYPT_RIJNDAEL_256`), the mode (`MCRYPT_MODE_CBC`), and a custom secret. When `encrypt()` is called, the unencrypted payload is serialized and encrypted with `mcrypt_encrypt()`. Because the encrypted data may contain incompatible characters for cookies, we need to `base64_encode()` it. Decrypting works in exactly the opposite way (decoding, decrypting, and unserializing).

Here's an example:

```

// password and payload
$secret = 'phpadvent';
$data = array('christmas' => 'fun');

// encrypting
$cookie = new Cookie($secret);
$encrypted = $cookie->encrypt($data);

/* Possible output:
* string(88)
"aAH84W30XJTRC1aFdvleJdv3H0Dzj/TPLVSYKyWe2yo=LtoWXG8ewGK6btLn02OLGsOfTc6T97TLww
ogmMXORHI="
*/
var_dump($encrypted);

// of course, we can decrypt it again

```

```
$decrypted = $cookie->decrypt($encrypted);

/* Output:
* array(1) { ["christmas"]=> string(3) "fun" }
*/
var_dump($decrypted);
```

If you are very careful, you can first add an HMAC signature to your payload and then encrypt it all.

## **Tasting**

Now that we know the ingredients, lets bake secure cookies with Lithium:

```
use lithium\storage\Session;

Session::config(array('default' => array(
    'adapter' => 'Cookie',
    'strategies' => array('Encrypt' => array('secret' => 'p$hp#4dv3nTT'))
));

Session::write('mykey', 'myvalue');
```

That was easy, right? Lithium provides you with a transparent abstraction to both session adapters (cookies and native PHP), and security strategies (HMAC and encryption), so you can mix them as you like. You can even write your own adapters (like a custom database adapter) and still profit from encryption strategies out of the box.

The key message I want you to take away is to take care of your cookies! Don't store a single bit more than necessary on the client side, encrypt it, and make sure it was not modified outside of your control. PHP and the **mcrypt** extension provide you with everything you need, and good frameworks provide you with handy abstractions, so use them!

~~~ End of Article ~~~



## Session 21: Session Management in PHP

### Sessions

|                          |                                                                                                                                     |
|--------------------------|-------------------------------------------------------------------------------------------------------------------------------------|
| <b>Source</b>            | <a href="http://interoperating.info/courses/introphp/intro/sessions">http://interoperating.info/courses/introphp/intro/sessions</a> |
| <b>Date of Retrieval</b> | 03/09/2013                                                                                                                          |

HTTP is a 'stateless' protocol, which means it has no built in way of keeping track of users between requests. In this, it is quite different from other Internet protocols like FTP, Telnet, and SSH, all of which establish a session between the client and the server as the first order of business. That's why you always have to log in to an FTP server (even when connecting anonymously), but can connect to most websites without any kind of login process up front.

This radical simplicity is part of what gives HTTP its great power and accessibility, but it comes at a cost. For all but the simplest web transactions, it's usually necessary to establish some kind of server-side awareness of what individual users are doing. For example, a shopping cart application needs to have some way of remembering what items it contains, and who those items are associated with. You can't do this with straight HTTP.

So if the protocol layer doesn't handle sessions, responsibility for establishing and maintaining sessions must be pushed up the stack to the application layer. Which is where PHP comes in.

### Cookies

PHP and pretty much every other web application use cookies to maintain sessions. Cookies aren't native to PHP, they are simply part of the client/server environment in which PHP operates. Briefly, a cookie is a small token originated on a particular server, and stored by the user's browser. The cookie is returned to the originating server every time the browser sends an http request to that server.

Cookies can't store much information themselves, but they can provide a unique token, or signature, by which the applications on the server can recognize that requests are coming from the same client. Such tokens are generally called session cookies. Data can be stored on the server and associated with the user's session cookie, making it possible for web applications to store server-side data and associate it with a particular user. That's typically how shopping cart applications remember what's in your cart.

## Session Functions

PHP has a number of built-in functions that make it relatively straightforward to handle sessions. Let's take a look at an example:

```
<?php

session_start();

$_SESSION['name'] = 'J.Q. Hacker';
$_SESSION['email'] = 'jqhacker@2600.org';

echo '<p><a href="session2.php">Go to the next page</a></p>';

?>
```

Copy that code into a file, and save it as *session1.php* in your htdocs folder. Feel free to change the 'name' and 'email' values in the `$_SESSION` array, if you like.

```
<?php

session_start();

echo $_SESSION['name'] . "<br />";
echo $_SESSION['email'] . "<br />";

echo session_id() . "<br />";

?>
```

Copy the above code into a file, and save it as *session2.php* in your htdocs folder. Open up *session1.php* in your browser, and then click through to *session2.php*. If all goes well, you should see the values entered in the first script displayed in the second.

## Client Side Sessions

It might be possible, in theory, to pass all of the session information back to the client, and then send it back to the server using hidden form values. This would in effect maintain state on the client rather than on the server. Back the mid-90s in fact there were many web applications that did this, but developers quickly realized the practice was cumbersome and insecure. Hidden form values are easily modifiable by the user, so must be rigorously checked every time they are processed by the application. And schlepping all that data across the network repeatedly was a waste of bandwidth and computing power.

So what exactly is going on here? First off, PHP's session-handling capabilities were invoked by the 'session\_start()' function. 'session\_start()' is a bit misleading as a function name. It does start a new session if none exists, but it also re-establishes an already-existing session. This is why it needs to be invoked in both scripts, not just the first one.

On the following lines you see a variable called `$_SESSION`. As you've probably already guessed, this is another of PHP's reserved variables (like `$_REQUEST` and `$_SERVER`, which we've seen previously). `$_SESSION` is an associative array where session values are stored.

In this case, a name and an email address are stored with appropriate keys in the `$_SESSION` array. Keys and values in the `$_SESSION` array can be whatever you want, within the bounds of what's permissible in PHP arrays generally.

In `session2.php`, the session is re-invoked by the `session_start()` function. This gives the script access to the `$_SESSION` array, where the name and email values were stored in the previous script. Since this script knows the keys for those values, it can access them directly, and echo them back to the browser.

At the end of the second script, you see another function, `session_id()`. That returns the internal value PHP is using to reference this particular session. If you didn't like that value, you could use the `session_id()` function to change it, simply by supplying a value to the function. For example, `session_id('jqhackers_session')` would set the value accordingly. Note that it's hugely important for `session_ids` to be unique, unless you want your users putting stuff in each others shopping carts.

Notice that there is a bunch of stuff going on here that PHP is doing under the hood. A cookie is being sent to the user's browser, a session ID is created, values are being stored and retrieved on the server, and you didn't have to handle any of the mechanics of getting that done. The session-handling functions did it all for you.

*~~~ End of Article ~~~*



## Session 22: Handling E-mail with PHP

### Sending Mail Attachment(s) with PHP Mail Function: Rules

|                   |                                                                                                                                                                                       |
|-------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Source            | <a href="http://i-code-today.blogspot.in/2009/01/sending-mail-attachments-with-php-mail.html">http://i-code-today.blogspot.in/2009/01/sending-mail-attachments-with-php-mail.html</a> |
| Date of Retrieval | 03/09/2013                                                                                                                                                                            |

#### The Rules:

1. In the mail header few header elements (like "From ", "Cc ", "Bcc", "MIME-Version" ) will remain as it is, irrespective of the fact that the mail has any attachment or not.
2. a. If a mail doesn't contain any attachment, then **Content-type** will be "**text/html**" (for html formatted mail) or "**text/plain**" ( for plain text mail) . Content-type should have an additional attribute "charset" to define character encoding of the message.  
  
b. In case when mail consists of one or more attachments, **Content-type** will be "**multipart/mixed**". Content-type will also have an attribute called "boundary". The "**boundary**"attribute contains a *unique signature* which is use to denote the starting point of each message/attachment of the mail as well as the end of mail.
3. a. In case of mail without attachment the **message string** should be passed to mail function as *message parameter*.  
  
b. If mail contains one or more attachments, then the **message** and the **attachment file** contents will be attached to the header string, separated by a boundary notation  
  
-- [boundary value]  
  
the start message will also have similar boundary notation to denote the start of message and A blank string or null value should be passed to the message parameter of mail function.
4. In multipart mail each *message/attachment will have their own Content-type* . For a multipart mail header the message will have Content-type "text/html" (for html formatted mail) or "text/plain" (for plain text mail). And attachment will have Content-type depending upon the type of content. If you are not sure about content use **Content-type "application/octet-stream"**. For attachment content type Additional Attribute name can be use to denote the base name of the attachment file.
5. Each attachment block also contain header "**Content-Transfer-Encoding**" to specify the *content encoding*

6. Each attachment block will have **Content-Disposition** as "*attachment*" and attribute **filename** will denote the *name of the downloaded file*.
7. After the end of all attachments there will be end of mail notation

~~~ End of Article ~~~



## Session 24: OOP Concepts

### Magic Methods in PHP

|                          |   |
|--------------------------|---|
| <b>Source</b>            | <a href="http://www.techflirt.com/tutorials/oop-in-php/magic-methods-in-php.html">http://www.techflirt.com/tutorials/oop-in-php/magic-methods-in-php.html</a> |
| <b>Date of Retrieval</b> | 03/09/2013  |

Here we will discuss some of the most common magic methods of PHP which will be used in object-oriented programming. First, let us review all magic methods with short descriptions.

Magic

#### List of List of Magic Methods in PHP

| <b>Magic Method</b>       | <b>Description</b>   |
|---------------------------|--|
| <code>__construct</code>  | This magic method is called when someone creates an object of your class. Usually this is used for creating a constructor in PHP5.   |
| <code>__destruct</code>   | This magic method is called when an object of your class is unset. This is just the opposite of <code>__construct</code> .   |
| <code>__get</code>        | This method is called when your object attempts to read a property or variable of the class which is inaccessible or unavailable.  |
| <code>__set</code>        | This method is called when an object of your class attempts to set a value of the property which is really inaccessible or unavailable in your class.  |
| <code>__isset</code>      | This magic method triggers when <code>isset()</code> function is applied on any property of the class which is inaccessible or unavailable.  |
| <code>__unset</code>      | <code>__unset</code> is something opposite of <code>isset</code> method. This method triggers when <b><code>unset()</code></b> function is called on an inaccessible or unavailable property of the class. |
| <code>__call</code>       | <code>__call</code> magic method triggers when you are attempting to call a method or function of the class which is either inaccessible or unavailable.   |
| <code>__callstatic</code> | <code>__callstatic</code> executes when an inaccessible or unavailable method is in a static context.  |
| <code>__sleep</code>      | <code>__sleep</code> methods trigger when you are going to serialize your class object.  |
| <code>__wakeup</code>     | <code>__wakeup</code> executes when you are unserializing any class object.  |



---

`__toString`      `__toString` executes when you are using echo on your object.

---

`__invoke`      `__invoke` called when you are using object of your class as function

---

Above list is the most common used magic methods in php object oriented programming. Above magic methods of php executes on some specific events occur on your class object. For example if you simply echo your object then `__toString` method trigger. Let us create group of related magic method and analyze how it is working.

### **`__construct` and `__destruct` magic method in PHP**

`__construct` method trigger on creation of object. And `__destruct` triggers on deletion of object. Following is very basic example of **`__construct`** and **`__destruct`** magic method in php:

```
class test
{
function __construct()
{
echo 1;
}
function __destruct()
{
echo 2;
}
}

$objT = new test(); // __construct get automatically executed and print 1 on
screen

unset($objT); // __destruct triggers and print 2.
```

### **`__get`, `__set`, `__call` and `__callStatic` Magic Methods**

`__get`, `__set`, `__call` and `__callStatic` all magic methods in php directly related with non accessible method and property of the class.

**`__get`** takes one argument and executes when any inaccessible property of the method is called. It takes name of the property as argument.

**`__set`** takes two property and executes when object try to set value in inaccessible property. It take first parameter as name of the property and second as the value which object is try to set.

**`__call`** method fires when object of your class is trying to call method of property which is either non accessible or not available. It takes 2 parameter First parameter is string and is

name of function. Second parameter is an array which is arguments passed in the function.

**\_\_callStatic** is a static magic method. It executes when any method of your class is called by static techniques.

Following is example of `__get` , `__set` , `__call` and `__callStatic` magic methods

```
class test
{
function get($name)
{
echo "__get executed with name $name ";
}
function set($name , $value)
{
echo "__set executed with name $name , value $value";
}
function __call($name , $parameter)
{
$a = print_r($parameter , true); //taking recursive array in string
echo "__call executed with name $name , parameter $a";
}
static function callStatic($name , $parameter)
{
$a = print_r($parameter , true); //taking recursive array in string
echo "__callStatic executed with name $name , parameter $a";
}
}
$a = new test();
$a->abc = 3; // __set will executed
$app = $a->pqr; // __get will triggered
$a->getMyName('ankur' , 'techflirt' , 'etc');// __call will be executed
test::xyz('1' , 'qpc' , 'test');// __callstatic will be executed
```

`__isset` and `__unset` magic methods

`__isset` and `__unset` magic methods in php are opposite of each other.

**\_\_isset** magic methods executes when function **isset()** is applied on property which is not available or not defined. It takes name of the parameter as an argument.

**\_\_unset** magic method triggers when unset() method is applied on the property which is either not defined or not accessible. It takes name of the parameter as an argument.

Following is example of \_\_isset and \_\_unset magic method in php

```
class test
{
function __isset($name)
{
echo " __isset is called for $name";
}
function __unset($name)
{
echo " __unset is called for $name";
}
}
$a = new test();
__isset($a->x);
__unset($a->c);
```

~~~ End of Article ~~~



## Session 26 : Generator Delegation and Throwable Interface

**Source** <https://pantheon.io/docs/php-errors/>

### PHP Errors and Exceptions

There are three basic kinds of PHP errors:

- **Notice:** room for improvement; typically unset variables or missing array keys.
- **Warning:** errors will probably occur if not addressed.
- **Error:** fatal, execution terminated. Often known as the "white screen of death".

Each of the PHP errors are handled differently depending on the site environment. On Dev, they are shown directly to the user in the browser. On Test and Live, PHP errors are not displayed to users, but they'll still be logged. Notices and warnings are logged in the database logs if `db\_log` is enabled for Drupal. The PHP constants `WP_DEBUG` and `WP_DEBUG_LOG` can be enabled for WordPress to save errors to `wp-content/debug.log`. PHP errors are also logged on the application server at `logs/php-error.log`.

Here's a breakdown of what errors are shown and where:

| Environment | Severity | Browser | Watchdog | logs/php-error.log |
|-------------|----------|---------|----------|--------------------|
| Dev         | notice   | Y       | Y        | N                  |
|             | warning  | Y       | Y        | N                  |
|             | error    | Y       | N        | Y                  |
| Test        | notice   | N       | Y        | N                  |
|             | warning  | N       | Y        | N                  |
|             | error    | N       | N        | Y                  |
| Live        | notice   | N       | Y        | N                  |
|             | warning  | N       | Y        | N                  |
|             | error    | N       | N        | Y                  |

### PHP Errors Slow Down a Site

An error, no matter what severity, is a problem that needs to be addressed. Any PHP error, even a notice, will drastically reduce the speed of PHP execution. Even if you don't see the error in your browser, and even if you explicitly disable logging, every single PHP error will slow your site down.

If database logging is enabled, your site will be even slower, requiring a database write for every error. However, disabling logging does not address the problem, it only hides the symptom.

Best practice is to fix every notice, warning, and error as you discover them. If they're in an extension (WordPress plugin or Drupal module), roll a patch and submit it to the project's issue queue.

## PHP Unhandled Exceptions on Pantheon

A PHP exception is a mechanism for defining error conditions and how to handle them. For more details on Exceptions, see the PHP documentation on Exceptions..

PHP Exceptions are errors, and depending on the severity and whether they are handled correctly can crash your site. As Exceptions are created in code and not by PHP itself, they are not logged in the PHP error log file and will not be visible in the Pantheon Dashboard. By default, Drupal will log exceptions to Watchdog.

## Handling Undefined Index Notices for PHP Variables

When you import your site or enable some new modules, PHP notices may be reported on your Dev site that have never been reported before. These notices are now being made apparent because of the Dev environment's strict error reporting level.

An example notice might look like this:

```
Notice: Undefined index: description in  
theme_imagefield_image_imagecache_lightbox2() (line 163 of  
/srv/bindings/xxxxxxx/code/sites/all/modules/contrib/lightbox2/light  
box2.formatter.inc) ..
```

Why is PHP reporting this?

Variable declaration is not required by PHP, but is a recommended practice that can help to avoid security vulnerabilities or bugs if one forgets to provide a value to a variable that be used later on. PHP issues an E\_NOTICE, a very low-level error, as a reminder.

No one is going to twist your arm about addressing these notices, but Pantheon believes that surfacing them in the Development environment will help developers address potential problems in the future before they can occur by following best practices.

## Fatal Error: require\_once(): Failed Opening Required

The `require_once()` function simply checks to see if a file has been included already, if it has not, then it will be included when checked.

When this error surfaces, it simply means that the file in question is not where it should be. For example, the error will look something like this:

```
Fatal error: require_once(): Failed opening required  
'/srv/bindings/xxxxx/code/sites/all/modules/redis/redis.autoload.inc'  
(include_path='.:usr/share/pear:usr/share/php') in  
/srv/bindings/xxxxxx/code/includes/bootstrap.inc on line 2394
```

To fix this error, look for the correct path to the file and update the `require_once()`.

# Exception Handling Syntax

**Source** [https://en.wikipedia.org/wiki/Exception\\_handling\\_syntax#PHP](https://en.wikipedia.org/wiki/Exception_handling_syntax#PHP)

Exception handling syntax varies between programming languages, partly to cover semantic differences but largely to fit into each language's overall syntactic structure. Some languages do not call the relevant concept 'exception handling'; others may not have direct facilities for it, but can still provide means for implementing it.

Most commonly, error handling uses a try...[catch...][finally...] block, and errors are created via a throw statement, but there is significant variation in naming and syntax.

Exception handling syntax varies between programming languages, partly to cover semantic differences but largely to fit into each language's overall syntactic structure. Some languages do not call the relevant concept 'exception handling'; others may not have direct facilities for it, but can still provide means for implementing it.

Most commonly, error handling uses a try...[catch...][finally...] block, and errors are created via a throw statement, but there is significant variation in naming and syntax.

Exception handling is only available in PHP versions 5 and greater.

## PHP

```
try
{
    // Code that might throw an exception

    throw new Exception('Invalid URL.');
```

```
catch (FirstExceptionClass $exception)
{
    // Code that handles this exception
}
```

```
catch (SecondExceptionClass $exception)
{
    // Code that handles a different exception
}
```

```
finally  
  
{  
    // Perform cleanup, whether an exception occurred or not.  
}
```

*~~~ End of Article ~~~*



# Why Throwing Exceptions is Better than Returning Error Codes?

**Source** <http://javierferrer.me/exceptions-vs-error-codes/>

In this post I would like to make a step by step guide specifying the pros and cons of every possible solution in order to illustrate why is better to throw an Exception than to return an error code. Also, we'll see how SOLID principles can be applied in this kind of scenarios in order to improve our Software Design and avoid code smells like the switch statement.

## 1. Base scenario – Error codes & switch statement

Here we have an example of a possible login validation service:

```
private function checkLogin() {  
    // ...  
    // Some validation to check if the credentials are valid  
    // ...  
    if ($hasNotValidCredentials) {  
        return -1;  
    }  
    // ...  
    // Some validation to check if the user has attempted too many times to  
login  
    // ...  
    if ($hasTooManyLoginAttempts) {  
        return -2;  
    }  
    // If all is OK, return "the successful code" 1  
    return 1;  
}
```

And here we have a probable use of this method

```
case -1:
```



```
// Invalid credentials case, log it into the error logs
$this->errorLogger->log("Invalid credentials");

break;

// Invalid credentials case, log it into the error logs
case -2:

    // Too many attempts case, log it into the error logs
    $this->errorLogger->log("Too many login attempts");

    break;

default:

    // Successful scenario, log in the user

    break;

}
```

## Problems:

Code semantics. It's quite difficult to understand what is happening there without reading the entire code. In order to know what does it mean to have a "-1" as a result of executing the checkLogin method, I have to checkout how I deal with it (the "Invalid credentials" message logged in the errorLogger), or I have to take a look at the checkLogin method implementation.

What about changing an error code value? I would have to review all the usages of the checkLogin method in order to change the received values!

## 2. Replace Magic Number with Symbolic Constant

Here you have another approach solving the 2 main problems described before:

```
private function checkLogin() {

    // ...

    // Some validation to check if the credentials are valid

    // ...

    if ($hasNotValidCredentials) {

        return self::INVALID_LOGIN_CREDENTIALS;

    }

    // ...

}
```

```
// Some validation to check if the user has attempted too many times to
login

// ...

if ($hasTooManyLoginAttempts) {

    return self::TOO_MANY_LOGIN_ATTEMPTS;

}

return self::LOGIN_SUCCESSFUL;
```

And, as in the previous step, here you can see one possible use of this method:

```
switch($this->checkLogin()) {

    case self::INVALID_LOGIN_CREDENTIALS:

        $this->errorLogger->log("Invalid credentials");

        break;

    case self::TOO_MANY_LOGIN_ATTEMPTS:

        $this->errorLogger->log("Too many login attempts");

        break;

    default:

        // Successful scenario, log in the user

        break;

}
```

### Improvements over the previous step:

We've applied the Replace Magic Number with Symbolic Constant refactor, which as we said before, improves the semantics of our code so that now we've a more readable code.

If we want to edit one error code value, we only have to edit the value of the constant, so it's more tolerant to changes.

### Problems:

The most important one arises if we want to make an Extract Method refactor inside the checkLogin method: We'll have to carry the error code from the checkLogin method until the switch-case statement which captures and deals with the error. Making all the intermediate methods conscience about this kind of behavior and having to return it. We probably need more flexibility in order to delegate in outer layers of our application the logic to deal with this exceptional situation.

### 3. Replace Error Code with Exception

Taking into account the problem described before, the only thing which can help in this case is to apply the Replace Error Code with Exception refactor, but for the sake of the learning process, we'll apply it in a weird way in order to see the benefits between the different intermediate approaches. We could end up with something like the following:

```
private function checkLogin() {
    // ...
    // Some validation to check if the credentials are valid
    // ...

    if ($hasNotValidCredentials) {
        throw new \RuntimeException("Invalid credentials",
self::INVALID_LOGIN_CREDENTIALS);
    }

    // ...
    // Some validation to check if the user has attempted too many times to
login
    // ...

    if ($hasTooManyLoginAttempts) {
        throw new \RuntimeException("Too many login attempts",
self::TOO_MANY_LOGIN_ATTEMPTS);
    }
}
```

Here you have how we would deal with the exception:

```
try {
    $this->checkLogin();
}
catch (\RuntimeException $loginException) {
    switch($loginException->getCode()) {
        case self::INVALID_LOGIN_CREDENTIALS:
            $this->errorLogger->log("Invalid credentials");
            break;
        case self::TOO_MANY_LOGIN_ATTEMPTS:
            $this->errorLogger->log("Too many login attempts");
            break;
    }
}

// Continue with the user login
```

#### Improvements over the previous step:

With this approach, **we're solving the problem of having the error code across our entire application**: from the error trigger point, to the error catching one. We can simply don't take into account this exception in one point of our application, making the outer points having to deal with it.

## Problems:

- We're introducing another kind of mistake here: we're catching all kind of RuntimeException that occurs below the execution flow of the checkLogin method. The problem is clear: What if the checkLogin method makes some DB queries in order to validate something and the DB server is down?

In this case, we'll throw a RuntimeException below the execution flow of the checkLogin method and we'll catch it at this point of the application, and as you can guess, we're not the ones who has the responsibility to deal with this other kind of exceptions.

This kind of scenarios already has a name: Pokemon Exception Handling (Pokemon – gotta catch 'em all).

- Additionally, here we're duplicating the error string. Something we could solve extracting it to a constant as we've done with the magic number, but since it is a specific situation in this point of the example, I prefer to focus on the previous point problem instead of distracting you with additional unnecessary code modifications.

### 3.1. Propagate other kinds of exceptions

We could make a workaround in the exception handling like this one:

```
try {
    $this->checkLogin();
}
catch (\RuntimeException $loginException) {
    switch($loginException->getCode()) {
        case self::INVALID_LOGIN_CREDENTIALS:
            $this->errorLogger->log("Invalid credentials");
            break;
        case self::TOO_MANY_LOGIN_ATTEMPTS:
            $this->errorLogger->log("Too many login attempts");
            break;
        default:
            throw $loginException;
            break;
    }
}

// Continue with the user login
```

## Improvements over the previous step:

We don't have to deal with exceptions with error codes different than the login related ones. **We're propagating these other kind of exceptions** to the outer layers of our application.

## Problems:

- What about **collisions in terms of error code numbers**? If another kind of error has the same error code than the login related ones, we'll catch it and deal with it as if it were a login related error.
- **Semantics**. We're letting go the opportunity to reflect in code our use case semantics. **We're using a generic *RuntimeException*** instead of taking profit of this opportunity and making an specific kind of Exception for our specific case.

## 4. Create specific *RuntimeException* child classes

So, finally, we could end up with **one specific exception for each exceptional situation**, something like:

```
class InvalidLoginCredentialsException extends \RuntimeException
{
}

private function checkLogin()
{
    // ...
    // Some validation to check if the credentials are valid
    // ...

    if ($hasNotValidCredentials) {
        throw new InvalidLoginCredentialsException();
    }

    // ...
    // Some validation to check if the user has attempted too many times to
login
    // ...

    if ($hasTooManyLoginAttempts) {
        throw new TooManyLoginAttemptsException();
    }
}
```

Using this approach, we could deal with the exceptions as you can see here:

```
try {
    $this->checkLogin();
}
catch (InvalidLoginCredentialsException $tooManyLoginAttemptsException) {
    $this->errorLogger->log("Invalid credentials");
}
catch (TooManyLoginAttemptsException $tooManyLoginAttemptsException) {
    $this->errorLogger->log("Too many login attempts");
}
```

## Improvements over the previous step:

We don't have possible collisions in terms of exception error codes, because we're "using the exception class as its identifier".

Also, as a collateral effect, our code has more semantics.

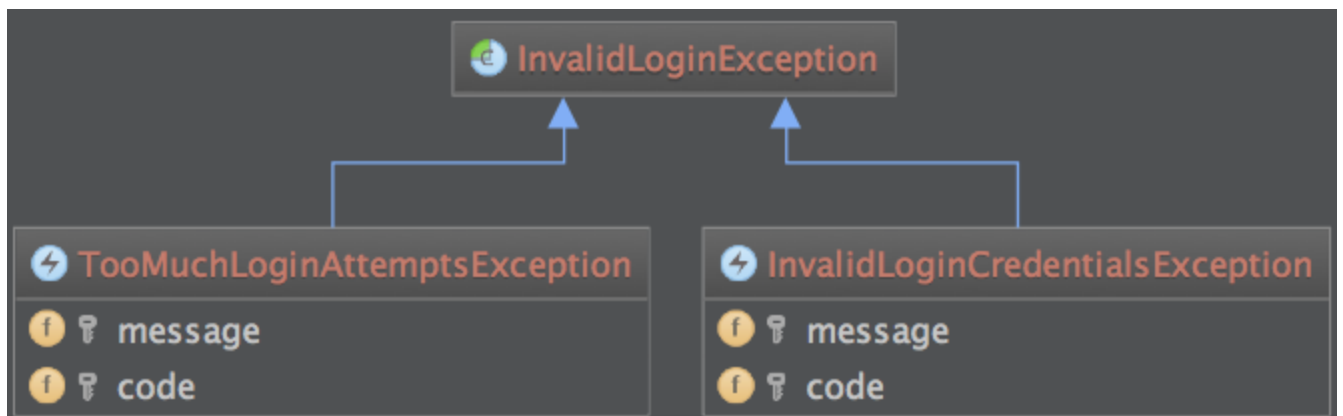
## Problems:

Even in this case, we're not dealing with the root of the problem: We're violating the Open/Closed Principle from the SOLID principles. We still have to modify "the hidden switch-case" that we've represented as different catch clauses. This kind of switch clauses are considered a Code Smell (even if they aren't a pure "switch" clause such as the serie of catch sentences or a serie of elseif). The explanation is simple:

- Now, we'll have to add another catch clause if we've to add another business rule that specify to raise a login error (banned users for instance).
- And if we've any other call to the checkLogin method in our application, we'll also have to edit those other try-catches.
- Furthermore, we're repeating code. In our specific case, we've to deal with the two errors in the same way. So we could abstract this kind of behavior.

## 5. Create an intermediate abstract Exception class

So finally, we can abstract this case thanks to an abstract exception class for the login use case exceptions, having the following class structure in terms of exceptions:



```

abstract class InvalidLoginException extends \RuntimeException
{
}

class InvalidLoginCredentialsException extends InvalidLoginException
{
    protected $message = 'Invalid credentials';
    protected $code = 2052;
}

class TooManyLoginAttemptsException extends InvalidLoginException
{

```

```
        protected $message = 'Too many login attempts';
        protected $code = 2051;
    }
```

We'll continue using them as we've done in the previous step:

```
private function checkLogin()
{
    // ...
    // Some validation to check if the credentials are valid
    // ...

    if ($hasNotValidCredentials) {
        throw new InvalidLoginCredentialsException();
    }

    // ...
    // Some validation to check if the user has attempted too many times to
login
    // ...

    if ($hasTooManyLoginAttempts) {
        throw new TooManyLoginAttemptsException();
    }
}
```

But in order to be able to capture both of them in the same catch, we'll capture the abstract one and make use of the *getCode* and *getMessage* *Exception* class methods:

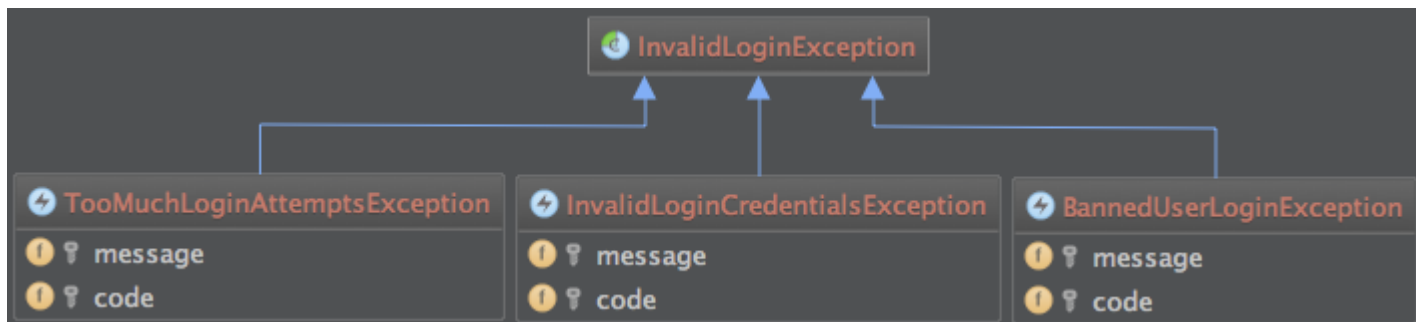
```
try {
    $this->checkLogin();
}
catch (InvalidLoginException $invalidLoginException) {
    $this->errorLogger->log('[Error ' . $invalidLoginException->getCode() .
    ']' . ' . $invalidLoginException->getMessage());
}
```

### Improvements over the previous step:

This change could be done thanks to the Replace Type Code with Subclasses and Replace Conditional with Polymorphism refactorings, and with this approach, if we implement another kind of error in the login process (banned user for instance), we'll only have to add another exception class without having to modify the point of dealing with this new error, because it's agnostic of the specific type of error. This makes our **code even more tolerant to changes and totally OCP compliant**. Here you have the example of adding this new case:

The new exception class would have to extend from the previously described *InvalidLoginException* abstract class in order to be able to catch it:





class BannedUserLoginException extends InvalidLoginException

```
{
    protected $message = 'Banned user tried to login';
    protected $code = 2053;
}
```

We can throw it as the previous ones:

```
private function checkLogin()
{
    // ...

    // Some validation to check if the credentials are valid
    // ...

    if ($hasNotValidCredentials) {
        throw new InvalidLoginCredentialsException();
    }

    // ...

    // Some validation to check if the user has attempted too many times to
    login

    // ...
}
```

```
        if ($hasTooManyLoginAttempts) {

            throw new TooManyLoginAttemptsException();

        }

        // ...

        // Some validation to check if the user has been banned in this moment

        // ...

        if ($hasBeenBanned) {

            throw new BannedUserLoginException();

        }

    }

}
```

And as you can imagine, there're no modifications to the try-catch block because of the previous exposed reason (extending from the *InvalidLoginException* class):

```
try {

    $this->checkLogin();

}

catch (InvalidLoginException $invalidLoginException) {

    $this->errorLogger->log('[Error ' . $invalidLoginException->getCode() .
    ']' . ' . $invalidLoginException->getMessage());

}
```

## Problems:

- Now, **we've too many logic inside the *checkLogin* method**. It's easy to suppose that it would have too many lines of code, making it more difficult to read.
- We'll not be able to reuse some of the validations.

### 5.1. Encapsulating the login validation

At this point, we can suppose that the business logic implemented in the *checkLogin* function is too long, so we can make use of the Extract Method refactoring in order to leave a more readable and reusable code:

```
private function checkLogin()

{
```

```
$this->checkLoginCredentials();  
  
$this->checkLoginAttempts();  
  
$this->checkBannedUser();  
  
}
```

The outer logic will remain as:

```
try {  
  
    $this->checkLogin();  
  
}  
  
catch (InvalidLoginException $invalidLoginException) {  
  
    $this->errorLogger->log('[Error ' . $invalidLoginException->getCode() .  
    ']' . $invalidLoginException->getMessage());  
  
}
```

## Conclusions

Once we've arrived to this point, we could have a sensation of "good enough" with our code, but taking a closer look at the final checkLogin method, we can see some kind of generic behavior there. I mean, we could make a 6th iteration with our solution in order to implement an observer design patter compositing the userValidator service with all the validators. But it could leave our class structure a little bit over-engineered and it's something out of this post scope

By the way, our design has a very little flaw: What if someone extends the InvalidLoginException exception not defining a specific code and message?

We could ensure that the login error code and message are present applying the template method design pattern with an abstract query method defined in the InvalidLoginException class. But for this specific case, I don't find it necessary despite being a robuster solution. As you can imagine, it depends on your case and context.

~~~ End of Article ~~~



# Ubuntu Tips: How Do you Display PHP Error Messages?

**Source** <http://matthewwittering.com/blog/ubuntu-tips/php-error-messages.html>

Like me you maybe using an Ubuntu computer running Apache, MySQL and PHP to develop websites. When developing it is important to see error messages to debug your code. On occasion I have installed PHP on to Ubuntu computers and by default errors are not displayed. As this is not a production web server I updated the php.ini file to display errors. In this post I explain how I updated the php.ini to display error messages and aid debugging.

## Open php.ini

Start by opening a new terminal window to open the php.ini file. Modifying the php.ini will allow you to tunes the setting to enable more descriptive error messages, logging, and better performance.

Enter the following command to begin editing php.ini. This will require the sudo command and therefore administrator privileges.

```
sudo nano /etc/php5/apache2/php.ini
```

## Display Errors

Now scroll down through the file until you find the following line.

```
display_errors = Off
```

Once you have found the display\_errors line replace the parameter 'Off' with 'On'. Once you have made the swap save the file and then exit the editor.

```
display_errors = On
```

## Restart Apache

Now that you have made the change to the php.ini file you need to restart the Apache web server to effect the change. To do this enter the command below into your terminal window to restart the service. Once the service has restarted you will be able to see error message in your PHP scripts instead of blank white pages.

```
sudo /etc/init.d/apache2 restart
```

~~~ End of Article ~~~

