**OOP(Object Oriented Programming)**

* Advance php features like design patterns,
  + OOPS,
  + Namespace,
  + Traits
* Object Oriented Programming
  + It is the core of any framework and if trainee is clear with OOPs they can easily jump or switch to any new framework and technology.
* Refence links
  + <https://www.youtube.com/watch?v=7GWwjn0bs5g&list=PLJ49JaW9k5klgfZ22BlgZiElIz3cz08ta&index=7>
* The fundamentals of OOPs
  + Traits
    - Traits is a special class introduce in php7 to overcome limitation of multiple inheritance
    - https://www.php.net/manual/en/language.oop5.traits.php
  + Namespace
    - Namespace is useful to understand the directory structure of any framework
    - https://www.sitepoint.com/php-53-namespaces-basics/
  + Standards
    - Standards will teach the trainee how to write the code?
    - <https://www.php-g.org/psr/>
    - Basic Coding Standard
    - Logger Interface
    - Autoloading Standard,
* **Composer**
  + Download composer.phar file and install one php library using php composer.phar
  + <https://getcomposer.org/>
  + Autoload that installed php libraray by doing practial.
  + Autoload understanding : <https://www.brainbell.com/php/auto-loading.html>
* **PHP design patterns**
  + Design patterns are necessary to understand core of most of the frameworks have code written using various design patterns. To understand the basics of design patter go through following links
  + <https://phptherightway.com/pages/Design-Patterns.html>
* **Reference**:
  + PHP design patterns
    - https://www.ibm.com/developerworks/library/os-php-designptrns <http://www.phptherightway.com/pages/DesignPatterns>.
    - html OOPS Basics
    - <https://www.tutorialspoint.com/php/php_object_oriented.htm>
    - Basics of Namespaces
    - <https://www.sitepoint.com/php-53-namespaces-basics/>
* **Type Hint**
  + With Type hinting we can specify the expected data type (arrays, objects, interface, etc.) for an argument in a function declaration. This practice can be most advantageous because it results in better code organization and improved error messages.
  + When we would like to force a function to get only arguments of the type array, we can put the keyword array in front of the argument name,
    - function functionName (array $argumentName)
    - {
    - //code
    - }
  + Type hinting can also be used to force a function to get an argument of type Object
    - class Car {
    - protected $driver;
    - // The constructor can only get Driver objects as arguments.
    - public function \_\_construct(Driver $driver)
    - {
    - $this -> driver = $driver;
    - }
    - }
    - class Driver {}

    - $driver1 = new Driver();
    - $car1 = new Car($driver1);
* **Namespace**
  + Namespace is useful to understand the directory structure of any framework
  + As the size of your PHP code library increases, the more likely you will accidentally reuse a function or class name that has been declared before
  + Namespaced code is defined using a single namespace keyword at the top of your PHP file
  + it must be the first command (with the exception of declare) and no non-PHP code or white-space can precede the command
  + <?php
  + // define this code in the MyProject namespace
  + namespace MyProject;
  + It is not possible to nest namespaces or define two or more namespaces for the same code block
  + However, you can define different namespaced code in the same file, e.g.
  + <?php
  + namespace MyProject1;
  + // PHP code for the MyProject1 namespace
  + namespace MyProject2;
  + **Sub-namespaces**
    - PHP allows you to define a hierarchy of namespace names so libraries can be sub-divided. Sub-namespaces are separated using a backslash (\) character
    - MyProject\SubName
    - MyProject\Database\MySQL
    - Example
      * In a file named lib1.php, we will define a constant, function, and class with the Html namespace:
      * namespace Html;
      * class Table {
      * public $title = "";
      * public $numRows = 0;
      * public function message() {
      * echo "<p>Table '{$this->title}' has {$this->numRows} rows.</p>";
      * }
      * }
      * $table = new Table();
      * $table->title = "My table";
      * $table->numRows = 5;
  + **Importing, Aliases, and Name Resolution**
    - Namespace Importing
      * Namespaces can be imported with the use operator,
      * use App\Lib1;
      * echo \App\Lib1\MYCONST . "\n";
      * echo \App\Lib1\MyFunction() . "\n";
      * echo \App\Lib1\MyClass::WhoAmI() . "\n";
    - Namespace Aliases
      * Namespace aliases are perhaps the most useful construct. Aliases allow us to reference long namespaces using a shorter name..
      * use App\Lib1 as L;
      * use App\Lib2\MyClass as Obj;
      * The first use statement defines AppLib1 as ‘L’. Any qualified names using ‘L’ will be translated to ‘AppLib1’ at compile-time
    - PHP Name Resolution Rules
      * PHP identifier names are resolved using the following namespace rules
      * Calls to fully-qualified functions, classes or constants are resolved at compile-time.
      * Unqualified and qualified names are translated according to the import rules
      * inside a namespace, all qualified names not already translated according to import rules have the current namespace prepended
      * Unqualified class names are translated according to current import rules and the full name is substituted for short imported name
      * Unqualified function calls within a namespace are resolved at run-time
      * Calls to unqualified or qualified class names are resolved at run-time.
  + **The namespace Keyword**
    - The namespace keyword can be used to explicitly reference an item within the current namespace or a sub-namespace. It is the namespace equivalent of self within classes:
    - Autoloading Namespaced Classes
      * One of the best time-saving features of PHP 5 is autoloading.
      * function \_\_autoload($class\_name) {
      * require\_once("classes/$class\_name.php");
      * }
      * the fully-qualified namespace and class name is passed to the \_\_autoload function
* **Traits**
  + Traits is a special class introduce in php7 to overcome limitation of multiple inheritance. It is widely used by us, please study it carefully
  + PHP implements a way to reuse code called Traits.
  + Traits are a mechanism for code reuse in single inheritance languages such as PHP. A Trait is intended to reduce some limitations of single inheritance by enabling a developer to reuse sets of methods freely in several independent classes living in different class hierarchies
  + The semantics of the combination of Traits and classes is defined in a way which reduces complexity, and avoids the typical problems associated with multiple inheritance and Mixins
  + A Trait is similar to a class,
    - trait ezcReflectionReturnInfo {
    - function getReturnType() { /\*1\*/ }
    - function getReturnDescription() { /\*2\*/ }
    - }
    - class ezcReflectionMethod extends ReflectionMethod {
    - **use ezcReflectionReturnInfo;**
    - /\* ... \*/
    - }
  + Traits are declared with the trait keyword:
  + trait message1 {
  + public function msg1() {
  + echo "OOP is fun! ";
  + }
  + }
  + Sometimes you may want to inherit from more than one class. Since this is not possible in PHP and other single inheritance languages like it, we have something called traits
  + Multiple Traits
    - Multiple Traits can be inserted into a class by listing them in the use statement, separated by commas.
    - use Hello, World;
    - as classes can make use of traits, so can other traits. By using one or more traits in a trait definition, it can be composed partially or entirely of the members defined in those other traits.
    - trait Hello {
    - public function sayHello() {
    - echo 'Hello ';
    - }
    - }
    - trait World {
    - public function sayWorld() {
    - echo 'World!';
    - }
    - }
    - trait HelloWorld {
    - use Hello, World;
    - }
    - class MyHelloWorld {
    - use HelloWorld;
    - }
  + Abstract Trait Members
    - Traits support the use of abstract methods in order to impose requirements upon the exhibiting class. Public, protected, and private methods are supported
    - trait Hello {
    - public function sayHelloWorld() {
    - echo 'Hello'.$this->getWorld();
    - }
    - abstract public function getWorld();
    - }
  + Static Trait members
    - Traits can define static variables, static methods and static properties.
    - trait StaticExample {
    - public static function doSomething() {
    - return 'Doing something';
    - }
    - }m
    - Example::doSomething();
  + **Constants**
    - Traits can, as of PHP 8.2.0, also define constants.
    - trait ConstantsTrait {
    - public const FLAG\_MUTABLE = 1;
    - final public const FLAG\_IMMUTABLE = 5;
    - }
* **Standards**
  + Standards will teach the trainee how to write the code?
    - Basic Coding Standard
      * Files MUST use only <?php and <?= tags.
      * Files MUST use only UTF-8 without BOM for PHP code
      * Files SHOULD either declare symbols (classes, functions, constants, etc.) or cause side-effects (e.g. generate output, change
      * Namespaces and classes MUST follow an "autoloading" PSR
      * Class names MUST be declared in StudlyCaps
      * Class constants MUST be declared in all upper case with underscore separators.
      * Method names MUST be declared in camelCase
      * PHP Tags
        + PHP code MUST use the long <?php ?> tags or the short-echo <?= ?> tags; it MUST NOT use the other tag variations.
      * Character Encoding
        + PHP code MUST use only UTF-8 without BOM.
      * Side Effects
        + The phrase "side effects" means execution of logic not directly related to declaring classes, functions, constants, etc., merely from including the file
        + Side effects" include but are not limited to: generating output, explicit use of require or include
        + ini\_set('error\_reporting', E\_ALL);
      * Namespace and Class Names
        + Namespaces and classes MUST follow an "autoloading" PSR:
        + This means each class is in a file by itself, and is in a namespace of at least one level: a top-level vendor name.
        + Class names MUST be declared in StudlyCap
      * Class Constants, Properties, and Methods
        + The term "class" refers to all classes, interfaces, and traits.
      * Constants
        + Class constants MUST be declared in all upper case with underscore separators
      * Properties
        + This guide intentionally avoids any recommendation regarding the use of $StudlyCaps, $camelCase, or $under\_score property names.
        + Whatever naming convention is used SHOULD be applied consistently within a reasonable scope. That scope may be vendor-level, package-level, class-level, or method-level.
      * Methods
        + Method names also declare with realted to functionality
        + Method names MUST be declared in camelCase().
    - Logger Interface
      * This document describes a common interface for logging libraries.
      * The main goal is to allow libraries to receive a Psr\Log\LoggerInterface object and write logs to it in a simple and universal way. Frameworks and CMSs that have custom needs MAY extend the interface for their own purpose, but SHOULD remain compatible with this document.
      * The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described
      * The word implementor in this document is to be interpreted as someone implementing the LoggerInterface in a log-related library or framework.
      * Every method accepts a string as the message, or an object with a \_\_toString() method. Implementors MAY have special handling for the passed objects. If that is not the case, implementors MUST cast it to a string
      * Placeholder names MUST correspond to keys in the context array.
      * Placeholder names MUST be delimited with a single opening brace { and a single closing brace }. There MUST NOT be any whitespace between the delimiters and the placeholder name.
      * Context
        + Every method accepts an array as context data. This is meant to hold any extraneous information that does not fit well in a string. The array can contain anything. Implementors MUST ensure they treat context data with as much lenience as possible. A given value in the context MUST NOT throw an exception nor raise any php error, warning or notice
        + Implementors MUST still verify that the 'exception' key is actually an Exception before using it as such, as it MAY contain anything.
      * Helper classes and interfaces
        + The Psr\Log\AbstractLogger class lets you implement the LoggerInterface very easily by extending it and implementing the generic log method.
        + Similarly, using the Psr\Log\LoggerTrait only requires you to implement the generic log method.
        + The Psr\Log\NullLogger is provided together with the interface. It MAY be used by users of the interface to provide a fall-back "black hole" implementation if no logger is given to them.
        + The Psr\Log\LoggerAwareInterface only contains a setLogger(LoggerInterface $logger) method and can be used by frameworks to auto-wire arbitrary instances with a logger.
        + The Psr\Log\LoggerAwareTrait trait can be used to implement the equivalent interface easily in any class. It gives you access to $this->logger
      * Psr\Log\LoggerInterface
        + public function emergency($message, array $context = array());
        + public function alert($message, array $context = array());
      * Psr\Log\LoggerAwareInterface
        + setLogger
      * Psr\Log\LogLevel
      * + class LogLevel
        + {
        + const EMERGENCY = 'emergency';
        + const ALERT = 'alert';
        + const CRITICAL = 'critical';
        + const ERROR = 'error';
        + const WARNING = 'warning';
        + const NOTICE = 'notice';
        + const INFO = 'info';
        + const DEBUG = 'debug';
        + }
    - Autoloading Standard
      * The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described,
      * This PSR describes a specification for autoloading classes from file paths. It is fully interoperable, and can be used in addition to any other autoloading specification, including PSR-0.
      * The term "class" refers to classes, interfaces, traits, and other similar structures.
      * A fully qualified class name has the following form:
        + \<NamespaceName>(\<SubNamespaceNames>)\*\<ClassName>
        + The fully qualified class name MUST have a top-level namespace name, also known as a "vendor namespace".
        + The fully qualified class name MAY have one or more sub-namespace names.
        + The fully qualified class name MUST have a terminating class name.
        + Underscores have no special meaning in any portion of the fully qualified class name.
        + Alphabetic characters in the fully qualified class name MAY be any combination of lower case and upper case.
        + All class names MUST be referenced in a case-sensitive fashion.
      * When loading a file that corresponds to a fully qualified class name
        + A contiguous series of one or more leading namespace and sub-namespace names, not including the leading namespace separator, in the fully qualified class name (a "namespace prefix") corresponds to at least one "base directory".
        + The contiguous sub-namespace names after the "namespace prefix" correspond to a subdirectory within a "base directory", in which the namespace separators represent directory separators. The subdirectory name MUST match the case of the sub-namespace names.
        + The terminating class name corresponds to a file name ending in .php. The file name MUST match the case of the terminating class name.
      * Autoloader implementations MUST NOT throw exceptions, MUST NOT raise errors of any level, and SHOULD NOT return a value.
      * The spl\_autoload\_register() function registers any number of autoloaders, enabling for classes and interfaces to be automatically loaded if they are currently not defined.
      * Any class-like construct may be autoloaded the same way. That includes classes, interfaces, traits, and enumerations
      * spl\_autoload\_register(function ($class\_name) {
      * include $class\_name . '.php';
      * });
      * Autoloading is the process of automatically loading PHP classes without explicitly loading them with the require(), require\_once(), include(), or include\_once() functions.
      * Each class must be defined in a separate file
      * Name your class files the same as your classes
      * An autoloader is a function that takes a class name as an argument and then includes the file that contains the corresponding class
      * As of PHP 7.2.0 the \_\_autoload() function has been deprecated and removed since PHP 8.0.0. Now it is recommended to use the spl\_autoload\_register for that purpose instead.
      * You can see all classes loaded without using the include or require keywords. Since PHP 5.3, you can use spl\_autoload\_register() with namespaces, which means that you can organize your project and autoload your PHP classes without the require or include keyword.
* **Composer**
  + Download composer.phar file and install one php library using php composer.phar
  + Autoload that installed php libraray by doing practial.
  + Autoload understanding
    - <https://www.brainbell.com/php/auto-loading.html>
  + **Basic usage**
    - composer.json: Project setup
      * To start using Composer in your project, all you need is a composer.json file
      * This file describes the dependencies of your project and may contain other metadata as well.
      * It typically should go in the top-most directory of your project/VCS repository.
      * **The require key**
        + The first thing you specify in composer.json is the require key. You are telling Composer which packages your project depends on.
        + Example  
          { "require": {

"monolog/monolog": "2.0.\*"

}

}

* + - * + As you can see, require takes an object that maps package names (e.g. monolog/monolog) to version constraints (e.g. 1.0.\*).
      * **Package names**
        + The package name consists of a vendor name and the project's name
        + Often these will be identical - the vendor name only exists to prevent naming clashes
        + For example, it would allow two different people to create a library named json. One might be named igorw/json while the other might be seldaek/json.
      * **Package version constraints**
        + version constraint 2.0.\*
        + This means any version in the 2.0 development branch, or any version that is greater than or equal to 2.0 and less than 2.1 (>=2.0 <2.1)
      * **Installing dependencies**
        + To initially install the defined dependencies for your project, you should run the update command
        + php composer.phar update
        + resolves all dependencies listed in your composer.json file and writes all of the packages and their exact versions to the composer.lock file, locking the project to those specific versions.
        + You should commit the composer.lock file to your project repo so that all people working on the project are locked to the same versions of dependencies (more below).
        + It then implicitly runs the install command. This will download the dependencies' files into the vendor directory in your project.
      * **Commit your composer.lock file to version control**
        + Committing this file to version control is important because it will cause anyone who sets up the project to use the exact same versions of the dependencies that you are using**.**
      * **Installing from composer.lock**
        + If there is already a composer.lock file in the project folder, it means either you ran the update command before, or someone else on the project ran the update command and committed the composer.lock file to the project
        + Either way, running install when a composer.lock file is present resolves and installs all dependencies that you listed in composer.json, but Composer uses the exact versions listed in composer.lock to ensure that the package versions are consistent for everyone working on your project
        + after fetching new changes from your VCS repository it is recommended to run a Composer install to make sure the vendor directory is up in sync with your composer.lock file.
        + **php composer.phar install**
      * **Updating dependencies to their latest versions**
        + As mentioned above, the composer.lock file prevents you from automatically getting the latest versions of your dependencies. To update to the latest versions, use the update command
        + php composer.phar update monolog/monolog [...]
      * **Packagist**
        + Packagist.org is the main Composer repository
        + A Composer repository is basically a package source: a place where you can get packages from.
        + Packagist aims to be the central repository that everybody uses
        + If you go to the Packagist.org website, you can browse and search for packages
        + Any open source project using Composer is recommended to publish their packages on Packagist. A library does not need to be on Packagist to be used by Composer, but it enables discovery and adoption by other developers more quickly
      * **Platform packages**
        + Composer has platform packages,which are virtual packages for things that are installed on the system but are not actually installable by Composer.
        + hhvm represents the version of the HHVM runtime and allows you to apply a constraint, e.g., ^2.3.
        + ext-<name> allows you to require PHP extensions (includes core extensions).
        + lib-<name> allows constraints to be made on versions of libraries used by PHP.
      * **Autoloading**
        + For libraries that specify autoload information, Composer generates a vendor/autoload.php file. You can include this file and start using the classes that those libraries provide without any extra work
        + require \_\_DIR\_\_ . '/vendor/autoload.php';
        + $log = new Monolog\Logger('name');
        + You can even add your own code to the autoloader by adding an autoload field to composer.json.
        + Example

{

"autoload": {

"psr-4": {"Acme\\": "src/"}

}

}

* + - * + Composer will register a PSR-4 autoloader for the Acme namespace.
        + After adding the autoload field, you have to re-run this command:

php composer.phar dump-autoload

* + - * + This command will re-generate the vendor/autoload.php file. See the dump-autoload section for more information.