

PHP II

Courseware by Rogue Wave Technical Training

Introduction

This module covers:

- Introductions
- Course details
- Web conferencing tool
- Course virtual machine

Introductions

Introductions

Student and instructor introductions. Please offer a little information about yourself. Here are some ideas:

- Your location
- The time of day where you are located
- Something about yourself unknown to the everyone

Course Details

Course Details

In this course, you will learn about intermediate PHP concepts, OOP, Web services, database, etc. By the time you are finished, you will have a working knowledge of:

- OOP concepts of PHP
- Database connection and adapters
- Data formats and web services
- Fundamental knowledge of regular expressions
- Output control
- System configuration

Course Approach

This class builds upon your knowledge and presents concepts at an intermediate level, it:

- Includes a combination of lecture and hands on labs.
- Lots of interaction between all participants and instructor.
- Includes a number of labs some of which accomplished as homework.
- Focuses on using PHP for web development based on the object model.
- Does not provide deep instruction on certain important topics, like security or application architecture, which are available in separate courses.
- Provides a virtual environment for completing course assignments.
- Provides one or more course projects.
- Provides access to the course instructor.
- Provides access to Zend resources.
- Prepares the student to move to the next level.

Prerequisite Understanding

Familiarity with the following concepts is necessary:

- Quotes and comments
- Data types (strings, integers, floats, booleans, arrays, resources, and null)
- Type casting and juggling
- Operators and order of precedence
- Constant and variable identifiers
- Arrays
- Conditional and looping constructs
- Functions
- Concept of encapsulation
- Scoping rules
- PHP/HTML integration

Course Exercises

The exercises within the course are designed to reinforce key concepts and provide you with relevant practice in applying your new skills.

Many of the exercises are designed to build understanding of how

Exercises are assigned by the instructor and accomplished on your own time. Review of exercises is done as time permits.

Some examples include:

- Creating classes and objects.
- Working with the database access object PDO.
- Building a form class and use it to build forms.
- Call a web service and process the returned data.
- Validate form input with regular expression functions.
- Building code to reduce server load.

End Objective

The content of this course will enable you to:

- Create database-driven web applications similar to the course applications.
- Leverage object-oriented programming (OOP) techniques in your applications.
- Use built-in objects to interface a database.
- Analyze input data and learn to filter and validate it, and why.
- Request a web service and process the return data.
- Learn how to better recognize inefficient coding practices, and improve them.
- Learn best practices.
- Learn a few concepts called software design patterns.
- How to throw and handle exception objects.

Web Conferencing Tool

Web Conferencing Tool

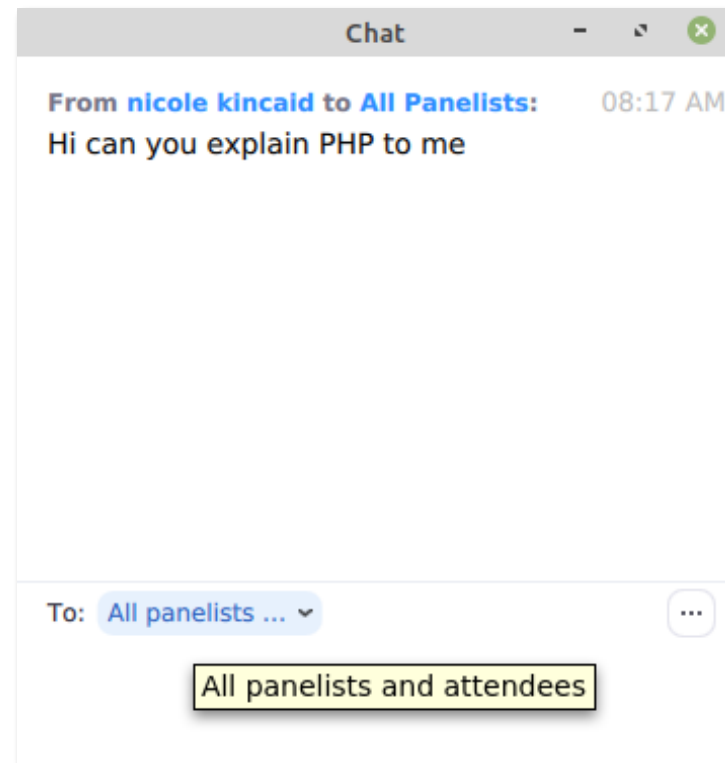
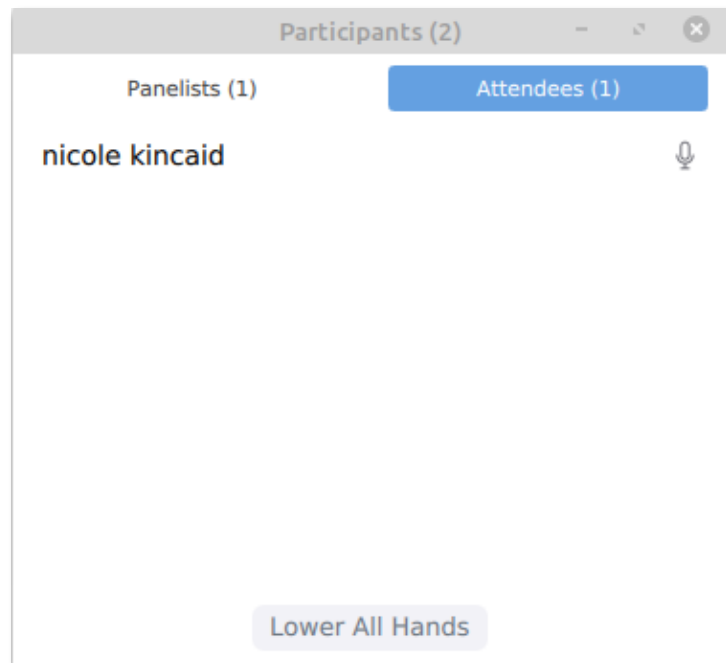
The course uses Zoom web conferencing and includes participant and chat panels.

Microphones are muted on entry to reduce ambient noise.



Panels

Examples of the participant and chat panels



Course Virtual Machine

Course Virtual Machine

The course uses a virtual machine (VM). A virtual machine is a separate operating system running inside a window of your main computer.

The VM installs software necessary to complete the course requirements, including:

- An operating system
- A web server
- Database servers
- The PHP language interpreter
- Gedit, the text editor
- Course applications
- Other software as required

Course Applications/Projects

Applications/Projects included:

- OrderApp
- Sandbox

Module Summary

This module included:

- Introductions
- Course Details
- Web Conferencing Tool
- Course Virtual Machine

Object Oriented PHP

This module covers:

- PHP namespaces
- Classes and class members
- Objects
- Inheritance
- Anonymous classes
- Overriding
- Magic methods
- Abstract classes and methods
- Interfaces
- Type hinting
- Exceptions
- Static properties and methods
- Polymorphism
- Traits
- Object cloning

PHP Namespaces

PHP Namespaces

A [namespace](#) is an encapsulating mechanism for classes, constants and functions. These constructs are susceptible to naming collisions with other code.

Encapsulation within a namespace solves naming collisions, and allows use of third party code without concern of naming collisions.

A namespace is declared at the beginning of a file with the *namespace* keyword, and given a name, sometimes including backslashes.

Namespaces are imported into a file with the *use* keyword. This functionality replaces the require/include imports.

Once code is namespaced, qualifying calls to other namespaces is required.

Namespaces may be aliased using the *as* keyword. This can shorten qualifiers.

Example

/some/path/Application/Controller/IndexController.php.

```
namespace Application\Controller;
class IndexController {
    public function indexAction() {
        // code (not shown)
    }
}
```

/some/path/Application/Form/LoginForm.php.

```
namespace Application\Form;
class LoginForm {
    public $username;

    public function setUsername(string $username) {
        $this->username = $username;
    }
}
```

Considerations

Notes:

- Namespaces can encapsulate with curly braces.
- Namespaces sometimes map directly to a directory structure thereby conforming to the [PSR-4](#) autoloading standard.
- Namespaces are highly recommended as they eliminate naming collisions with other code.
- The *namespace* keyword is resolvable to the current namespace name as is the magic constant `__NAMESPACE__`.
- The *global* namespace is qualified with a leading backslash (`\`).
- The number of namespaces used is unlimited.

Best Practice: Use one namespace per code file.

Namespaces and Autoloading

Autoloading is built into the PHP language. When you reference a class whose definition has not been included, PHP attempts to automatically load the class into memory.

To implement one or more auto loaders use [spl_autoload_register\(\)](#). Using this function, any or all of the following can be added as autoloaders:

- Anonymous functions
- Procedural functions
- Class methods (i.e. functions inside classes)
- Third party autoloaders (e.g. the Composer autoloader)

[PSR-4](#) defines the relationship between namespaces and autoloading.

In earlier versions of PHP, a global function [__autoload\(\)](#) was used. It is deprecated and removed in later versions of PHP in favor of `spl_autoload_register()`.

Composer Autoloading

Composer (a PHP package manager discussed later) includes a commonly used autoloader on installation.

When available, loading the Composer auto loader is required, then from that point, and assuming some configuration, Composer's auto loader will load all namespaces into memory. Here's an example at the top of an *index.php* file:

```
require 'vendor/autoload.php';
```

Then, register the autoload directories in the composer.json file:

```
//...  
"autoload": {  
    "psr-4": {  
        "App\\": "src/App/",  
        ...  
    }  
},  
//...
```

Once the auto loader is loaded, and directories specified as above, executing the following shell script at the project root will configure the auto loader using either global or local Composer installation:

```
vagrant@php-training:~$ <composer/php composer.phar> dump-autoload
```


Namespace Labs

Lab: Namespace

Have a look at the *OrderApp* in the course VM.

1. Identify the namespaces used
2. How is autoloading initiated?

Lab complete.

Classes and Class Members

The Class Construct

A [class](#) construct contains constants (as and if required), properties, and methods.

An example representing a user entity. Note the *class* keyword, the *class name*, and encapsulating curly braces.

```
class UserEntity {  
    public const TABLE = 'user';  
    protected $firstName, $lastName;  
    public function setFirstName(string $firstName){  
        $this->firstName = $firstName;  
    }  
    public function setLastName(string $lastName){  
        $this->lastName = $lastName;  
    }  
    public function getFirstName () {  
        return $this->firstname;  
    }  
    public function getLastName () {  
        return $this->lastname;  
    }  
    public function getFullName () {  
        return trim(($this->firstname . ' ' ?? ")  
            . ($this->lastname ?? "));  
    }  
}
```

Class Constants

A [class constant](#) is an unchanging (immutable) class member value, defined within the class construct with the `const` keyword. The below constant `TABLE` refers to an SQL database table.

```
class User {  
    // Constant *****  
    public const TABLE = 'user';  
    // *****  
  
    protected $firstName, $lastName;  
    public function setFirstName(string $firstName){  
        $this->firstName = $firstName;  
    }  
  
    // ...  
}
```

Class Constants

Usage

Class constant notables:

- They provide an immutable value to the enclosing class for inner or outer reference.
- They can have visibilities assigned just like properties.
- They can have pre-assigned scalar or array values.
- Their availability is determined by visibility.

Best Practice: Define the name using all upper case letters.

Class Property

A class property is a mutable (changeable) value that represents something about the class.

Called a *property* to distinguish from other application variables, and as a member of a class.

Unless otherwise specified, access level is *public* by default.

Can be referenced using the pseudo variable *\$this*.

Class properties can hold any value:

- A scalar
- null
- Another object
- An array
- A resource

Class Property

The [class property](#) is a changeable (mutable) class member identifier and value. Think of them as class member variables that represent a value related to a class concept, scope, or context. Example of class properties:

```
class UserEntity {
    public const TABLE = 'user';

    // Properties *****
    protected $firstName, $lastName; // <-- Properties, both with a "protected" visibility
    // *****

    public function getFirstName() {
        return $this->firstname;
    }

    // ...
}
```


Class Property

Best practices:

- Define property name beginning with a lower case letter or underscore.
- Be as descriptive with naming as possible to identify the type of value by quick glance.
- Define properties only as required, and used by, the application.
- Group same visibilities on a single line to reduce redundant declaration.
- Pre-assign values as needed.

Class Method

A class method is a local code encapsulation just like a function, which resides within the class as a member of the class. They are class member functions, but called *methods* as a distinction. Example of class methods in the *UserEntity* class:

```
class UserEntity {  
    protected $firstName, $lastName;  
  
    // METHODS *****  
    public function getFirstName() {  
        return $this->firstName;  
    }  
  
    public function getLastName() {  
        return $this->lastName;  
    }  
    // *****  
}
```

Method Method

Class methods:

- Are a re-usable block of code
- Called *method* to distinguish from other application functions, and as a member of a class
- Unless otherwise specified, are *public* visible by default
- Are referenced using the pseudo variable *\$this*
- Can accept arguments just like a regular function

Best Practices:

- Define name beginning with a lower case letter or underscore, followed by camel casing.
- Use verbs with descriptive naming.

Class Member Visibility

[Visibility](#) is a setting that specifies availability of a class constant, property or method to other code. Think of a visibility as a mechanism to control access.

Three visibilities are available:

- **Public:** Allows access from any scope and does not require accessor methods.
- **Protected:** Allows direct access from defined and derived class scopes only. Is inheritable.
- **Private:** Allows access only from defined class scope. Not inheritable.

\$this

The *\$this* special variable is assigned to the current object when the parser executes within the object's class scope.

\$this is only used within the context of objects.

Objects

Objects

An [object](#) is an instance of a class, and contains all defined class members.

They can have unique properties and methods.

They represent a unique instance of a class, like an individual user, employee, invoice, etc.

Example of instantiating a user entity. The *new* keyword is required:

```
$userEntity = new UserEntity();  
$userEntity->setLastname('Watney');  
$userEntity->setFirstname('Mark');  
echo $userEntity->getFullName(); // outputs: "Some Mark Watney"
```

Unique Instances

Objects can and usually do have unique values set at the time of instantiation. The previous examples just instantiated objects that were identical.

The object constructor method, called `__construct()`, is magic, which means it is called automatically when using the `new` keyword, and if defined in the class. Here's an example:

```
class UserEntity {  
    protected $firstName, $lastName;  
    public function __construct(string $firstName , string $lastName) {  
        $this->firstname = $firstName ;  
        $this->lastname = $lastName ;  
    }  
}
```

// Now for the instances

```
$user1 = new UserEntity('Jack' , 'Ryan');  
$user2 = new UserEntity('Monte' , 'Python');  
$user3 = new UserEntity('James' , 'Bond');
```


Considerations

Object constructors:

- Are not required
- If used, should set a hard dependency. A hard dependency is a value absolutely required on every instantiation.
- Are automatically triggered by the *new* keyword
- Constructs unique instances
- Are preceded with two underscores denoting *a magic method*

Best Practices:

- Use constructors when unique values are required.
- Use a type hint in the constructor signature for each parameter.
- Use to setup dependent properties.

Class Functions

PHP has several [functions](#) tailored for working with classes. Here are some of the more important ones:

get_class(): Return a class name.

get_class_methods(): Returns an array of a classes methods.

class_exists(): Returns boolean on check of a loaded class.

get_parent_class(): Returns the name of the parent class.

get_class_vars(): Returns an array of the classes default properties.

method_exists(): Returns boolean on check of a method existence.

property_exists(): Returns boolean on check of a property existence.

Class Labs

Lab: Create a Class

Complete the following:

1. Create a class definition that represents or models something. Give it a constant, some properties, and a few methods. Set appropriate visibilities for each.
2. Instantiate a couple of objects, and execute the methods created producing some output.
3. Create something which is realistic and appropriate to a current or future application for your domain.

Lab complete.

Inheritance

Inheritance

[Object inheritance](#) is one of the cornerstones of object oriented programming is the ability for classes to inherit certain properties and methods from another class. The inheriting classes are referred to as *derived*, *child*, or *sub* classes. The inherited class is referred to as the *base*, *parent* or *super* class.

Inheritance involves a class declaration to *extends* another class, thereby inheriting the inheritable properties and methods. Inheritable visibilities are *public* and *protected*.

```
class GuestUserEntity extends UserEntity {  
    public $role;  
  
    public function getRole() {  
        return $this->role;  
    }  
}
```

A Superclass

A superclass example:

```
class UserEntity {  
    public const TABLE = 'user';  
    protected $firstName, $lastName;  
    //...  
}
```

A subclass example:

```
class GuestUser extends User {  
    public $role;  
  
    public function getRole() {  
        return $this->role;  
    }  
}
```

A Superclass

Superclass Method Availability

A super class method becomes available from a subclass method by use of the *parent::<method name>* operator. Here's an example:

```
// The superclass
class UserEntity {
    protected $firstName, $lastName;
    public function __construct(string $firstName, string $lastName) {
        $this->firstName = $firstName;
        $this->lastName = $lastName;
    }
    // ...
}

// The subclass
class GuestUser extends UserEntity {
    public $role;
    public function __construct(string $firstName, string $lastName, string $role) {
        parent::__construct($firstName, $lastName);
        $this->role = $role ;
    }
    // ...
}
```


Class Inheritance Considerations

Notes:

- Visibilities allow, limit or prevent inheritance.
- It is necessary to either explicitly load, or autoload a super class before extending.
- PHP does not support multiple inheritance with classes. A subclass cannot extend from more than one superclass.

Best Practices:

- Design inheritance models moving from general constants, properties, and methods in a superclasses, to increasingly more specific properties and methods in subclasses.
- Keep classes limited in scope or aspect. Build multiple limited aspect classes if necessary.
- Maintain extensibility as much as possible.
- If a superclass methods exists, use it, otherwise remove it.

Inheritance Labs

Lab: Create an Extensible Super Class

Complete the following:

1. Using the code created in the previous exercise, create an extensible superclass definition. Set the properties and methods that subclasses will need.
2. Create one or more subclasses that extend the superclass with constants, properties and methods specific to the subclass.
3. Instantiate a couple of objects from the subclasses and execute the methods producing some output.

Lab complete.

Anonymous Classes

Anonymous Classes

[Anonymous classes](#) are useful when simple, one-off objects need to be created. They can contain as many methods as desired, including magic methods. They can implement interfaces and extend super classes.

Use Case

This example tests the `findById()` function. It needs a PDO instance, but without a database, a simulated one will do:

```
function findById(PDO $pdo, $id) {
    $pdo->prepare('SELECT name FROM customers WHERE id = ?');
    $stmt = $pdo->execute([$id]);
    return $stmt->fetch();
}

// Simulation
$fakePDO = new class() extends PDO {
    public function __construct() {}
    public function prepare($stmt, $options = NULL) {}
    public function execute($id) {
        return new class () extends PDOStatement {
            public function fetch($a = NULL, $b = NULL, $c = NULL)
            {
                return ['name' => 'Fred Flintstone'];
            }
        };
    }
};

var_dump(findById($fakePDO, 1));
```

Overriding

Method Overriding

Unless otherwise specified (see next slide), class properties and methods can be overridden. This means that a value assigned to a superclass property can change in a subclass property declaration, or a superclass method overridden in a subclass method.

```
// The superclass
class UserEntity {
    protected $firstName;
    public function setFirstName (string $firstName) {
        $this->firstName = $firstName;
    }
}

// The subclass
class GuestUser extends UserEntity {
    public function setFirstName(string $firstName, string $mi = null) {
        $this->firstname = (!$mi) ? parent::setFirstName($firstName) : $firstName . ' ' . $mi;
    }
}
```

This code will use the inherited method if only a first name is passed, otherwise it overrides the inherited method completely. Calling an inherited method like this when the inherited method is still serviceable is considered a best practice vs. duplicating the inherited code even if only executing a single statement.

Property Overriding

In this example, the super class has a generic table name. The sub class overrides this property.

```
// The super class
class AbstractModel {
    protected $services, $pdo;
    public $tableName = 'generic';
    public function __construct(Services $services) {
        $this->services = $services;
        $this->pdo = $this->services->getDb()->pdo;
    }
}

// The subclass
class UserModel extends AbstractModel {
    protected $userEntity;
    public $tableName = 'user';
    public function __construct(DomainServices $services, UserEntity $userEntity) {
        $this->services = $services;
        $this->pdo = $services->getDomainDb()->pdo;
        $this->userEntity = $userEntity;
    }
}
```

Final Declaration

The [final declaration](#) prevents property and method override in a subclass. A fatal error occurs if attempted.

```
// The superclass
class UserEntity {
    protected $firstName;
    public final function setFirstName ($firstName) {
        $this->firstname = $firstName;
    }
}

// The subclass
class GuestUser extends User {
    // **** NOT ALLOWED!!! **** //
    public function setFirstName($firstName, $mi = null) {
        $this->firstname = ($mi) ? $firstName . ' ' . $mi : $firstName;
    }
}

// output: "Fatal error: Cannot override final method User::setFirstName()"
```

Magic Methods

Magic Methods

[Magic methods](#) are methods with a name starting with two underscores (__) and the method name. The names are reserved.

Here is the current list:

- __construct()
- __destruct()
- __call()
- __callStatic()
- __get()
- __set()
- __isset()
- __unset()

- __sleep()
- __wakeup()
- __toString()
- __invoke()
- __set_state()
- __clone()
- __debugInfo()

Triggers

Magic methods are triggered under certain circumstances, and therefore considered *magic*. One such method `__construct()` has already been discussed. It's triggered when creating a new instance of an object. The opposite is `__destruct()` which is triggered when an object is destroyed, or at runtime completion. Here is a short list of available magic methods:

- `__get()`: Triggered by a get call to an inaccessible property.
- `__set()`: Triggered by a set call to an inaccessible property.
- `__call()`: Triggered by a call to an undefined method.
- `__callStatic()`: Triggered by a static call to an undefined method.
- `__sleep()`: Triggered by a `serialize()` call, and executes prior to serialization.
- `__wakeup()`: Triggered by an `unserialize()` call, and executes prior to object reconstitution.
- `__clone()`: Triggered by the `clone` keyword, and executes prior to duplication.
- `__toString()`: Triggered by treating an object as a string, requires a string return, and does not allow thrown exceptions.
- `__destruct()`: Triggered by `unset()`, or runtime completion, and is called last in/first out (LIFO) sequence.
- `__invoke()`: Triggered by treating an object as a function. Makes the object callable; useful when defining event listeners.

__toString()

Here is an example using `__toString()`, which is active when an object is used as if it were a string:

```
class UserEntity {  
    protected $firstName, $lastName;  
    public function __construct(string $firstName, $lastName) {  
        $this->firstName = $firstName;  
        $this->lastName = $lastName;  
    }  
  
    // Must return a string  
    public function __toString() {  
        return $this->firstName . ' ' . $this->lastName;  
    }  
}  
  
$userEntity = new UserEntity('Mark', 'Watney');  
echo $userEntity; // outputs: "Mark Watney"
```

__get()

Here is an `__get()` example:

```
class UserEntity {  
    protected $firstName, $lastName;  
    public function __construct(string $firstName, string $lastName) {  
        $this->firstName = $firstName;  
        $this->lastName = $lastName;  
    }  
  
    // Returns an inaccessible property  
    public function __get($value) {  
        return $this->$value;  
    }  
}  
  
$userEntity = new UserEntity('Mark', 'Watney');  
echo $userEntity->$firstName; // outputs: "Mark"
```

Best Practices

Keep these practices in mind:

- Use magic `__set()` and `__get()` for special use cases only, and not as a general abstract getter or setter.
- Use explicit property getters and setters to allow certain types of hydrator objects (which populate, or extract data), from objects. They also facilitate writing test units, and API documentation automation.
- Exercise discretion in using anything *magic* as it can cause confusion: magic methods are not invoked explicitly by your program code, but rather indirectly as trigger situations arise.

Magic Method Lab

Lab: Magic Methods

Complete the following:

1. Using the code from the previous exercises, add four magic methods, one of which is the magic constructor.
2. The magic constructor should accept parameters and set those parameters into the object on instantiation.
3. Create an index.php file.
4. Load, or autoload, the created classes.
5. Instantiate object instances, and exercise the magic methods implemented.

Lab complete.

Abstract Classes and Methods

Abstract Classes and Methods

Abstract Classes

[Abstract classes](#) serve the purpose of a superclass, the existence of which serves the inheritance model of concrete subclasses. Subclasses are the objects instantiated, rather than the abstract superclass.

Abstract Form Class

An abstract class example:

```
namespace OrderApp\Core\Form;
// ...
/**
 * Abstract Form Superclass
 */
abstract class Form {
    // ...
    public function getStartTag() {
        return '<form ' . $this->addTagAttributes() . '>';
    }

    public function addTagAttributes(){ ... }

    // ...
}
```

Abstract Methods

An abstract method is one that is marked *abstract*, which is a declaration only without implementation. They serve to mandate method implementation in extending subclasses.

Abstract Methods

In the superclass example below, the method *getInput()* is mandatory, but the implementation is unknown, and is implemented by concretes.

```
namespace OrderApp\Core\Form\Inputs;
// ...
abstract class BaseInput extends GlobalHtmlAtt {
    // ...

    // Upon this declaration, all extending class must provide implementation.
    public abstract function getInput();

    // ...
}
```

An extending concrete subclass:

```
namespace OrderApp\Core\Form\Inputs;
class Text extends BaseInput implements InputInterface {
    public $size, $readonly, $type = 'text';
    public function getInput() {
        $input = "<input type=\"{$this->type}\"";
        // ...
        $input .= '>';
        return $input;
    }
}
```

Abstract Keyword Uses

Use the *abstract* keyword:

- To serve as a superclass, and prevent object instantiation.
- To mandate concrete subclass implementation.

Note: It is possible to use an abstract class, and methods, as a template for other classes.

Important: Load abstract classes prior to loading concrete classes.

Abstract Class Considerations

Notes:

- Classes marked *abstract* cannot be instantiated.
- Abstract classes are commonly used as super classes.
- Classes with abstract methods should be marked *abstract*.
- Abstract classes often serve as a place to put static methods.
- When inheriting from an abstract class with abstract method(s), implementation is provided by subclass.
- Concrete subclasses providing implementation of inherited abstract methods, must apply the same or loosened visibility.
- Concrete subclass method signatures must match inherited abstract method signatures.

Abstract Class and Method Labs

Lab: Abstract Classes

Complete the following:

1. Turn a superclass into an abstract class.
2. In the abstract superclass, define an inheritable abstract method declaration that will instantiate an object of another class, and returns it.
3. Extend the abstract superclass with a concrete subclass implementing the inherited abstract method.
4. Instantiate a subclass instance.
5. Call the method and retrieve the object it builds.

Lab is complete.

Interfaces

Interfaces

An [object interface](#) is a language construct that specifies methods required by implementing classes. Similar to an abstract class declaration, but does not serve the inheritance purpose of a superclass. The declared methods are often required due to an underlying code base need, and/or a desire to type hint on an interface.

The Service Interface

Interface Example:

```
interface ServiceInterface {  
    public function getServices();  
    public function setServices();  
}
```

A class *implements* an interface in the class signature, and by method implementation:

```
abstract class AbstractController implements ServiceInterface  
{  
    protected $services;  
    public function setServices() {  
        $this->services = new Services();  
        return $this ;  
    }  
  
    public function getServices() {  
        return $this->services;  
    }  
}
```

Interface Considerations

Notes:

- Think of an object interface as like a contract, and the methods specified as a contract stipulation.
- Interfaces are implemented in classes using the *implements* keyword in the class signature.
- Interfaces are used when an underlying code base requires certain methods, and by implementation of an interface, the code succeeds.
- Interfaces are commonly type hinted.
- Methods declared in an interface must be public, and can have arguments and type hints specified.
- Interfaces can have constants declared.
- Interfaces can extend other interfaces using the *extends* operator. It is the only construct that allows multiple inheritance.
- Multiple interface implementations are possible separated by a comma.

Best Practices

Include:

- Use interfaces to allow for alternate implementations on type hints.
- Naming convention consistent with class names: Begin with upper case letter followed by camel casing.
- Keep interfaces small and focused.
- Base interfaces on a particular, or set of particular, behaviors desired.

Interface Lab

Lab: Interfaces

Complete the following:

1. Create an object interface with two methods.
2. Implement the interface in your superclass.
3. Add some code to the index.php file that calls one of the superclass methods implemented.

Lab is completed.

Type Hinting

Type Hinting

[Type hinting](#), in general, is specifying a particular data type is passed, whether that is a scalar type, array, callable, class or interface name.

Type hinting on a class name will ensure only an instance of the specified class is passed. This can be very limiting however as alternate implementations are blocked.

Type hinting on interfaces allows alternate implementations. That means type hints can specify an interface allowing different objects passed. If a number of classes implement a particular interface, objects of those classes will qualify the interface type hint.

Interface type hinting makes it much easier to write test code, such as mock objects or test doubles, which can be passed and tested in substitution for an actual object.

Scalar

This class setter requires a string type passed in the parameter:

```
class UserEntity {  
    // ...  
    public function setFirstName(string $firstname) {  
        $this->firstname = $firstname;  
    }  
}
```

Array

This class constructor requires an array type passed in the parameter:

```
class Test {  
    protected $config = [];  
  
    public function __construct(array $config ) {  
        $this-> config = $config;  
    }  
}
```

Callable

This type hint requires a *callable* type passed, which the callback closure provides:

```
class Test {  
    public function callIt(callable $callback, array $params) {  
        return $callback($params);  
    }  
}  
  
$operands[0] = 2;  
$operands[1] = 3;  
$callback = function ($p) {  
    return 'The result of '  
        . $p[0] . ' times ' . $p[1]  
        . ' is ' . ($p[0] * $p[1]);  
};  
  
$test = new Test;  
echo $test->callIt($callback, $operands);  
// output: "The result of 2 times 3 is 6"
```

Class and Interface

Example class name type hint in the constructor. Here a Service instance is required:

```
use OrderApp\Core\Service\Services;
class AbstractModel implements ModelInterface {
    // ...

    public function __construct(Services $services) {
        $this->services = $services;
        $this->db = $this->services->getDb();
    }
}
```

Here's the same example, but with an interface type hint. Here an instance implementing the interface is required.

```
use OrderApp\Core\Service\Services;
class AbstractModel implements ModelInterface {
    // ...

    public function __construct(ServiceInterface $services) {
        $this->services = $services;
        $this->db = $this->services->getDb();
    }
}
```


Nullable Types

There may be times when you want to use type hinting, but you also want the ability to pass `NULL` as a value. This is very often the case where you have more than one optional parameter where the middle parameter might be skipped.

Below is a class which produces a set of "topics" using `H2`, `H3` and `UL / LI` tags.

```
class Topic {
    protected $title, $subTitle, $bullets;
    public function __construct(string $title, ?string $subTitle, array $bullets) {
        $this->title = $title;
        $this->subTitle = $subTitle;
        $this->bullets = $bullets;
    }
}
```

Strict Typing

Declaration is applied at the file level and specified at the very beginning of the file after the PHP start tag.

Applies strict typing only for the file code. If declared, all code contained in the file that has type declarations, including function/method return types, will be strictly enforced.

```
declare(strict_types=1);

class Select extends BaseInput implements InputInterface {
    protected $arg1;
    protected $arg2;

    public function __construct(string $arg1, int $arg2) {
        $this->arg1 = $arg1 ;
        $this->arg2 = $arg2 ;
    }
}
```

Return Typing

A function or method return type is specified in the signature.

Note: The use of `declare(strict_types=1)` is not needed for declarations only. Declaring strict typing is necessary, however, if strict typing enforcement of input arguments is desired.

```
class Select extends BaseInput implements InputInterface {
    public function getInput( ) : string {
        // ...
    }

    public function setOptions( array $options ) : InputInterface {
        // ...
    }

    public function isAutoFocus() : bool {
        return $this->isAutoFocus;
    }
}
```

Considerations

Notes:

- Type hinting improves code quality by requiring specific data types passed. Failure to pass the correct datatype is easily caught by test code, and in development.
- Strict typing with the declare function is optional, but, if declared is enforced.
- Strict typing is enforced at the file level.

Best Practices:

- Use where there is a possibility of incorrect data type passed, or for clean coding practice.
- Use interface type hints rather than class-specific type hints for more flexibility.
- Limit strict typing on a case-by-case basis as needed.
- Implement strict typing on a file by file basis, and confirm operation with tests.

Strict Typing Lab

Lab: Type Hinting

Complete the following:

1. Create a new class with some properties and methods.
2. Add a constructor.
3. Type hint in the constructor for the interface created in the last exercise.
4. Instantiate an object from one of your previous subclasses.
5. Add it as a dependent object to the new object created in step one, and store it.

Lab is complete.

Exceptions

Exceptions

An [exception](#) is an object based on a built-in base class, which is a class that's always available. It is similar to regular objects, but different by one distinct uniqueness -- an exception is thrown with the `throw new` keywords rather than instantiated with the `new` keyword.

An exception is normally thrown in a *try/catch* language construct. The *try/catch* construct is built to catch and handle a thrown exception.

```
try {  
    throw new Exception( 'Something bad happened', 401);  
} catch (Exception $e ) {  
    $logEntry = time() . '|' . $e->getMessage() . '|' . $e->getCode();  
    error_log($logEntry, 3, 'path/to/error_log.php');  
}
```

Note: All built-in classes reside in the *Global* namespace, which requires an appropriate *use* statement.

Custom Exceptions

A custom exception is defined to name a particular type of exception, and/or provide additional functionality. A custom exception should always extend the built-in base exception class and call itâ€™s constructor, if the custom exception defines a constructor.

```
class ModelException extends Exception {  
    public function __construct($msg, $en = null) {  
        parent::__construct($msg, $en);  
        // ...  
    }  
}
```

SPL Exceptions

The *Standard PHP Library* (SPL) contains a number of built-in [classes which extend Exception](#), and can be used at any place within an application.

Here is a list of additional Exception classes offered by the SPL:

BadFunctionCallException
BadMethodCallException
DomainException
InvalidArgumentException
LengthException
LogicException
OutOfBoundsException

OutOfRangeException
OverflowException
RangeException
RuntimeException
UnderflowException
UnexpectedValueException

Try and Multiple Catch Blocks

Multiple catch blocks are possible. If handling the built-in exception, the catch block must always be last in the runtime sequence.

```
try {  
    // ...  
} catch (PDOException $e) {  
    $logEntry = time() . '|' . get_class($e) . ':' . $e->getMessage() . PHP_EOL;  
    error_log($logEntry, 3, 'path/to/database_error_log.php');  
} catch (Exception $e) {  
    $logEntry = time() . '|' . get_class($e) . ':' . $e->getMessage() . PHP_EOL;  
    error_log($logEntry, 3, 'path/to/error_log.php');  
}
```

Try, Multi-Catch, and Finally Blocks

A finally block appends to the try/catch construct as an option. Code contained in a finally block is executed under all circumstances.

```
try {
    $pdo = new PDO('mysql:host=localhost;dbname=course', 'vagrant', 'vagrant');
    if (!$result = $pdo->exec("Select * from Orders"))
        throw new Exception("Unable to access data");
} catch (PDOException $e) {
    $logEntry = time() . '|' . get_class($e) . ':' . $e->getMessage() . PHP_EOL;
    error_log($logEntry, 3, 'path/to/database_error_log.php');
} catch (Exception $e) {
    $logEntry = time() . '|' . get_class($e) . ':' . $e->getMessage() . PHP_EOL;
    error_log($logEntry, 3, 'path/to/error_log.php');
} finally {
    error_log('Database Access Attempted: ' . date('Y-m-d H:i:s'), 3,
        'path/to/access.log');
}
```

Multiple Catch Type Hints

Multiple catch type hints in a single catch are possible:

```
try {  
    // ...  
} catch (ExceptionType1 | ExceptionType2 $e) {  
    // Handle either exception ...  
} catch (Exception $e) {  
    // Handle the base exception ...  
}
```

Throwable Interface

The [throwable](#) interface is the base interface for any object that can be thrown via a throw statement, or by the default *Error* object, and cannot be implemented directly, but used by extending the base *Exception* class. The base *Exception* implements the *Throwable* interface.

A *Throwable* catch block type hint can trap both base *Exception*, and *Error* objects.

```
try {  
    // ...  
} catch (Throwable $e) {  
    // Handle exception and error objects, and custom exceptions by the same handling code ...  
}
```

Error Class

The [Error class](#) is the base class for all application errors thrown by the parser.

In addition, there are several classes which extend *Error*, most notably:

- [ParseError](#)
- [TypeError](#)

The errors: *ArgumentCountError*, *ArithmeticError*, *AssertionError*, *DivisionByZeroError*, and *CompileError* inherit from the base *Error* class.

Error Class

There are certain situations where *Error* is not thrown:

- Infinitely recursive function calls
- Infinite loops
- Memory limit exceeded
- Anything which results in a *Segmentation fault*

Here is an example of bad code which will *not* throw an *Error*.

```
function badBad() {  
    // results in "Segmentation fault (core dumped)"  
    return badBad();  
}  
  
try {  
    badBad();  
} catch(Error $e) {  
    echo $e . PHP_EOL;  
}
```


Exception Lab

Lab: Build Custom Exception Class

Complete the following:

1. Create a file and build a custom exception class with a constructor that accepts parameters.
2. Call the parent *Exception* constructor.
3. Add some new functionality in the custom exception constructor.
4. Add a try/catch/catch/finally block set.
5. In the try portion, throw an instance of the Exceptions object, and an instance of the custom exception class.
6. Handle both by logging in the associated catch blocks.
7. Echo something in the finally block.

Lab is complete.

Static Properties and Methods

Static Properties and Methods

A class can contain static properties and methods. A static property or method is one that is accessible without having an object instance.

```
class Currency {  
    public static $currency = '$' ;  
    public static $before = TRUE;  
  
    public static function format($arg) {  
        return ((self::$before) ? self::$currency : "  
            . sprintf('%.2f', $arg)  
            . ((!self::$before) ? self::$currency : "");  
    }  
  
    public static function setCurrency($arg) {  
        self::$currency = $arg;  
    }  
  
    public static function setPlacement($arg) {  
        self::$before = $arg;  
    }  
}
```

Static Access

Accessing a [static](#) property or method is possible by using the class name, the scope resolution operator (::) and the name of the static property or method.

Example.

```
class Currency {  
    // as defined in previous slide  
}  
$symbol = Currency::$currency; // $symbol == "$"  
echo Currency::formatCurrency(100); // output: "$100.00"
```

Considerations

Notes:

- Static properties and methods observe the rules of visibility settings.
- Static reference is made within the class using the self keyword. Self is bound to the current class name. Do not use \$this within static methods.
- Static keyword denotes static accessibility at the property and method declaration.
- A collection of static methods and properties are sometimes grouped in a class, because they don't fit anywhere else.
- Static members are sometimes considered procedural code in a class wrapper.
- Static methods can be difficult to test.

Best Practices:

- Limit the use to creating objects, and when needed for utility.
- Use as a technique to provide a singleton instance.

Late Static Binding

Late static binding refers, in part, to a binding for the *static::* keyword, to identify the calling class. Itâ€™s what makes static inheritance work.

The static keyword is not bound to a value until late at runtime when references are made to the class inheritance structure. At that time, the static keyword is bound to the calling class.

The value bound to *static::* is evaluated at runtime just prior to the point of use. Think of this as an internal resolution and resolved, and not resolved at class definition.

The value bound at run time to *self::* is always the current class, therefore the late *static* binding is necessary for the calling class reference.

The value bound at run time to *parent::* is always the immediate parent or superclass of the current class.

Ref: [Late Static Binding](#)

Late Static Binding

An example:

```
class Superclass {
    public static function getClass_name() { return __CLASS__; }
    public static function getClass() {
        $output = self::getClass_name(); // bound to the current class
        $output .= ' : ' ;
        $output .= static::getClass_name(); // bound to calling class
        $output .= PHP_EOL;
        return $output;
    }
}

class Subclass extends Superclass {
    public static function getClass_name() { return __CLASS__; }
}

echo Superclass::getClass();
echo Subclass::getClass();
// echos: "Superclass : Superclass \n Superclass : Subclass"
```


Polymorphism

Polymorphism

The root meaning of *polymorphism* is many.

It is a software design principal that allows a single interface to objects of different types. Different types of *Polymorphism* exist, but for PHP the focus is on *subtype polymorphism*.

Subtype polymorphism denotes different subclasses that are related to a superclass. It is what makes subclassing and object/interface type hinting work.

It limits what is passable as a type hint. Class type hints allow passing objects of the typed class, and all subtypes.

It does not allow ancestor class instances passed to a subclass type hint.

Why is it Important?

Subclasses have pointers to their ancestors, which is consulted making inherited properties and methods possible.

Late binding of name references is a result of consultation with ancestors at runtime. It's what makes binding the static keyword to the calling class work in static inheritance and *late static binding*.

Subtype polymorphism impacts exception catch block sequence. The higher the exception type hint is in the inheritance tree, the lower it should be type hinted for in a series of catch blocks.

Use Case

Form Class

In the *OrderApp* there is a super class *OrderApp\Core\Form\Form* that all domain forms extend:

```
namespace OrderApp\Core\Form;
use OrderApp\Core\Form\{Inputs, InputInterface};
abstract class Form {
    protected ...
    public function __construct(array $args) { ... }
    public function getStartTag() { ... }
    protected function addTagAttributes() { ... }
    public function getEndTag() { ... }
    public function addElement($args) : InputInterface { ... }
    public function setModels($models) { ... }
    public function getModels() { ... }
    public function setConfig($config) { ... }
    public function getConfig() { ... }
    public function getElements() { ... }
    public function setData(array $post) { ... }
    public function getData() { ... }
}
```

Use Case

Concrete Subclasses

There are two concrete sub-classes which extend the *Form* superclass.

The *OrderForm* class:

```
namespace OrderApp\Form;  
use OrderApp\Core\Form\Form, Service\Services;  
class OrderForm extends Form { ... }
```

The *AddOrderForm* class:

```
namespace OrderApp\Form;  
use OrderApp\Core\Form\Form, Service\Services, Form\Inputs>SelectAssoc;  
class AddOrderForm extends Form { ... }
```

Use Case

Type Hint

The *Domain* class processes form inputs, and the method *processInput()* requires an instance of *OrderApp\Core\Form\Form* as an argument type hint:

```
namespace OrderApp\Domain;
use OrderApp\Core\Service\Services, Form\Form;
use OrderApp\Form {OrderForm, AddOrderForm};
use OrderApp\Service\DomainService;
class Domain {
    // ...
    public function processInput(Form $form) {
        switch ($form) {
            case $form instanceof OrderForm :
                // not all code is shown
            case $form instanceof AddOrderForm :
                // not all code is shown
        }
        // ...
    }
}
```

The *Form* type hint allows all concrete subclasses of the *Form* base class. That means the *OrderForm* and *AddOrderForm* instances will pass the superclass *Form* type hint.

Traits

Traits

A [Trait](#) is a language construct that allow blocks of code inserted into more than one class at a time.

A Trait is a code reuse mechanism, and used in a class by the *use* operator.

Traits have precedence, visibility, and a potential for naming collisions.

Trait precedence rules apply to method naming collisions. If a trait has methods name the same as a current class, or ancestor class, trait precedence rules come into play:

- Subclass methods override trait methods.
- Trait methods override ancestor methods.

Trait Example

```
trait GroundVehicleTrait {  
    public $steeringWheelDiameter;  
  
    public function setSteeringWheelDiameter(int $value) {  
        $this->setSteeringWheelDiameter() = $value;  
    }  
  
    public function getSteeringWheelDiameter() : int {  
        return $this->steeringWheelDiameter;  
    }  
}
```

Trait Used in a Class

```
class Car {  
    use GroundVehicleTrait;  
    // ...  
}  
  
class Truck {  
    use GroundVehicleTrait;  
    // ...  
}
```

Visibilities

Visibilities are changeable in the using class. These visibilities can be tightened or loosened as the class needs dictate.

Trait naming duplication between traits creates a conflict and must be resolved. A method named the same in another trait is resolved by the *insteadof* operator, and defining which method takes precedence.

```
trait GroundVehicleTrait {  
    public function getType(){  
        return $this->type;  
    }  
    // ...  
}  
  
Class Vehicle {  
    use GroundVehicleTrait {getType as protected}  
}
```

Naming Aliases

Trait method aliases allow for use of methods that are otherwise involved with a naming conflict, using the `as` operator:

```
trait GroundVehicleTrait {  
    public function getType(){  
        return $this->type;  
    }  
    // ...  
}  
  
trait AirVehicleTrait {  
    public function getType(){  
        return $this->type;  
    }  
    // ...  
}  
  
Class Vehicle {  
    use GroundVehicleTrait, AirVehicleTrait {  
        GroundVehicleTrait::getType insteadof AirVehicleTrait,  
        AirVehicleTrait::getType as getAirType;  
    }  
}
```

Trait Lab

Lab: Traits

Complete the following:

1. In separate files, create two traits, each with two methods, one of the methods named the same in both traits.
2. In another file, create a class that uses the two traits.
3. Resolve the naming collision, and change the method visibilities.
4. Instantiate an instance of the class and execute the trait methods.

This lab is complete.

Object Cloning

Object Cloning

[Object cloning](#) means *objects* are implicitly passed by reference. As a result, a way to copy an object is necessary and provided by the *clone* keyword.

```
require 'UserEntity.php';
$user1 = new UserEntity();
// $user2 is now a "backup" with the original values

$user2 = clone $user1;
// $user1 can now be modified with desired values

$user1->setFirstName('Mark');
$user1->setLastName('Watney');
```


Object Comparisons

Now that an object is cloneable, a way to distinguish if an object is a clone, or a reference becomes necessary:

```
require 'User.php';
$user1 = new User();
$user2 = clone $user1;
var_dump($user1 == $user2); // TRUE: values are the same at this point
var_dump($user1 === $user2); // FALSE: not the same object!

$user3 = $user1;
$user1->setFirstName('Julia');
$user1->setLastName('Roberts');
var_dump($user1 == $user2); // FALSE: values are now different
var_dump($user1 == $user3); // TRUE: values are the same
var_dump($user1 === $user3); // TRUE: objects are the same
```

Use Case

Consider the following example involving the *DateTime* class. A 30/60/90 day aging report is required. The *DateTime::add()* method creates three dates:

```
$date = new DateTime(); // today's date
for ($x = 30; $x < 100; $x += 30) {
    $day[$x] = $date;
    $day[$x]->add(new DateInterval('P' . $x . 'D'));
    echo '
' . $day[$x]->format('Y-m-d') . PHP_EOL;
}
// outputs:
<br>2017-11-02
<br>2018-01-01
<br>2018-04-01
```

When the code is run, the intervals end up as today + 30, today + 90 and today + 120!

Since objects are passed by reference, when a new interval is added, the effect is cumulative.

Ref: </path/to/sandbox/public/ModOOP/Cloning.php>

Use Case

In order to use the same basis for today's date, use the *clone* keyword:

```
$date = new DateTime(); // today's date
for ($x = 30; $x < 100; $x += 30) {
    $day[$x] = clone $date;
    $day[$x]->add(new DateInterval('P' . $x . 'D'));
    echo '<br>' . $day[$x]->format('Y-m-d') . PHP_EOL;
}
// outputs:
<br>2017-11-02
<br>2017-12-02
<br>2018-01-01
```

Ref: </path/to/sandbox/public/ModOOP/Cloning.php>.

Considerations

Things to keep in mind:

- Cloning an object becomes useful when a number of changes are made after the original instantiation.
- If a duplicate object is intended, be sure to clone to preserve the original.
- Be careful with an object modification if it is possible that it is intrinsically a reference, check to be sure.
- The magic `__clone()` method is triggered by the `clone` keyword.
- If a superclass and subclass have a magic `__clone()` method, the parent::`__clone()` must be explicitly invoked from the subclass.

Module Summary

This module included:

- PHP Namespaces
- Class and class members
- Objects
- Inheritance
- Overriding
- Magic methods
- Abstract classes and methods
- Interfaces
- Type hinting
- Exceptions
- Static properties and methods
- Polymorphism
- Traits
- Object cloning

OrderApp OOP Implementation

This module covers:

- OrderApp OOP implementation
- File organization
- Application classes
- Layout scripts

OrderApp OOP Implementation

OrderApp OOP Implementation

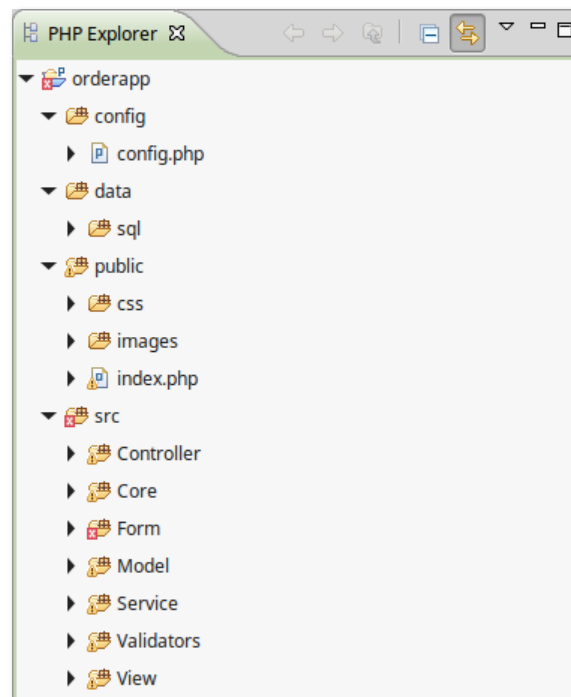
The OrderApp was developed for the PHP Fundamentals courses and covers all basic features of a web-based application, such as form validation and database storage

This module will take the PHP object oriented constructs learned in the last module and apply them to the OrderApp application. You will see how most of them are implemented.

The application implements a number of software design patterns. Where it does, the associated design pattern is identified or discussed along with a code example.

File Organization

File Organization



The Config Directory

The "config" directory contains the OrderApp configuration. Itâ€™s a file, when called, returns an array of configuration.

Using PHP arrays to contain configuration options provides a native data structure directly to an application without the need for parsing.

The config array contains top level keys for database configuration. The key "db" provides configuration for the database server, and a "services" key that provide application-wide service configuration. A concept discussion of a service, or service layer is forthcoming.

The Data Directory

The "data" directory provides a location to write to, and it contains the OrderApp SQL dump.

It provides a write to location for application code.

This directory needs both read and write access by the web server user, in the case of our class virtual machine, the "www-data" user.

Best Practice: It is considered best to not write data to application directories, or in this case, anything in the other directories, or especially the "src" directory. If needed, write to the data directory.

The Public Directory

The "public" directory contains the entry `index.php` file, and directories and files for the frontend.

Many applications use a directory like this to contain folders for CSS, HTML snippets, JavaScript, and in some cases, web server configuration files like Apache `.htaccess` and template files.

The src Directory

The "src" directory is where the application files live. This is the architectural structure of the application. It contains directories for the following file types:

Controller: Contains the application controller class. Controller classes are the "C" part of the Model/View/Controller (MVC) design pattern.

Core: Contains global classes that are non-application-specific class and can be considered similar to framework classes.

Form: Contains application form classes. Form classes model HTML forms and input classes model HTML form tags, like the HTML "<input >" tag.

Model: Contains application model classes and interfaces. Model classes are the "M" part of the MVC design pattern acronym. They interface the database and provide data to the application.

Domain: Contains the application domain class. The Domain class provides a container for business logic.

View: Contains a layout directory with application HTML layout scripts.

The src Directory

Core Components

The core components are class that are not application-specific, but contain re-usable code that serves the application. All of the classes and interfaces are extensible.

- **Controller:** Contains the abstract controller superclass.
- **DB:** Contains the abstract model superclass, the DB PDO wrapper class, the model exception classes, and a ModelInterface.
- **Form:** Contains the form base class and input classes. These classes abstract an HTML form with inputs.
- **HTML:** Contains a global HTML attributes class.
- **Messaging:** Contains a global messaging class.
- **Service:** Contains a service interface and services class. Service classes provide an extra layer of structure to an application beyond those found in the MVC design pattern.
- **Validator:** Contains validation classes to represent certain types of validation, and a validator interface.
- **View** A container class for view variables, and also is responsible for rendering layouts. This class is non-application specific and extensible by the application.

The src Directory

Application Classes

In addition to application controllers, the classes that directly relate to the application being built. These define the application architecture and are organized and created as the domain requires.

- **The Controller directory:** Contains the initial controller called by the index file and other controllers as dictated by the application.
- **The Domain directory:** The domain (business) logic container.
- **The Form directory:** Classes that build domain forms.
- **The Model directory:** Database table gateway classes.
- **The View directory:** The HTML layout files that are rendered by the browser.

We will detail each of these in the next slides.

Application Classes

Application Classes

Controllers

Controller: Contains controller classes the OrderApp needs. We kept it simple and only have one controller, the "IndexController".

Domain

Contains domain methods.

- Contains domain (business) logic.
- Defines and contains the domain models.
- Processes form input for the application forms.
- Employs the domain models to retrieve application data.
- Provides utility functionality.

Forms

Forms are classes that provide an object oriented interface to creating HTML forms. Form classes define a constructor which:

- Sets form tag attributes.
- Calls the parent constructor passing the tag attribute arguments.
- Calls the inherited `addElement()` method adding element configuration.
- Sometimes calls a model object method to retrieve database information for use in an element.
- Inherits from the *Form* superclass.

Models

Provide an object oriented interface to SQL database tables.

- Includes a model interface that requires a constructor injected with a service instance.
- Model classes that represent examples of the table data gateway design pattern.
- The database object is retrieved using the services instance within the abstract model superclass.

Layout Scripts

Layout scripts are where the HTML lives.

- They include the main HTML file and layouts representing the application forms.
- In the OrderApp, the default HTML merges with the forms to provide the application views.

Module Summary

This module covered Object Oriented Implementation of the OrderApp

Data Persistence

This module covers:

- Relations and SQL
- Data modeling
- PHP Data Objects (PDO)
- Database operations

Relations and SQL

Relational Data Models and Structured Query Language (SQL)

A special programming language that interface relational databases in order to build schema, or perform create, read, update, and delete operations on data.

Queries are groupings of statements stringed together. Statements are terminated with semicolons.

Two major SQL statement groups:

- **DML:**
Data Manipulation Language commands perform CRUD operations on data
- **DDL:**
Data Definition Language commands perform structural manipulation, like creating databases and tables, or removing them

DML Keywords

The most frequent DML keywords follow.

The acronym **CRUD** stands for Create|Read|Update|Delete, and is in popular use regarding the major data manipulation.

- **INSERT:** Inserts new data into a table. It is the "C" in CRUD.
- **SELECT:** Reads data from one or more tables. It is the "R" in CRUD.
- **UPDATE:** Updates existing data in a table. It is the "U" in CRUD.
- **DELETE:** Deletes existing data in a table. It is the "D" in CRUD.
- **JOIN:** Joins tables together based on mapped keys. It's what makes relational databases relational.
- **FROM:** Used with SELECT, UPDATE and DELETE to specify a table.
- **WHERE:** Specifies conditional logic within a statement.
- ...

DML Keywords (cont.)

- ...
- **BEGIN...END:** Encapsulates a transaction.
- **CALL:** Calls a stored procedure.
- **WHILE:** Sets a repeatable condition of a statement block.
- **ORDER BY:** Sets the result set sort order.
- **LIMIT:** How many rows in the result set.
- **OFFSET:** Specifies how many rows to skip.
- **GROUP BY:** Collects data across multiple records and groups the results by one or more columns.

DDL Keywords

The most frequent DDL keywords follow.

- **CREATE DATABASE:** Creates a new database.
- **DROP DATABASE:** Deletes/drops a database.
- **CREATE TABLE:** Creates a new table.
- **DROP TABLE:** Deletes/drops a table.
- **ALTER TABLE:** Alters a table schema.

Relations and SQL Labs

Lab: SQL Statements

Identify the result of each of the following SQL statements:

```
SELECT * FROM users;  
SELECT firstname, lastname FROM users AS u WHERE u.id = 25;  
INSERT INTO users (firstname, lastname) VALUES(James, Bond);  
UPDATE users SET firstname=Rube, lastname=Goldberg WHERE users.id=420;  
DELETE FROM users WHERE firstname=Rube;  
SELECT * FROM users ORDER BY lastname DESC;
```

Data Modeling

Data Modeling

Design practice that creates and maps a database system to a domain.

A **domain** is a field, area, or discipline. An commercial enterprise is considered a domain, but may have other domains within called subdomains.

Data modeling with a relational database system is an exercise of designing a solution by defining domain entities (employee, service, billing, etc.) to a database and tables, and how the tables relate to each other.

Data modeling defines relationships types between tables, including:

- **One-to-One:** How a single table row maps directly to a single row in another table.
- **One-to-Many:** How a table row maps to multiple table rows in another table.
- **Many-to-Many:** How multiple table rows map to multiple table rows in other tables.
- **Many-to-One:** How multiple table rows map to a single table row in another table.

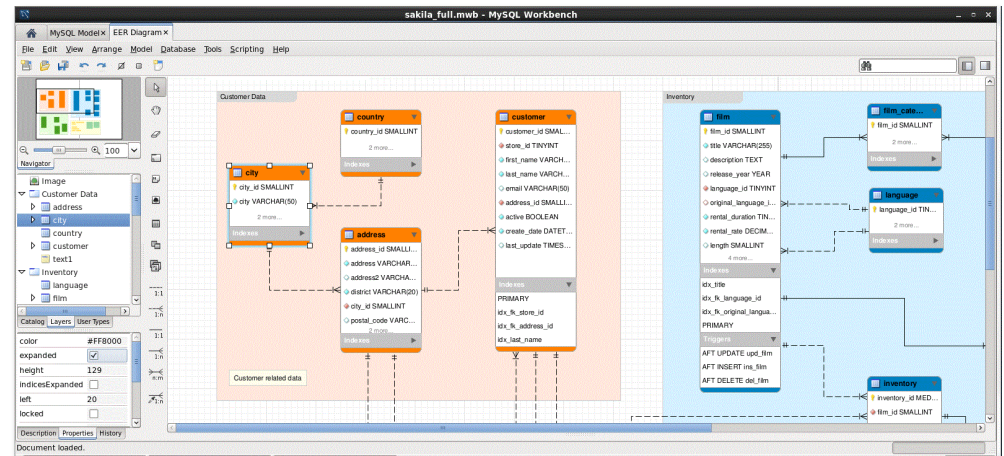
Data Modeling

Tools

Most relational data bases have tools written to help with data modeling for the Domain.

All database engines have modeling tools. Find one that works for your needs and learn to use it.

Here's an example of one used to model MySQL databases. It's called [MySQL Workbench](#).



Domain Model

A **Domain Model** is a loose term for a pattern that comprises multiple patterns which offers full domain and persistence abstraction in the modeling layer. Appropriate to use when you want to have complete ignorance of the implemented persistence in the API of the model with which a controller will interact

A pattern of patterns

- **Creational:** Factory, Singleton, Builder.
- **Structural:** Adapter, Module, Decorator, Proxy.
- **Behaviorial:** Pub/Sub, Strategy, Iterator.
- **Concurrency:** Active Object, Lock, Reactor, Scheduler.

Domain Model

Model Components

Domain Model components include:

- Domain entities
- Domain value objects
- Hydrators and mappers
- Repositories

Model Components

Entities

Entities are important and meaningful components of a domain, like an employee, an event registrant, a shipping, etc. They

- are modeled by classes to represent a particular domain entity.
- always have an identity of some kind.

Entities

RegistrantEntity

A *RegistrantEntity* class modeling a person registering for an event, like a concert, conference, etc. A *Registrant* is what the domain calls this component, so it's named the same:

```
class RegistrantEntity {
    protected $id, $firstname, $lastname;
    public function __construct(string $firstname, string $lastname) {
        $this->firstname = $firstname;
        $this->lastname = $lastname;
    }
    public function getFirstname(): string { return $this->firstname; }
    public function getLastName(): string { return $this->lastname; }
    public function getId() { return $this->id; }
}
```

Entities

EventEntity

An *EventEntity* class modeling an event, like a concert, conference, etc. The domain calls this component an *Event*:

```
class EventEntity {  
    protected $id, $name, $startdate, $enddate;  
    public function getId() { return $this->id; }  
    public function setId($id): void { $this->id = $id; }  
    public function getName() { return $this->name; }  
    public function setName($name): void { $this->name = $name; }  
    public function getStartDate() { return $this->startdate; }  
    public function setStartDate($startdate): void { $this->startdate = $startdate; }  
    public function getEndDate() { return $this->enddate; }  
    public function setEndDate($enddate): void { $this->enddate = $enddate; }  
}
```

Entities

RegistrationEntity

A *RegistrationEntity* class modeling a registration. The domain calls this component a *Registration*:

```
class RegistrationEntity {  
    protected $id, $eventEntity, $registrantEntity, $registrationTime;  
    public function getId(): int  
        { return $this->id; }  
    public function setId(int $id): void  
        { $this->id = $id; }  
    public function getEvent()  
        { return $this->eventEntity; }  
    public function setEvent($eventEntity): void  
        { $this->eventEntity = $eventEntity; }  
    public function getRegistrant()  
        { return $this->registrantEntity; }  
    public function setRegistrant($registrantEntity): void  
        { $this->registrantEntity = $registrantEntity; }  
    public function getRegistrationTime()  
        { return $this->registrationTime; }  
    public function setRegistrationTime($registrationTime): void  
        { $this->registrationTime = $registrationTime; }  
}
```


Value Objects

Domain value objects are immutable and transient. They exist to add value as required by the domain.

Value objects are defined when it is deemed that their properties have a high value. An example in an educational domain might be a list of courses, or a currency used for tuition payment. In a different domain, that same list of courses, or currency may require identification and therefore would be an entity.

With value objects, it is unnecessary to care about whether one object is a reference to another, or a completely different, but equal object.

A change on an immutable value object would generate a new object.

Currency Value Object

Here the *Currency* value object class models a currency.

```
class CurrencyValueObject {  
    protected $amount, $symbol;  
    public function __construct(float $amount, string $symbol)  
    {  
        $this->amount = $amount;  
        $this->amount = $symbol;  
    }  
}
```

Hydrators and Mappers

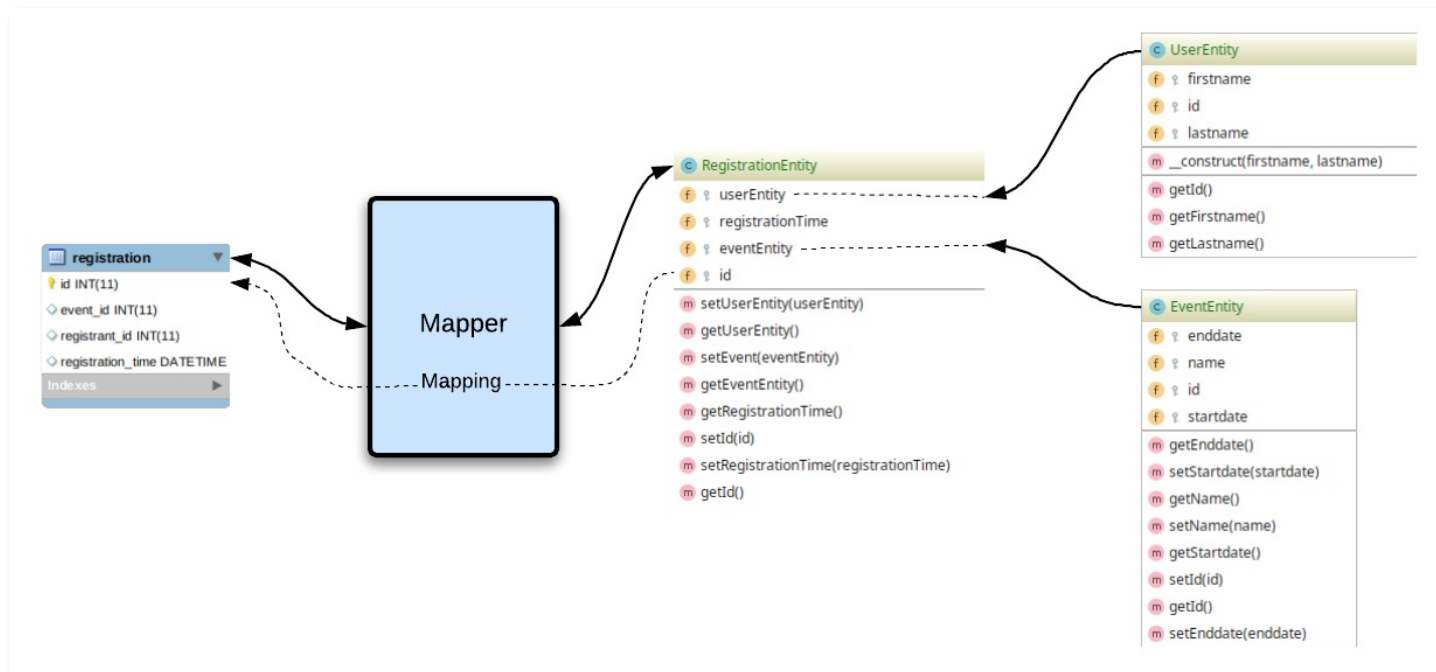
Some components in a domain model are only known to the developers building the software. Hydrators and mappers are two such components:

- **Hydrators:** Populate, or extract, data from an entity (object model).
- **Mappers:** Maps the object model to the data model.

A database might have a column named *first_name* , but the object model property name is *\$firstName*, so they are not directly linkable without a mapper. The mapper maps the two together.

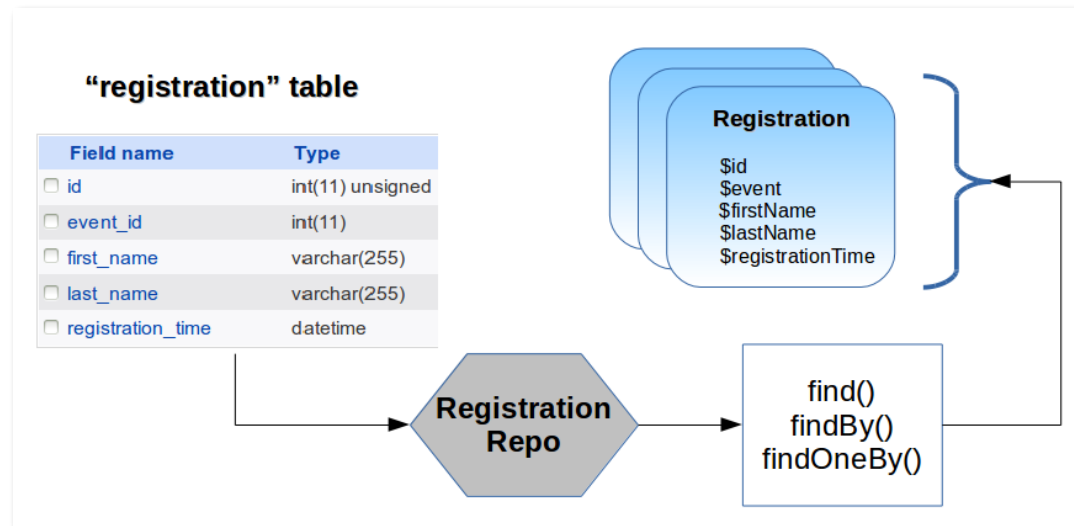
Mapping

A mapper maps the entities to their associated database table(s).



Repositories

Repositories act as *gateways* to a one or more entities. When a client (controller, model, etc.) needs an entity, or a collection of entities, repositories provide them. Depending on how big the model is, repository classes may, or may not, reside in the same directory, or namespace, as the entity classes they manage.



Mapper Libraries

An Object Relational Mapper (ORM) is a third party library which provides object to data model mapping. Here are a few:



PHP Data Objects (PDO)

PDO Overview

A lightweight interface library for database access. It permits platform access abstraction, this means its only knowledge of the platform is at connection.

Following are access abstraction benefits.

- No knowledge of the platform except at connection.
- No platform-specific functions necessary. Code is portable.
- Parameter escaping is automatic.
- Built-in support for SQL prepared statements, stored procedures, and transactions.

PDO API

An [API](#) for OOP connection abstraction

- **PDO:** Represents the connection object; has direct query capability; supports prepared statements and transactions.
- **PDOStatement:** Represents a prepared statement and result set; supports parameter, value and column bindings; supports multiple fetch return types.
- **PDOException:** PDO specific exceptions.
- **PDO Driver Classes:** CUBRID Functions (PDO_CUBRID), Microsoft SQL Server and Sybase Functions (PDO_DBLIB), Firebird Functions (PDO_FIREBIRD), IBM Functions (PDO_IBM), Informix Functions (PDO_INFORMIX), MySQL Functions (PDO_MYSQL), Microsoft SQL Server Functions (PDO_SQLSRV), Oracle Functions (PDO_OCI), ODBC and DB2 Functions (PDO_ODBC), PostgreSQL Functions (PDO_PGSQL), SQLite Functions (PDO_SQLITE), 4D (PDO)

PDO Fetch Modes

[PDO API constants](#) represent integers particular to the PDO API. Among them are fetch method constants that tailor the result types.

Fetch Method Constants

- **PDO::FETCH_ASSOC:** Returns an associative array by column name.
- **PDO::FETCH_NUM:** Returns a numeric array.
- **PDO::FETCH_BOTH:** Returns both associative and numeric arrays.
- **PDO::FETCH_OBJ:** Returns each row as an object with property names that correspond to the column names returned in the result set.
- **PDO::FETCH_CLASS:** Returns a new instance of the requested class, mapping the columns to named properties in the class.
- **PDO::FETCH_INT:** Updates an existing instance of the requested class, mapping the columns to named properties in the class.

PDO with try / catch

The PDO API should always execute inside a try/catch construct and the PDO connection object set with an exception error type attribute.

```
try {  
    // Get the connection instance  
    $pdo = new PDO('mysql:host=localhost;dbname=course', 'vagrant', 'vagrant');  
    // Set error mode attribute  
    $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);  
    // Statements ...  
} catch (PDOException $e) {  
    // Handle exception...  
}
```

Note: Try/Catch constructs should be used whenever working with an API that can throw an exception, or when throwing a custom exception manually.

Using PDO

Connect to the database by creating a PDO instance. The first argument is referred to as a Data Source Name (DSN). Use sockets if available, otherwise use "host" (which then has to go through the TCP/IP stack).

```
$pdo = new PDO('mysql:unix_socket=/var/run/mysqld/mysqld.sock;  
              dbname=course','vagrant','vagrant');
```

Get the connection PDO instance via Unix socket

Or

```
$pdo = new PDO('mysql:host=localhost;dbname=course', 'vagrant', 'vagrant');
```

Get the connection PDO instance from a host IP

The SELECT Statement

```
try {  
    // Execute a one-off SQL statement and get a statement object  
    $stmt = $pdo->query('SELECT * FROM orders');  
  
    // Returns an associative array indexed by column name  
    $results = $stmt->fetchAll(PDO::FETCH_ASSOC);  
  
    // Output the results  
    print_r($results);  
} catch (PDOException $e){  
    //Handle error  
}
```

The UPDATE Statement

```
try {  
    // Setup a one-off SQL statement and get a statement object  
    $stmt = $pdo->query( "UPDATE orders SET description='something special'  
        WHERE id=1" );  
  
    // Check results  
    echo "\nUpdate was ";  
    echo ($stmt->rowCount()) ? "successful!\n" : "NOT successful!\n";  
  
    // Get the record and see the update  
    $stmt = $pdo->query( 'SELECT * FROM orders where id=1' );  
  
    // Returns an associative array indexed by column name  
    $result = $stmt->fetch(PDO::FETCH_ASSOC);  
  
    // Output the result  
    print_r($result);  
} catch (PDOException $e){  
    //Handle error  
}
```

The INSERT INTO Statement

```
try {
    $sql = "INSERT INTO orders (date,status,amount,description,customer)
        VALUES ('" . time() . "', 'active','200','cool backpack','4')";
    $stmt = $pdo->query($sql);
    $id = $pdo->lastInsertId(); // Get last insert ID

    // Retrieve the update
    if ($id) {
        $stmt = $pdo->query( 'SELECT * FROM orders WHERE id = ' . $id );

        // Get the new entry by associative array
        $result = $stmt->fetch(PDO::FETCH_ASSOC);

        print_r( $result );
    } else {
        throw new Exception('Insert unsuccessful');
    }
} catch (Throwable $e){
    // Handle error
    // Respond to client
    echo $e->getMessage();
}
```

The DELETE Statement

```
try {  
    // Setup a one-off SQL statement and get a statement object  
    $id = 10 ;  
    $stmt = $pdo->query( "DELETE FROM orders WHERE id= $id " );  
  
    // Get the number of rows affected (should be 1)  
    $affected = $stmt->rowCount();  
  
    // Get the records and see the update  
    if (!$affected) throw new Exception('Delete unsuccessful');  
  
    // Get the records and see the deletion  
    $stmt = $pdo->query( 'SELECT * FROM orders' );  
  
    // Get the rows including the new insert as an associative array by column name  
    $result = $stmt->fetchAll(PDO::FETCH_ASSOC);  
    print_r( $result ); // Output the result  
} catch (Throwable $e){  
    // Handle error  
    // Respond to client  
    echo $e->getMessage();  
}
```


Database Operations

Prepared Statements

A prepared statement is an SQL statement that includes parameter markers for parameters. The statement is executed which compiles and stores itself on the database engine. A parameter for each marker is required.

Parameter markers are not available for SQL identifiers (i.e. table or columns), only values.

Benefits

- Performance: compiled once and reusable.
- Security: only parameters are exposed to injection.

Prepared Statements

Parameter Bindings

Binding to positional placeholders:

```
try {  
    // Get the connection instance  
    $pdo = new PDO('mysql:host=localhost;dbname=course', 'vagrant', 'vagrant');  
  
    // Set error mode attribute  
    $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);  
  
    // Setup a one-off SQL statement and get a statement object  
    $stmt = $pdo->prepare('INSERT INTO customers (firstname, lastname)  
        VALUES (?,?)');  
  
    // Hard coded input parameters  
    $fname = 'Mark';  
    $lname = 'Watney';  
  
    // Parameter bindings  
    // The second parameter is referenced so must be an identifier  
    $stmt->bindParam(1, $fname);  
    $stmt->bindParam(2, $lname);  
  
    // Execute the SQL statement  
    $stmt->execute();  
} catch (PDOException $e) {  
    //Handle error  
}
```

Prepared Statements

Parameter Bindings

Passing positional parameters as an array:

```
try {  
    // Get the connection instance  
    $pdo = new PDO('mysql:host=localhost;dbname=course', 'vagrant', 'vagrant');  
  
    // Set error mode attribute  
    $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);  
  
    // Setup a one-off SQL statement and get a statement object  
    $stmt = $pdo->prepare('INSERT INTO customers (firstname, lastname)  
        VALUES (?, ?)');  
  
    // Hard coded input parameters  
    $fname = 'Mark';  
    $lname = 'Watney';  
  
    // Execute the SQL statement and pass params  
    $stmt->execute([$fname, $lname]);  
} catch (PDOException $e) {  
    //Handle error  
}
```

Prepared Statements

Parameter Bindings

Bind to named parameters:

```
try {  
    // Get the connection instance  
    $pdo = new PDO('mysql:host=localhost;dbname=course','vagrant','vagrant');  
  
    // Set error mode attribute  
    $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);  
  
    // Prepare an SQL statement and get a statement object  
    $stmt = $pdo->prepare('INSERT INTO customers (firstname,lastname)  
        VALUES (:firstname, :lastname) ');  
  
    // Hard coded input parameters  
    $fname = 'Mark';  
    $lname = 'Watney';  
  
    // Parameter bindings  
    // The second parameter is referenced so must be an identifier  
    $stmt->bindParam(':firstname', $fname);  
    $stmt->bindParam(':lastname', $lname);  
  
    // Execute the SQL statement  
    $stmt->execute();  
} catch (PDOException $e){  
    //Handle error  
}
```

Prepared Statements

Parameter Bindings

Passing named parameters as an array:

```
try {  
    // Get the connection instance  
    $pdo = new PDO('mysql:host=localhost;dbname=phpcourse','vagrant','vagrant');  
  
    // Set error mode attribute  
    $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);  
  
    // Setup a one-off SQL statement and get a statement object  
    $stmt = $pdo->prepare( 'INSERT INTO customers (firstname,lastname)  
        VALUES (:first, :last)' );  
  
    // Hard coded input parameters  
    $fname = 'Mark';  
    $lname = 'Watney';  
  
    // Execute the SQL statement and pass params  
    $stmt->execute([':first' => $fname, ':last' => $lname]);  
} catch (PDOException $e){  
    //Handle error  
}
```

Stored Procedure

A stored procedure is an SQL statement wrapper that can include parameter markers for parameters. The statement is executed which compiles and stores itself on the database engine. Think of a stored procedure equivalent to a PHP function.

A Parameter for each marker is required.

Parameter markers are not available for SQL identifiers, only parameters.

Benefits

- Performance: compiled once and reusable.
- Security: only parameters are exposed to injection.

Stored Procedure Loading

This is a simple example using only a single statement, but multiple statements using bound parameters are possible within the BEGIN and END statements. Think of a stored procedure as an SQL equivalent to a PHP function.

The stored procedure could be initialized with PDO->exec(), or added as plain SQL to the engine as shown here.

Note: The SQL for this stored procedure is in /oop/sandbox/public/ModDB/storedProc.sql.

```
DROP PROCEDURE IF EXISTS course.newCustomer;
DELIMITER $
CREATE PROCEDURE course.newCustomer(
    p_firstname varchar(50),
    p_lastname varchar(50))
BEGIN
    insert into customers (firstname, lastname) values (p_firstname,p_lastname);
    -- other statements ...
END
$
DELIMITER ;
```


Stored Procedure

Positional Parameters

Calling with positional parameters:

Note: Remember to run the storedProc.sql at the mysql command line first.

```
try {  
    // Get the connection instance  
    $pdo = new PDO('mysql:host=localhost;dbname=course','vagrant','vagrant');  
  
    // Set error mode attribute  
    $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);  
  
    // Prepare an SQL statement and get a statement object  
    $stmt = $pdo->prepare('CALL newCustomer (?,?) ');  
  
    // Hard coded input parameters  
    $fname = 'Mark';  
    $lname = 'Watney';  
  
    // Execute the SQL statement  
    if ($stmt->execute([$fname, $lname])) {  
        echo "New user $fname $lname added";  
    }  
} catch (PDOException $e){  
    //Handle error  
}
```

Transactions

A transaction is an SQL statement wrapper that operates in isolation. All transaction statements must successfully complete prior to a commit, at which time the data becomes available to other processes. Transactions are **ACID** compliant.

The **ACID** acronym stands for

- **Atomicity:** Ensures all operations within the transaction wrapper are successful; otherwise, the transaction is aborted at the point of failure, and previous operations are rolled back to a state prior to the transaction.
- **Consistency:** Ensures database state change upon successful commit.
- **Isolation:** Enables independent and transparent operation.
- **Durability:** Ensures state persistence in case of failure.

Transactions

Transaction Code

A typical transaction code sequence:

```
try {  
    // Get the connection instance  
    $pdo = new PDO('mysql:host=localhost;dbname=course', 'vagrant', 'vagrant');  
  
    // Set error mode attribute  
    $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);  
  
    // Begin the transaction  
    $pdo->beginTransaction();  
  
    // Series of SQL statements, all of which have to succeed  
  
    // Commit success  
    $pdo->commit();  
} catch (PDOException $e) {  
    $pdo->rollBack(); // Rollback in case of failure  
    // log and communicate error  
}
```

Database Operation Labs

Lab: Prepared Statements

1. Create a prepared statement script.
2. Add a try/catch construct.
3. Add a new customer record binding the customer parameters.

Lab: Stored Procedure

1. Create a stored procedure script.
2. Add the SQL to the database.
3. Call the stored procedure with parameters.

Lab: Transaction

1. Create a transaction script.
2. Execute two SQL statements.
3. Handle any exceptions.

Module Summary

This module covered:

- Relations and SQL
- Data modeling
- PHP Data Objects (PDO)
- Database operations

Internet Communications

This module covers:

- Output control
- Email

Output Control

Output Control

Output control is a technique that captures application output, stores it on a server as it is generated, then, when processing is about finished, recovers the output and sends it to a client in a single response.

Output buffering facilitates setting response headers and creating cookies as dictated by the PHP script where needed without conflict caused by a prior echo or print statement.

PHP has a number of functions to facilitate buffering options.

Functions that start and end output buffering, and functions that get buffer contents and deal with what is left in the buffer.

PHP's Output buffering employs a nesting mechanism and indexes beginning at 1 and incremented as the function `ob_start()` is called. A current nesting level is retrievable and used where a specific nesting level is targeted.

Output Buffering

```
// Start buffering
ob_start ();

// Output of the echo commandss are buffered
echo '<h1>Hello Buffer</h1>';
echo '<p>Store this away in your memory</p>';

// This can execute anyplace
header('Contents: Something very cool for you');

// Okay, get the buffer contents, clean the buffer and send the contents
echo ob_get_clean();
```

Nested Buffer Example

```
// Top level buffer
ob_start();

echo 'some content in the first buffer';

// Nested buffer
ob_start();
echo 'some content in the second buffer';

// Get the nested buffer contents
$content = ob_get_contents();
ob_end_flush(); // Flush the nested buffer to the outer

// Get and clean from the outer buffer
$content = ob_get_clean();
echo $content;
```

HTTP Headers and Browser Cache Example

The PHP function `header()`, provides a mechanism for setting response headers. Response headers are the metadata to a response.

```
// Setting header examples
```

```
$current = time();
```

```
// The date/time after which the response is considered stale.
```

```
header('Expires: ' . gmdate ( 'D, d M Y H:i:s' , $current + 50000));
```

```
// Specifies directives that client should cache this response.
```

```
header('Pragma: cache');
```

```
// Freshness; max-age = Maximum age in seconds to be considered fresh.
```

```
header('Cache-Control: must-revalidate, max-age=0');
```

HTTP Headers and Browser Cache Considerations

HTTP headers are metadata that payload with a response body.

Browser caching refers to modern browsers that cache the last page response, which makes it possible to use this default caching behavior to help unload a server.

It is a good idea to employ these techniques if a page is relatively static for a certain period of time.

Provide an Etag header token associated with a page, and test for it in a subsequent request, or use the Last-Modified HTTP header to determine freshness of a requested page.

If a page is within the time specification set in the code, set a location header with 304 response and exit.

ETag Header Example

```
// If we have the key, and therefore an etag, and
// it is less the the stored expiration time
if (isset($_SERVER['HTTP_IF_NONE_MATCH'])
    && $_SERVER['HTTP_IF_NONE_MATCH'] < $_SESSION['expires']) {
    // We don't need to do anything except send a 'Not modified' header and exit
    header('Not Modified', true, 304);
    exit();
}

// No etag header, create one for this page
$current = time(); // Current time
$oneWeek = $current - 6.048e+2; // 1 week earlier in seconds
$duration = 2.628e+6; // 1 month in seconds
$etag = $current - $oneWeek;
$_SESSION['etag'] = $etag;
$expires = $current + $duration;
$_SESSION['expires'] = $expires;

header('Expires: ' . gmdate ("D, d M Y H:i:s", $expires) . " GMT");
header('Last-Modified: ' . gmdate ("D, d M Y H:i:s", $current) . " GMT");
header(" ETag : $etag ");
header("Pragma: cache");
header("Cache-Control: public, must-revalidate, max-age=0");
```


Super Global \$_SERVER

The \$_SERVER is a global array containing raw request header data, among other data.

Important keys include:

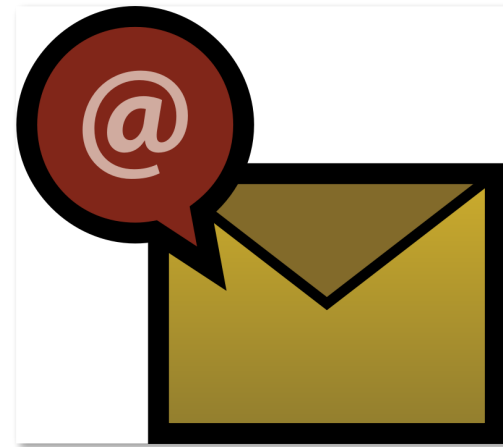
- **\$_SERVER["HTTP_USER_AGENT"]**: The user agent (browser) identification.
- **\$_SERVER["HTTP_HOST"]**: The host identification.
- **\$_SERVER["HTTP_CACHE_CONTROL"]**: A maximum page age setting.
- **\$_SERVER["HTTP_ACCEPT"]**: The page MIME type acceptable to the client.
- **\$_SERVER["HTTP_ACCEPT_ENCODING"]**: The compression encodings that the browser supports.
- **\$_SERVER["HTTP_ACCEPT_LANGUAGE"]**: The language acceptable to the client.
- **\$_SERVER["HTTP_IF_NONE_MATCH"]**: The etag header.

Email

Email

In this unit we will cover:

- mail() Function
- Headers
- Libraries



Email

PHP has a function [mail\(\)](#) that facilitates sending email. Configuration in the php.ini file for a mail transport agent (MTA) is required.

The mail function is elementary by design, which means other API™s are available to enhance the core email functionality, API™s like Zend\Mail.

```
$to = 'lets_bug_clark@zend.com';  
$subject = 'Test mail';  
$message = 'This is sent from a PHP script';  
$from = 'employee@zend.com';  
$cc = 'bozo@circus.com';  
$vers = 'X-Mailer: PHP/' . phpversion();  
$headers = "From: $from\r\nCC: $cc\r\n$vers";  
mail($to, $subject, $message, $headers);
```

Additional Email Headers

Any standard headers, as defined by [RFC 2822](#) section 3.6 (Field Definitions) are acceptable

Represents the fourth argument to the `mail()` function

Headers are the form of a single contiguous string

Each discreet header is separated from the next by means of CR-LF (i.e. `"\r\n"`)

// example of HTML email

```
$to = 'Clark <clark@zend.com>';  
$subject = 'About That Beer You Promised Me ...';  
$body = '<h1>About That Beer</h1><p>Blah</p><p>Blah</p><p>Blah</p>';  
$headers[] = 'MIME-Version: 1.0';  
$headers[] = 'Content-type: text/html; charset=iso-8859-1';  
$headers[] = 'From: Doug <doug@unlikeysource.com>';  
$headers[] = 'Cc: daryl@datashuttle.net';  
$headers[] = 'Bcc: cal@ecalevans.com';  
mail($to, $subject, $body, implode("\r\n", $headers));
```

Email Configuration

Configuration directives directly impacting PHP's email function include:

- **smtp:** Identifies the host server.
- **smtp_port:** Identifies the host server port.
- **sendmail_from:** Identifies which from email address used by PHP.
- **sendmail_path:** Identifies the sendmail application path, if used.

Email Libraries

More complex email operations are made easier by incorporating third party open source software

Most frameworks have an email component, including:

- **Zend Framework 3** [zend-mail](#)
- **Code Igniter** [Email Class](#)
- **CakePHP** [Cake\Mailer\Email](#)

In addition there are "stand-alone" libraries which you can use. Probably the best known example is **PHPMailer** [source code on github.com](#)

Module Summary

This module covered:

- Output control
- Email

Regular Expressions

This module covers:

- Regular expressions
- Minimum requirements
- Meta characters
- Character classes
- Quantifiers
- Greediness
- Pattern modifiers
- PHP Perl Compatible Regular Expressions (PCRE) functions

Regular Expressions

Regular Expressions

A string (referred to as a "pattern") is used to describe, parse or search another string.

Can cause performance degradation if used in massive loops. Consider using `ctype_*` or `str*` functions instead.

Used for:

- Validation of form data
- Intelligent search and replace
- Exploding a string based upon a pattern
- Extracting substrings

See: <http://php.net/manual/en/reference.pcre.pattern.syntax.php>

Minimum Requirements

Minimum Requirements

Minimum requirements include:

- PHP implementations must have a *delimiter*, which is the same character which identifies the start and end of a pattern
- Most commonly used character is a slash ("/")
- Also commonly used: "!" and "#"
- This syntax is also allowed: [*regex*]
but could get confused with the syntax for custom character classes (discussed shortly).
- *Meta Characters* as using inside the delimiters and have special meanings.

Meta Characters

Meta Characters

Dot, Escaping, Sub-Patterns

Metacharacter requirements include:

- Use the dot (".") to match any character.
- If you need to specifically look for a dot, you must first "escape" it using backslash ("\").
- Use the pipe symbol ("|") to indicate alternates.
- Use parentheses "(xxx)" to designate a sub-pattern.

Examples of Dot, Escaping, Sub-Patterns

`/Table.*\.php/`

Matches a string which contains "Table", followed by any character, ending with ".php"

`/<p>(.*?)</p>/`

Capture contents of `<p>xxx</p>` tags

`/<a.*?href=(\"|')(.*?)\"|').*?>/`

Capture contents of the "href" attributes in `<a>xxx` tags

Positioning

These characters are used to tell the analyzer where to start or stop

- **^**: Patterns begins with following character or group.
- **\$**: Patterns ends with preceding character or group.
- **\b**: Word boundary.
- **\A**: Absolute start.
- **\B**: Absolute end.

Positioning Examples

`/^[A-Z].*/`

Matches string which starts with letters A to Z

`/.*(jpg|png)$/`

Matches string which ends with "jpg" or "png"

`/\bERROR\b/`

Matches string which contains exactly the distinct word "ERROR"

Character Classes

Character Classes

Definition

Character classes are a collection of characters that are a selection of characters offered for match option where only one of the selection is used as a match token. Here are some examples:

- **[admpt]**: Matches one of the characters.
- **[a-z]**: Matches one lowercase letter in the alphabet range.
- **[A-Z]**: Matches one uppercase letter in the alphabet range.
- **[0-9]**: Matches one digit in the range between zero and nine.
- **[a-zA-Z0-9_]**: Matches one character from the three ranges, and an underscore character.
- **\w**: Shortcut for [a-zA-Z0-9_]
- **\d**: Shortcut for [0-9]

Remember: Only matches one of the presented characters from the character class.

Built-In

- **\d**: Digits.
- **\w**: "Word" characters: includes A-Z, a-z, 0-9 and "_"
- **\s**: White space.
- **\D**: Non-digits.
- **\W**: Non-word characters.
- **\S**: Non-white space.

Built-In Character Class Examples

`/^\w*$`

Matches string which only contains alpha numerics

`/^\d*$`

Matches string which only contains digits

Custom Character Classes

Use square brackets ("[]") to enclose custom classes. Use a dash ("-") to indicate a range of characters or numbers. Using the caret ("^") inside the square brackets indicates the opposite.

- **[A-Z]**: Uppercase alpha characters.
- **[a-z]**: Lowercase alpha characters.
- **[0-9]**: Digits 0 to 9
- **[A-Za-z0-9]**: Upper or lowercase alphanumeric characters.
- **[^A-Za-z]**: Anything which is **not** upper or lowercase letters

Custom Character Class Examples

```
/^[A-Za-z]*$/
```

Matches string which only contains only upper or lowercase letters

```
/^[^A-Za-z]*$/
```

Matches string which does not contain any upper or lower case letters.

Quantifiers

Quantifiers

Quantifiers quantify the number of single character token matches required.

Generic, imprecise, quantifiers include the following:

- ? Matches 0 or 1 items for the preceding meta-character or sub-pattern
(i.e. if you add "?" after a meta-character or sub-group, it makes that character or grouping optional).
- * Matches 0 or more items for the preceding meta-character or sub-pattern.
- + Matches 1 or more items for the preceding meta-character or sub-pattern.

Generic Imprecise Quantifier Examples

```
/<h1>.+</h1>/
```

Matches 1 or more characters between <h1> tags.

```
/^http(s)?\w*/i
```

Matches any string which starts with http or https and is followed by 0 or more alphanumeric characters.

Precision Quantifiers

Using curly braces ("{xxx}") allows you to precisely define how many of a certain type of character to expect.

- **{n}**: Exactly "n" characters.
- **{n,}**: At least "n" characters.
- **{,n}**: At most "n" characters.
- **{n,m}**: From "n" to "m" characters.

Precision Quantifier Examples

```
/^\d{3}-\d{2}-\d{4}$
```

Matches string a US Social Security Number

```
/([A-Z]\d[A-Z] \d[A-Z]\d)([A-Z]{2}\d{1,2} \d{1,2}[A-Z]{2})/i
```

Matches a UK or Canadian Postal Code

Greediness

Greediness Definition

By default the analyzer tries to match *all possible* characters that fit the pattern.

Adding a question mark "?" following the imprecise quantifiers, or by using the "U" pattern modifier, un-greedy the parser.

```
$test = '<p>Paragraph 1</p><p>Paragraph 2</p><p>Paragraph 3</p>';  
$pattern = '/<p>.*</p>/';  
preg_match($pattern, $test, $matches);  
echo $matches[0];  
// output: "<p>Paragraph 1</p><p>Paragraph 2</p><p>Paragraph 3</p>"
```

Non-Greedy Example

```
$test = '<p>Paragraph 1</p><p>Paragraph 2</p><p>Paragraph 3</p>';  
$pattern = '/<p>.*?</p>/';  
preg_match($pattern, $test, $matches);  
echo $matches[0];  
// output: "<p>Paragraph 1</p>"
```

Pattern modifiers

Pattern Modifiers

Modifiers occur *after* the end delimiter. You can stack up as many as you want. They affect the operation of the entire pattern. Here is a partial summary of some of the more important ones:

- **i**: Case insensitive.
- **m**: Informs the analyzer that the string contains multiple lines, and to treat it as a "single" line despite the presence of new line characters.
- **S**: Pre-analyze the pattern. Useful if you intend to use this pattern inside a massive loop.
- **s**: Causes the dot (".") to match all characters, including new lines.
- **U**: Forces "Un-Greedy" behavior.
- **u**: Pattern and subject string are treated as UTF-8

See: <http://php.net/manual/en/reference.pcre.pattern.modifiers.php>

Examples

```
/^[A-Z].*$i
```

Matches a string which starts with case insensitive A to Z

```
/à,>à,£à,°.*u
```

Matches a UTF-8 string

PHP Perl Compatible Regular Expressions (PCRE) Functions

PHP Perl Compatible Regular Expressions (PCRE) Functions

The currently maintained PHP extension for regular expressions is PCRE

The `ereg` family has been deprecated

Another set of regex functions is available in the `mb_string` extension

To process multi-byte regular expressions and strings, use the "u" modifier

Summary of PCRE Functions

`preg_filter()`

Perform a regular expression search and replace

`preg_grep()`

Return array entries that match the pattern

`preg_last_error()`

Returns the error code of the last PCRE regex execution

`preg_match`

Returns boolean after first match

`preg_match_all`

Returns all matches

Summary of PCRE Functions Continued

`preg_quote()`

Quote regular expression characters

`preg_replace_callback_array()`

Perform a regular expression search and replace using callbacks

`preg_replace_callback`

Perform a regular expression search and replace using a callback

`preg_replace()`

Perform a regular expression search and replace

`preg_split()`

Split string by a regular expression

See: <http://php.net/manual/en/ref.pcre.php>

preg_match*() Examples

Matches string a US Social Security Number

```
$input = '111-22-3333-5566';  
$pattern = '/^\d{3}-\d{2}-\d{4}$/';  
echo preg_match($pattern,$input) ? "MATCH" : "NO MATCH";
```

Matches words in a string

```
$test = 'To this,ha ha,I say tally ho. Ta ta. "Ha hah";  
$words = array();  
// This does not work:  
// $words = explode(' ', $test);  
  
// Try this instead:  
preg_match_all('/\w+?\b/', $test, $words);  
var_dump($words);  
  
// this also works:  
$words = preg_split('/[^\w]/i', $test, 0, PREG_SPLIT_NO_EMPTY);  
var_dump($words);
```

preg_replace_callback() Example

Pulls out the basename of a filename embedded in an error log

```
function returnBasename($matches) {  
    if (is_array($matches)) {  
        $result = basename($matches[0]);  
    } else {  
        $result = NULL;  
    }  
    return $result;  
}  
  
// here is the error message with directory info  
$test = 'Notice: Undefined offset: 1 in  
    /var/www/CodeArchive/application/files/php/basics/array_example.php on line 4';  
  
// Create regex  
$pattern = '|(\\w+)+\\w+\\.php|';  
  
// Print results w/ only the filename  
echo preg_replace_callback($pattern, 'returnBasename', $test);
```


Regular Expression Labs

Lab: Validate an Email Address

Use `preg_match()` to validate an email address

Module Summary

This module covered:

- Regular expression engine
- Minimum requirements
- Meta characters
- Character classes
- Quantifiers
- Greediness
- Pattern modifiers
- PHP PCRE functions

Composer

This module covers:

- Composer
- Important files
- The vendor directory
- Packagist

Composer

Ref: [Composer](#) is a dependency management utility for PHP available on a per project basis. Composer offers the following utility:

- Initial installation of dependent code libraries
- Updates to installed libraries
- A class autoloader for classes in the vendor directory

Dependent libraries are installed in a "vendor" directory inside the project thereby separating the dependent libraries from application code.

Composer is multi-platform and runs well on Linux, MacOS and Windows.

Two installations are offered:

- Locally: the composer executable is available to a specific project.
- Globally: the composer executable is available to all directories.

Important Files

Files that are installed as part of the local Composer install in the project root include:

- **composer.phar**: The main PHP executable.
- **composer.json**: The project specification JSON file.
- **composer.lock**: A file that maintains the correct installed dependency version.
- **vendor**: The directory in which dependent libraries are installed.
- **vendor/composer**: A directory containing the Composer autoloader.

Important Files

composer.phar

The composer.phar file is a file executed using the CLI version of PHP. It is normally run at a terminal prompt at the project root location. It can also integrate in most IDEs. Once installed globally, it can run in any directory.

```
vagrant@php-training:~$ vagrant@php-training:~$ php composer.phar install|update|self-update
```

To run at a terminal prompt locally within the project root:

```
vagrant@php-training:~$ vagrant@php-training:~$ composer install|update|self-update
```

To run at a terminal prompt after global installation:

composer.phar

Important Command Options

- **Install:** Initial install of specified dependent libraries.
- **Update:** Update of installed dependent libraries.
- **Self-update:** Updates the composer executable. This is prompted for if the Composer installation itself is older than a certain length of time—usually around a month.
- **dump-autoload:** Rebuild the autoloader updating for additions to the autoload key in the composer.json file.

composer.json

The composer.json file is the specification file for dependent libraries. It is a JSON file containing keys that are used by Composer. For the purposes of our class, the important keys are:

- **Name:** The name of the application.
- **Description:** A short description of the application.
- **License:** The application license designation.
- **Version:** The version.
- **Keywords:** Keywords used to categorize the application.
- **Homepage:** The application URL.
- **Require:** The application dependency specification. It's the key detailed next.

Other top level keys exist. [Here is a good reference for commands and keys.](#)

This file is JSON and strict formatting is required.

composer.json (cont.)

Require Key

The require key is where the dependency specification lives. Here is an example:

```
"require": {  
  "php": ">=5.3.3",  
  "zendframework/zendframework": "2.3.*"  
}
```

Subkeys denote the dependency name.

composer.json (cont.)

Require Key

The [version specification](#) includes operators `>`, `=` and `*`.

- `">":` Denotes a version greater than the specified version.
- `">=":` Denotes a version greater than or equal to the version specification.
- `"*":` Is a wild card in the version specification.
- `"~":` Denotes range, such that `~1.5` is equivalent to `>=1.5` and `<=2.0`. It is useful for projects respecting semantic versioning.
- `"^":` Denotes range as well, such that `^1.5.3` is equivalent to `>=1.5.3 <2.0.0`. Similar to `"~,"` but it sticks closer to semantic versioning, and will always allow non-breaking updates.
- A missing operator denotes a specific version.

Best Practice: Be careful using unbounded version constraints like `"2.*"`, as this could possibly break backward compatibility. Be specific with versioning constraints until test verifies compatibility.

composer.json (cont.)

Expanded composer.json File

Here is a larger sampling of a completed composer.json specification for the order application.

```
{
  "name": "Order Application",
  "description": "Order application for customer orders",
  "keywords": [
    "Order App"
  ],
  "homepage": "http://orderapp/",
  "require": {
    "php": ">=5.5",
    "guzzlehttp/guzzle": "~6.0"
  }
}
```

Composer Labs

Lab: Composer with OrderApp

1. Add composer to the OrderApp project.
2. Edit the composer.json file to match the JSON shown in the Order Application sample in the previous slide.
3. Execute Composer and install the specified dependencies.

Lab: Composer with OrderApp

After you work on the exercise we'll discuss the solution as a group.

The Vendor Directory

The Vendor Directory

A directory that Composer initializes and installs the dependent libraries, along with a directory for itself.

Note: Some libraries may include specific configuration for customizing the library functionality. Check the README.md files for the library installation for instruction.

Best Practices

- Don't edit in the "vendor" directory. Update via the composer.json file and execute the update command.
- Don't include the "vendor" directory in a version control system when pushing updates to remote. Reference [this](#).

Packagist

The Composer site maintains API documentation . The documentation contains all information about commands.

[Packagist](#) is the official Composer repository for PHP packages. The repository allows for browsing all the available packages.

Packages are listed by latest releases, by new packages, and by most popular.

Module Summary

This module included:

- Composer, what it is and how to install it.
- Important Composer files
- Important Composer commands
- The vendor directory
- Composer API documentation
- Packagist

Web Services

This module covers:

- Data formats
- Web services
- Streams Architecture

Data Formats

XML

Responses to web service requests are formatted in a number of different ways. Some of the most common include:

[Extensible Markup Language \(XML\)](#): An older tag-based format still in use today.

```
<?xml version = "1.0" encoding = "UTF-8" ?>
<produce xmlns:ea = "test">
  <vegetables>
    <vegetable unit = "pound">
      <name> tomatoes </name>
      <price> 2.99 </price>
    </vegetable>
  </vegetables>
</produce>
```

JSON

[JavaScript Object Notation \(JSON\)](#): A native JavaScript object. The most popular data format today. Popular because it is native JavaScript.

```
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 25,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  }
}
```

YAML

[YAML Ain't Markup Language \(YAML\)](#): A simple data serialization format. Note: The YAML extension requires the library Libyaml installation and it is not part of the PHP core. As a result, the course will not demo YAML parsing, but it is available via the Libyaml library if installed and compiled.

```
monolog:
  handlers:
    main:
      type: fingers_crossed
      action_level: error
      handler: nested
    nested:
      type: stream
      path: "%kernel.logs_dir%/%kernel.environment%.log"
      level: debug
  console:
    type: console
```


Parsing APIs

PHP has a number of parsing extensions which present APIs for data formats used in web services. If XML, YAML, or JSON is the data payload of a web service call, then parsing that payload become necessary. Here are the important parsing libraries for each:

- **DOM:** A large and comprehensive object-oriented XML parsing library capable of doing anything with XML.
- **SimpleXML:** A limited object-oriented XML parsing library intended as a simple API framework for consuming XML.
- **XMLReader:** A object-oriented XML stream-based pull parser library intended for parsing XML as a stream of incoming data.
- **XMLWriter:** An object-oriented XML stream-base push parser intended for parsing XML as a stream of outgoing data.
- **YAML:** A simple parsing library for YAML.
- **JSON:** A simple parsing library for JSON.

SimpleXML

Generally considered the best library for consuming XML.

SimpleXML can take an XML file, an XML string, or in some cases, another object, and parse it into an instance of the SimpleXMLElement class. This provides an object-oriented interface to the consumed XML. All of the import techniques shown below return an instance of SimpleXMLElement. Once instantiated, the SimpleXMLElement object and associated methods allow for obtaining parts of the XML as either other SimpleXMLElement objects, or as arrays.

SimpleXMLElement::__construct()

Accepts an XML parameter string.

simplexml_load_file()

A function that accepts a file path containing XML.

simplexml_load_string()

A function that accepts a string of XML.

SimpleXMLElement Example:

```
// A simpleXML load file example
$xml = simplexml_load_file( 'produce.xml' );

// Get the vegies
$vegies = $xml->vegetables;

// Get the first vegie using array notation
$vegie = $vegies->vegetable[0]->name;

// Output item data
foreach ( $vegies as $node ) {
    echo "Content: " . $node->vegetable->name . "\n" ;
}

// Output XML from the SimpleXMLElement object
echo $xml->asXML();

// Output to an xml file
$xml->asXML( 'newproduce.xml' );
```

JSON Parsing

JSON parsing is very simple. Generally, and in most cases, two functions are all that's required, one to encode, the other to decode. They are:

`json_encode()`

Encodes all data types except resource.

`json_decode()`

Decodes to either standard class object or array.

```
// Get the JSON
$json = file_get_contents('user.json');

// Decode as a standard class object
$userObject = json_decode( $json );
var_dump( $userObject ). PHP_EOL;

// Decode as an associated array
$userArray = json_decode( $json , true );
var_dump( $userArray );
```

YAML Parsing

Much like JSON, the YAML extension consists of functions which read or write YAML. They are:

`yaml_parse()`

Parses an YAML string into an array

`yaml_emit()`

Produces a YAML string from any native PHP data type except for resource

Variations include `yaml_parse_file()` and `yaml_emit_file()`

IMPORTANT: the `yaml` extension is **not** installed or enabled by default on most PHP installations!

YAML Parsing Example

```
$yaml = <<<EOT
imports:
  - { resource: config.yml }
monolog:
  handlers:
    main:
      type:      fingers_crossed
      action_level: error
      handler:    nested
    nested:
      type: stream
      path: "%kernel.logs_dir%/%kernel.environment%.log"
      level: debug
    console:
      type: console
EOT;
$yaml = yaml_parse($yaml);
echo yaml_emit($yaml);
```

Web Services

Web Services

Web services are information services provided by service providers. Some services are free, some at a cost, some require registration and Application Programming Interface (API) keys, some do not.

Web services amount to making a request to the service via an API, and receiving data back in a response.

All types of web services API™s exist, such as:

- [Google Maps APIs](#)
- [Ebay Developers Program](#)
- [Amazon Web Services](#)
- [Reddit Developers API](#)

In addition, there are indexing sites that index available services, like the [Programmable Web API Directory](#).

Using Web Services

To use a web service, make a server-side request to a service API using the request parameters defined in the service API. A number of tools are available to request from a service API. Here are a few:

- [Zend Framework Zend\HttpClient](#): A powerful modular HTTP client.
- [Client URL Request Library \(cURL\)](#): A terminal-based request library.
- [Guzzle PHP HTTP Client](#): A powerful asynchronous-capable request client.
- [Httpful Client](#): The REST friendly PHP HTTP client.

In the case of cURL, there are APIs for interface to the cURL library. One such library is PHP's own [cURL API](#).

In addition to the above listed clients, classes are defined in most frameworks that are interfaced to work with the chosen framework code.

Requests are made from these clients and responses returned in a particular data format.

Web Service Requests

Web service requests are requests for data from a third party server, the returned data from which is then parsed and used as needed.

There are multiple techniques to request data, and PHP APIs to facilitate. Of the PHP APIs available, one is most common, while the other doesn't require an API at all, rather a third part HTTP request client or just a URL request.

The two most common web service requests today are:

- [SOAP](#): Simple Object Access Protocol: A messaging protocol.
- REST: Representational State Transfer: A software architecture that permits requests over, typically, HTTP/HTTPS protocol using the HTTP verbs of GET, POST, PUT, PATCH AND DELETE.

Styles of Services

There are two primary styles of web services:

XML-RPC eXtensible Markup Language - Remote Procedure Call
REST Representational State Transfer

SOAP (Simple Object Access Protocol) is was developed in parallel with XML-RPC

XML-RPC

Provides a mechanism which supports remote procedure calls using HTTP as a transport and XML as the data format

Created in 1998 by Dave Winer of UserLand Software and Microsoft

PHP provides supports for both consuming remote XML-RPC services and building new XML-RPC servers

XML-RPC services target actions and operations which are all procedural

Zend Framework provides two classes to handle these:

- `Zend\XmlRpc\Server`
- `Zend\XmlRpc\Client`

To install: add `zendframework/zend-xmlrpc` to Composer requirements

SOAP

SOAP services are relatively more complex than RESTful services, and use XML as the data format.

PHP's SOAP API defines client and server objects to consume and serve SOAP services respectively.

SOAP Client and Server

A SOAP client instance is required to request data from a third party SOAP server, and SOAP server instance is required to service an outside data request.

A SOAP client and server use an XML document called Web Service Description Language (WSDL), to describe the service, identifying functions and properties available.

A SOAP client makes an initial request for a WSDL (which is XML) and parses it into a SoapClient instance. All subsequent requests are made via the SoapClient instance using the functions and properties provided.

A SOAP server can provide a WSDL file to a SOAP client identifying the functions and properties available, or no WSDL in certain cases, and respond to requests.

WSDL Example

A simple WSDL file

```
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
xmlns:xs = http://www.w3.org/2001/XMLSchema
xmlns:soap = "http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:soap12 = "http://schemas.xmlsoap.org/wsdl/soap12/"
xmlns:tns = "http://footballpool.dataaccess.eu"
name = "Info" targetNamespace = "http://footballpool.dataaccess.eu">
  <types>
    <xs:schema elementFormDefault = "qualified"
      targetNamespace = "http://footballpool.dataaccess.eu">
      <xs:complexType name = "tPlayerNames">
        ...
      </types>
    <operation name = "AllPlayerNames">
      <documentation> Returns an array with the id, name, country and
        flag reference of all players </documentation>
      <input message = "tns:AllPlayerNamesSoapRequest"/>
      <output message = "tns:AllPlayerNamesSoapResponse"/>
    </operation>
    ...
  </definitions>
```

A SOAP Client Example:

Note: SOAP clients and servers should be executed in a try/catch block due to the availability of a SoapFault exception.

```
try {  
    // Get a soap client instance passing WSDL URL  
    $client = new SoapClient("http://footballpool.dataaccess.eu/data/info.wso?wsdl");  
  
    // Make the request. Returns a standard class object  
    $result = $client->TopGoalScorers(['iTopN' => 20]);  
  
    // $result contains the result of the traversed object structure  
    var_dump($result->TopGoalScorersResult->tTopGoalScorer);  
} catch (SoapFault $e){  
    //Handle error  
}
```


A SOAP Server Example:

Note: A SOAP server is not a stand alone web server, and requires an HTTP server, like the Apache web server.

```
class MySoapService {
    public function sayHello() {
        return "Hello" ;
    }
}
try {
    /*Instantiate a SOAP server instance with no WSDL
    Here we are specifying a virtual host.*/
    $server = new SoapServer(null, ['uri' => 'http://url.of.soapserver/']);
    // Set the service class
    $server->setClass('MySoapService');
    // Start the new server
    $server->handle();
} catch ( SoapFault $e ){
    // Handle error
}
```

RESTful Web Services

RESTful services are the most popular way to consume and deliver web services today. They can be a simple URL request from the PHP function `file_get_contents()`, or a client request API. RESTful services:

- Directly use the HTTP protocol.
- Do not perform the additional round trip to a server to get a services definition WSDL “like a SOAP service.”
- Are not limited to any particular data type.

HTTP Verbs

HTTP verbs and provide meaningful methods for CRUD operations. REST uses HTTP built-in methods:

- **POST:** Create a data record.
- **GET:** Return a collection or data list of records.
- **PUT:** Update/Replace a data record.
- **PATCH:** Update/Modify a data record.
- **DELETE:** Delete a data record.



A web service is required to process requests for the HTTP verbs. For an easy introduction to RESTful API™s check out the Zend Framework apigility API builder .

file_get_contents() REST Example

```
// Make a request for JSON
$url = 'https://api.unlikelysource.com/api?city=Rochester&country=US';
$response = file_get_contents($url);
var_dump(json_decode($result));
```

GuzzleHttpClient REST Example

```
require __DIR__ . '/../vendor/autoload.php'; // Get the Composer autoloader

$client = new GuzzleHttp\Client(); // Get a client instance

// Make the request
$url = 'https://api.unlikelysource.com/api';
$response = $client->request('GET', $url,
    ['query' => [
        'city' => 'Rochester',
        'country' => 'US']
    ]
);

// Test for return status
if ( $response->getStatusCode() === 200 ) {
    // Output the JSON directly
    echo $response->getBody();
    // Output a PHP array
    print_r( json_decode ( $response->getBody() ) );
}
```

Data Format Labs

Lab: REST

1. Make a RESTful web service request to the post code lookup API using `file_get_contents()`, and/or optionally an HTTP client like `GuzzleHttpClient`.
2. Here are the parameters:
 - **URL:** `https://api.unlikelysource.com/api`.
 - **Data payload:** JSON.
 - **Server variables:**
 - **City:** City of your choice.
 - **Country:** ISO2 country code of your choice.

Streams

Streams Architecture

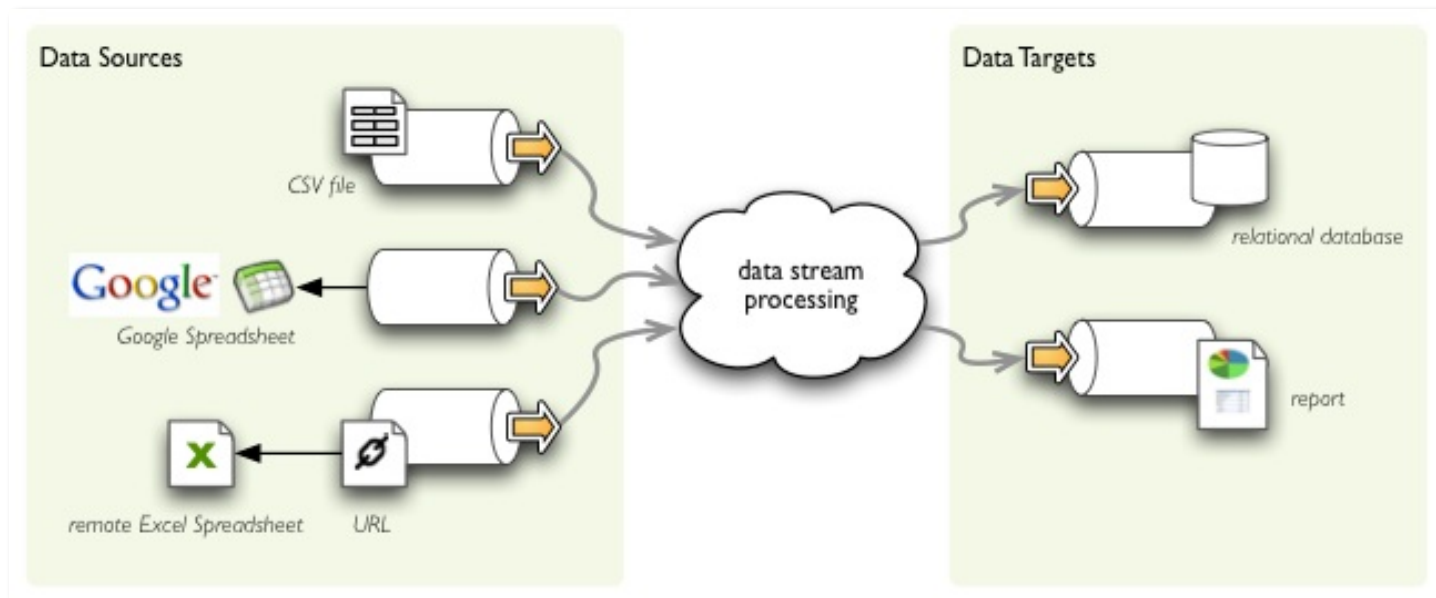
Streams are a resource and a PHP data type. Stream resources are used to read and/or write data to a data store including:

- Files
- Variables
- URLs
- Databases
- Memory

Streams Architecture

Streams are data pipelines to the resource. Writes and reads constitute two pipelines, one for each. Think of streams as the linear input/output of an application layer.

The PHP streams API includes functions that act on linear data streams, including tailoring stream protocols, compression, and filtering operations, etc.



Components

Components of data streams include:

- **Protocols and Wrappers:** http://, php://, ftp://, file://, zlib://, glob://, phar://, etc.
- **Sockets:** The connection. Think of them as a connection plug.
- **Filters:** Filtering of data as it is being read from or written to a stream resource.
- **Contexts:** A set of parameters and wrapper options which modify or enhance stream behavior.
- **Metadata:** Data about the stream including set parameters and options.

Custom Wrapper

Note: This class is less full method functionality for brevity.

```
class DataStream {
    public $position, $data;
    public function stream_open( $path, $mode ) {
        $this->data = parse_url($path)['host'];
        $this->position = 0;
        return true;
    }

    public function stream_write( $input ) {
        global $data; // Store in the global variable
        $length = strlen($input);
        $data = substr($input, 0, $this->position) . $input .
            substr($input, $this->position += $length);
        return $length;
    }
}
```

Custom Wrapper Runner

```
// Register the new stream class
stream_wrapper_register('mystream', 'DataStream');

// Initialize the data store variable to stream to
$data = " ";

// Open the stream resource
$fh = fopen('mystream://data', 'r+');

// Write to the stream resource
fwrite($fh, 'some new stuff');

// Close the resource
fclose( $fh );
```

Socket Example

```
// Get the resource
$fh = stream_socket_client('tcp://www.example.com:80', $errno, $errstr, 30);

// Check
if (!$fh) {
    // Output error data on failure
    echo "$errstr ( $errno )<br />\n";
} else {
    // Write to the stream
    fwrite($fh , "GET / HTTP/1.0\r\nHost: www.example.com\r\nAccept: */*\r\n\r\n");

    // Read from the stream
    while (!feof($fh)) echo fgets($fh, 1024);

    // Close the resource
    fclose( $fh );
}
```

Streams Lab

Lab: Custom Wrapper

1. Create a stream class with `stream_open` and `stream_write` methods.
2. Initialize the new stream as a resource and test.
3. Write something to the new resource.

Module Summary

This module covered:

- Data formats
- Web services
- Streams

Final Bindings

This module covers:

- PHP Unit, Documentor, and security
- PHP configuration
- Language updates

PHP Unit, Documentor, and Security

This unit covers:

- PHP Unit
- PHP Documentor
- Security Overview

PHP Unit

Software unit testing is test code that tests application code.

A full understanding of unit testing and test driven design is beyond the scope of our class, but references are provided along with a simple example.

Unit testing provides the following benefits:

- Identifies code problems early prior to deployment.
- Facilitates code refactoring helping to identify if refactoring causes other problems.
- Facilitates testing in small units from the bottom up and making integration testing easier.
- Can help with code understanding by demonstrating expected outcomes.
- Can drive the design of software if employed first, commonly referred to as test-driven design. Each test unit can specify functionality that is required.

See: <https://phpunit.de/>

PHP Unit Example

```
namespace src\ModFinalBindings\test;
use OrderApp\Core\Service\Services;
use OrderApp\Core\Db\ {Db, AbstractModel};
use PHPUnit\Framework\TestCase;
class TestDomain extends TestCase {
    public function setUp() : void {
        define('BASE', realpath(__DIR__ . '/../../../orderapp'));
    }
    public function testDomainHasServices() {
        // Get the service instance
        $serviceLocator = Services::getInstance();
        // Make an instance of domain
        $domain = $serviceLocator->getDomain();
        // Ensure the services dependency is available
        $services = null;
        $services = $domain->getServices();
        $this->assertNotEmpty($services);
        $this->assertInstanceOf(Services::class, $services, 'Invalid instance');
    }
}
```

PHP Documentor

PHP Documentor is a documentation creator and parser. It is capable of creating API documentation based on doc blocks placed in source code. Tags are used to specify parts of the code and are parsed by the documentor.

```
class InArray implements ValidatorInterface
{
    public $values = [];

    public function validate(array $value = null)
    {
        if ($this->values && in_array($value, $this->values)) {
            return true;
        }
        return false;
    }
}
```

See: <https://www.phpdoc.org/>

Web Security

Web security is a branch of information security that includes multiple techniques to protect information available from web-facing applications.

It is VERY important to learn how to write secure code and prevent vulnerabilities in applications.

Web security is limited in this class due to the availability of an in-depth security training curriculum provided by Zend.

The big picture of web security involves two primary parts:

- Filtering and Validating input
- Escaping output

Web Security: Validating Input

```
// Simulate a POST request
$_POST = ['firstname' => 'Mark',
    // try switching these 2 assignments an re-run
    'lastname' => 'Watney',
    'occupation' => 'martian',
    // 'lastname' => '<bad>Watney</bad>',
    // 'occupation' => 'earthling',
    'education' => 'Zend PHP II'];

// Build white lists
$occ = ['martian', 'developer'];
$ed = ['Zend PHP I', 'Zend PHP II', 'Zend PHP III'];

// Validators
$alpha = function ($input) { return ctype_alpha($input); };
$list = function ($input, $array) { return in_array($input, $array); };
$run = ['firstname' => ['validator' => $alpha],
    'lastname' => ['validator' => $alpha],
    'occupation' => ['validator' => $list, 'data' => $occ],
    'education' => ['validator' => $list, 'data' => $ed]];
```


Web Security: Validating Input Continued

```
// run the validation
$expected = count($_POST); // total fields
$valid = 0;
foreach ($run as $field => $callback) {
    if (isset($callback['data'])) {
        $valid += $callback['validator']($_POST[$field], $callback['data']);
    } else {
        $valid += $callback['validator']($_POST[$field]);
    }
}
if($valid == $expected) {
    // Input good, use it
    echo 'Good to go!' . PHP_EOL;
    $goodtogo = [$_POST['firstname'], $_POST['lastname'],
        $_POST['occupation'], $_POST['education']];
    foreach ($goodtogo as $item) echo htmlspecialchars($item) . PHP_EOL;
} else {
    echo 'Input invalid';
}
```

Web Security: Filtering Input

```
// Simulate a POST request
$_POST = [
    'firstname' => 'Mark',
    'lastname'  => 'Watney ',
    'occupation'=> ' martian ',
    'education' => '<bogus>Zend PHP II</bogus>'
];

// Define Filters
$trim = function ($input) { return trim($input); };
$tags = function ($input) { return strip_tags($input); };

// assign filters
$filters = [
    'firstname' => [$trim, $tags],
    'lastname'  => [$trim, $tags],
    'occupation'=> [$trim, $tags],
    'education' => [$trim, $tags]
];
```

Web Security: Filtering Input Continued

```
// insert validation code here
$valid = TRUE;
if ($valid) {
    // Input is valid: now perform filtering
    $goodtogo = [];
    foreach ($filters as $field => $run) {
        $item = $_POST[$field];
        // run $item through 1 or more filters
        foreach ($run as $callback) $item = $callback($item);
        // assign filtered item to the final array
        $goodtogo[$field] = $item;
    }
    print_r(htmlspecialchars($goodtogo));
} else {
    echo 'Input invalid';
}
```

Web Security: Escaping Output

```
// Input good, use it
$goodtogo = [
    htmlspecialchars($_POST['firstname']),
    htmlspecialchars($_POST['lastname']),
    htmlspecialchars($_POST['occupation']),
    htmlspecialchars($_POST['education'])
];

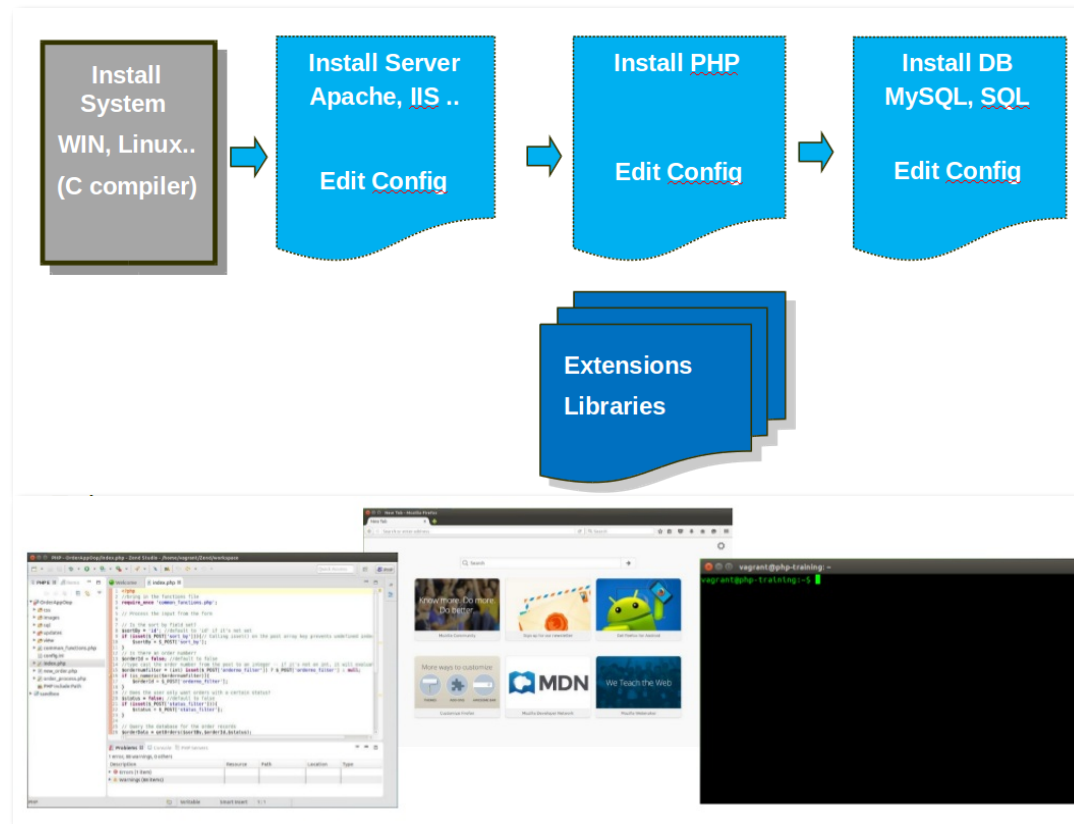
print_r($goodtogo);
```

PHP Configuration

This unit covers:

- Setting up a PHP environment
- What PHP configuration directives are available
- Where directives are set
- How to change a directive value at runtime
- Where to find the main php.ini configuration files in the virtual machine
- How to see the current configuration

PHP Development Environment



Directives

There are a number of configuration directives found in the php.ini file. A comprehensive definition of each directive is found on the PHP manual site located: <http://php.net/manual/en/ini.list.php>

Looking over this site reveals a table of:

- Directive links
- Default values
- Changable modes
- Changelog

Changeable Directives

Each directive links to manual pages detailing the directive. Here are the changable modes:

Can Set Using ...	PHP_INI_SYSTEM	PHP_INI_PERDIR	PHP_INI_USER	PHP_INI_ALL
ini_set()			X	X
Windows Registry			X	X
.user.ini		X	X	X
.htaccess		X		X
Web Server Conf	X	X	X	X
php.ini	X	X	X	X



NOTE: *Web Server Conf* file will vary depending on the web server and OS. For example, in Ubuntu Linux, if you use the pre-compiled binaries, the main configuration will be under `/etc/apache2/apache2.conf`. However, in CentOS linux, the main configuration defaults to `/etc/httpd/conf/httpd.conf`. The *nginx* configuration file most often default si `/etc/nginx/nginx.conf`.

Changing Directives

Changing directive values at runtime is accomplished with the `ini_set()` function. Directives that have specific setting requirements are found in the `php.ini` file under the Quick Reference heading. The current configuration settings are obtainable as follows:

- For web server: Execute the function `phpinfo()` in a script
- For PHP CLI at a terminal: `php -i`

phpinfo() output:

PHP Version 7.0.2-4+deb.sury.org~trusty+1 	
System	Linux php-training 3.19.0-25-generic #26~14.04.1-Ubuntu SMP Fri Jul 24 21:16:20 UTC 2015 x86_64
Server API	Apache 2.0 Handler
Virtual Directory Support	disabled
Configuration File (php.ini) Path	/etc/php/7.0/apache2
Loaded Configuration File	/etc/php/7.0/apache2/php.ini
Scan this dir for additional .ini files	/etc/php/7.0/apache2/conf.d
Additional .ini files parsed	/etc/php/7.0/apache2/conf.d/20-json.ini, /etc/php/7.0/apache2/conf.d/20-mysql.ini, /etc/php/7.0/apache2/conf.d/20-opcache.ini, /etc/php/7.0/apache2/conf.d/20-pdo_mysql.ini, /etc/php/7.0/apache2/conf.d/20-readline.ini
PHP API	20151012
PHP Extension	20151012
Zend Extension	320151012
Zend Extension Build	API320151012.NTS
PHP Extension Build	API20151012.NTS
Debug Build	no
Thread Safety	disabled
Zend Signal Handling	disabled
Zend Memory Manager	enabled
Zend Multibyte Support	provided by mbstring
IPv6 Support	enabled
DTrace Support	enabled
Registered PHP Streams	https, ftps, compress.zlib, php, file, glob, data, http, ftp, phar, zip
Registered Stream Socket Transports	tcp, udp, unix, udg, ssl, tls, tlsv1.0, tlsv1.1, tlsv1.2
Registered Stream Filters	zlib.*, convert.iconv.*, string.rot13, string.toupper, string.tolower, string.strip_tags, convert.*, consumed, dechunk
This program makes use of the Zend Scripting Language Engine: Zend Engine v3.0.0, Copyright (c) 1998-2015 Zend Technologies with Zend OPcache v7.0.6-dev, Copyright (c) 1999-2015, by Zend Technologies	
	

"php -i" output:

```
vagrant@php-training: ~/Zend/workspace

vagrant@php-training:~/Zend/workspace$ php -i
phpinfo()
PHP Version => 7.0.2-4+deb.sury.org-trusty+1

System => Linux php-training 3.19.0-25-generic #26~14.04.1-Ubuntu SMP Fri Jul 24 21:16:20 UTC 2015 x86_64
Server API => Command Line Interface
Virtual Directory Support => disabled
Configuration File (php.ini) Path => /etc/php/7.0/cli
Loaded Configuration File => /etc/php/7.0/cli/php.ini
Scan this dir for additional .ini files => /etc/php/7.0/cli/conf.d
Additional .ini files parsed => /etc/php/7.0/cli/conf.d/20-json.ini,
/etc/php/7.0/cli/conf.d/20-mysql.ini,
/etc/php/7.0/cli/conf.d/20-opcache.ini,
/etc/php/7.0/cli/conf.d/20-pdo_mysql.ini,
/etc/php/7.0/cli/conf.d/20-readline.ini

PHP API => 20151012
PHP Extension => 20151012
Zend Extension => 320151012
Zend Extension Build => API320151012,NTS
PHP Extension Build => API20151012,NTS
Debug Build => no
Thread Safety => disabled
Zend Signal Handling => disabled
Zend Memory Manager => enabled
Zend Multibyte Support => provided by mbstring
IPv6 Support => enabled
DTrace Support => enabled

Registered PHP Streams => https, ftps, compress.zlib, php, file, glob, data, http, ftp, phar, zip
Registered Stream Socket Transports => tcp, udp, unix, udg, ssl, tls, tlsv1.0, tlsv1.1, tlsv1.2
Registered Stream Filters => zlib.*, convert.iconv.*, string.rot13, string.toupper, string.tolower, string.strip_tags, convert.*, consumed,
dechunk

This program makes use of the Zend Scripting Language Engine:
Zend Engine v3.0.0, Copyright (c) 1998-2015 Zend Technologies
    with Zend OPcache v7.0.6-dev, Copyright (c) 1999-2015, by Zend Technologies

Configuration

bcmath
```

Common Configuration Directives

- allow_url_include
- date.timezone
- display_errors
- display_startup_errors
- error_log
- error_reporting
- file_uploads
- include_path
- session.name
- short_open_tag
- upload_max_filesize

Environment Configuration Considerations

Environment configuration involves edits to a number of files for each technology.

Be aware that configuration is adjusted for a target environment. If you employ staging, production and testing servers, or the like, each will have configuration settings specific to the intended environment. This is critical for security.

Here are the path locations in the Vagrant VM:

- **Apache web server:** /etc/apache2/apache2.conf.
- **Virtual hosts:** /etc/apache2/sites-available/<host>.conf.
- **Mysql:** /etc/mysql/my.cnf.
- **PHP Apache module:** /etc/php/<version>/apache2/php.ini.
- **PHP CLI:** /etc/php/<version>/cli/php.ini.

Language Updates

This unit covers:

- PHP 7.2 Updates
- PHP 7.3 Updates
- PHP 7.4 Updates
- PHP 8.0 Updates

PHP 7.2

Array Destructuring with List()

The list() construct now allows keys.

```
$array = [1 => 2, 2 => 4, 'eight' => 8];  
list(1 => $oneInt, 2 => $twoInt, 'eight' => $threeInt) = $array;  
echo $oneInt . PHP_EOL . $twoInt . PHP_EOL . $threeInt;
```

PHP 7.2

Invalid Strings in Arithmetic Warning

Strings can be used in a math operation when the PHP parser can type juggle the string in an attempt to execute the math operation. However if an invalid string is passed and the type juggler ignores it, no warning is emitted until now. New behavior of the type juggler will emit a warning on invalid strings.

```
$apples = 5;  
$oranges = '10';  
echo $apples + $oranges . PHP_EOL; // No warning  
  
$oranges = '10 oranges';  
// PHP Notice: A non well formed numeric value encountered in ...  
echo $apples + $oranges;
```


PHP 7.2

Standardizing on Negative String Offsets

In most PHP functions, providing a negative value as string offset means 'n positions counted backwards from the end of the string'. This mechanism is widely used but, unfortunately, these negative values are not supported everywhere this would make sense. So, as PHP developers cannot easily know whether a given string functions accepts negative values or not, they regularly need to refer to the documentation. If it does not, they need to insert a call to the `substr()` function, making their code less readable and slower.

An obvious example is `strrpos()` accepting negative offsets, while `strpos()` does not. The same with `substr_count()` rejecting negative offset or length, when `substr()` accepts them.

PHP 7.2

Void Return Type

Added to the strict typing for PHP, the "void" return type is now supported.

```
function should_return_nothing(): void {  
    return 1; // Fatal error: A void function must not return a value  
}
```

PHP 7.2

Object Type Hint

A new type, `object`, has been introduced that can be used for parameter and return typing of any objects.

```
function test(object $obj) : object
{
    return $obj;
}

var_dump(test(new stdClass()));
var_dump(test(new class() {}));
var_dump(test(new DateTime()));
/* Output:
object(stdClass)#1 (0) {}
object(class@anonymous)#1 (0) {}
object(DateTime)#1 (3) { ... }
*/
```

PHP 7.2

Parameter Type Widening

Parameter types from overridden methods and from interface implementations may now be omitted.

```
interface A
{
    public function test(array $input);
}
class B implements A
{
    // type omitted for $input
    public function test($input)
    {
        return 'You requested ' . htmlspecialchars($input);
    }
}
$b = new B();
echo $b->test('something');
// PHP 7.1 and below: "Fatal error: Declaration of B::test($input) must be compatible with A ..."
// PHP 7.2 and above: "You requested something"
```

PHP 7.2

Grouped Namespace Trailing Comma

A trailing comma can now be added to the group-use syntax introduced in PHP 7.0.

```
namespace Manage\Service;

use Interop\Http\ServerMiddleware {
    DelegateInterface,
    MiddlewareInterface,
};

class Auth implements MiddlewareInterface
{
    // code
}
```

PHP 7.2

PDO Emulated Prepare

Extended string types for PDO

PDO string types now support the national character type when emulating prepare, adding these constants:

- PDO::PARAM_STR_NATL
- PDO::PARAM_STR_CHAR
- PDO::ATTR_DEFAULT_STR_PARAM

```
$db->quote('trÃ`s Ã©levÃ©', PDO::PARAM_STR | PDO::PARAM_STR_NATL);
```

Prepare Debugging

PDOStatement::debugDumpParams() method has been updated to include the SQL being sent to the DB, where the full, raw query (including the replaced placeholders with their bounded values) will be shown.

PHP 7.2

Sodium Extension

The modern Sodium cryptography library has now become a core extension in PHP. For a complete function reference, see the [Sodium documentation](#).

```
$text = 'This is something I want to encrypt!!!';  
$key = sodium_randombytes_buf(SODIUM_CRYPTO_SECRETBOX_KEYBYTES);  
$nonce = sodium_randombytes_buf(SODIUM_CRYPTO_SECRETBOX_NONCEBYTES);  
$message = sodium_crypto_secretbox(  
    $text,  
    $key,  
    $nonce  
);  
var_dump($message);
```

PHP 7.3

Probably the most significant area of improvement in PHP 7.3 is in string handling, especially more flexible handling for **heredoc** and **nowdoc**:

- The requirement to add a semi-colon after the ending delimiter is now lifted.
- In addition, the delimiter no longer has to be flush left.

In the **mbstring** extension, string handling improvements include:

- *Case handling is more accurate*
Specifically addressed are situations where case-folding or case-mapping may be used.
- *Support for Unicode 11 has been added*
- *Support is now available for strings larger than 2 GB in size*
- *Multibyte string operations are now significantly faster*

PHP 7.3

Other improvements

Some errors which were **Fatal** in PHP 7.2 are now thrown as **CompileError**.

Trailing commas are now allowed in function/method calls

Support for hashing algorithms based on **Argon2id** has been added

New **php.ini** options have been added when running PHP using FPM (Fast Processing Module), which facilitate independent logging.

Full support for the [LDAP extension](#) have been added.

PHP 7.4

Here is a list of RFCs (Request(s) For Comment) which have already been implemented in the Alpha version of PHP 7.4

- [Direct support for weak references](#)
- [Foreign Functions Interface support](#)
- [Typed properties within classes](#)
- [Null coalescing assignment operator](#)
- [Preloading scripts](#)
- [Make PHP hash extension standard](#)
- [Ability to register new PHP hashing algorithms](#)
- [Improvements in `openssl_random_pseudo_bytes\(\)`](#)
- [Multi-byte string split](#)
- [Perform *Reflection* on references](#)
- [Deprecate the WDDX extension](#)
- [Improve serialization support by adding `__serialize\(\)` and `__unserialize\(\)` magic methods](#)

PHP 7.4

Foreign Functions Interface Support

This ability allows the developer to directly work with **C language** from a PHP script. You can define your own PHP functions in C, and then call them from your application. This feature also allows userland PHP code to access **C** variables, arrays, and internal core PHP C functions. Example:

```
// create gettimeofday() binding
$ffi = FFI::cdef("
    typedef unsigned int time_t;
    typedef unsigned int suseconds_t;
    struct timeval {
        time_t    tv_sec;
        suseconds_t tv_usec;
    };
    struct timezone {
        int tz_minuteswest;
        int tz_dsttime;
    };
    int gettimeofday(struct timeval *tv, struct timezone *tz);
", "libc.so.6");
$tv = $ffi->new("struct timeval");
$tz = $ffi->new("struct timezone");
var_dump($ffi->gettimeofday(FFI::addr($tv), FFI::addr($tz)));
var_dump($tv->tv_sec);
var_dump($tz);
```

PHP 7.4

Typed Properties Within Classes

This might eliminate the need for certain "getters" and "setters". As an example, if you have this:

```
class User {  
    private $id;  
    private $name;  
    public function __construct(int $id, string $name) {  
        $this->id = $id;  
        $this->name = $name;  
    }  
    public function getId(): int { return $this->id; }  
    public function setId(int $id): void { $this->id = $id; }  
    public function getName(): string { return $this->name; }  
    public function setName(string $name): void { $this->name = $name; }  
}
```

You can do this instead, and achieve the same result:

```
class User {  
    public int $id;  
    public string $name;  
    public function __construct(int $id, string $name) {  
        $this->id = $id;  
        $this->name = $name;  
    }  
}
```

PHP 7.4

Pre-loading Scripts

Pre-loading is already supported when you use **OpCache**. However when an item is retrieved from OpCache, certain checks still need to be made such as have any changes occurred. Also, OpCache is unable to determine if there are dependencies between classes.

This new PHP 7.4 feature allows developers define a **php.ini** parameter **opcache.preload** which represents a PHP script. This PHP script will then be executed upon every execution, which has the effect of instructing OpCache to *always* cache the bytecode for PHP classes and files referenced in the script.

There is a further provision to scan for files through an entire directory structure which lets you do things like always cache entire frameworks.

IMPORTANT: this new feature will improve performance, but will cause the server to *always* cache classes and files referenced in the pre-load script. Accordingly this feature needs to be disabled in a development environment.

PHP 7.4

Register New Hash Algorithms

Currently, when using the PHP **password_*** functions, when a new hashing algorithm becomes available, developers must wait until it has been integrated before use. Further, even if a new algorithm is available on the OS of one server, its use might not be possible because functionality has not yet been integrated into the **password_*** family.

This new feature will enhance the **password_*** family by expanding the internal structure to allow the developer to add new hash algorithms as they are available, which will provide faster access to new algorithms as they become available.

PHP 8

Here is a list of RFCs which have already been implemented in the Alpha version of PHP 8

- [Consistent support numeric arrays with starting with negative indices](#)
- [Provide consistency for throwing *TypeError*](#)
- [Always generate a Fatal Error when method signatures don't match in the process of class inheritance.](#)
- [JIT \(Just In Time\) compiler ... hooray!!!](#)

Future Directions

One big question is what is the future direction of PHP. To answer this question we consult the roadmap, which is outlined here: <https://wiki.php.net/rfc>.

Proposals, currently in discussion or undergoing voting, include the following:

- [Support for annotations version 2 \(do not need to be embedded in a docblock\)](#)
- [Update the current DOM extension to the most up-to-date standard](#)
- [Add support for property level type hinting](#)
- [Comprehensions \(short generators as seen in Python, for example\)](#)

IMPORTANT: none of the proposals mentioned here have been accepted. The list presented here is designed to give you taste of what is being considered by way of language additions for the proposed PHP 8.

Module Summary

This module covered:

- PHP Unit, Documentor and Security
- PHP Configuration
- Language Update

Class Summary

This class completed:

- Understand PHP object-oriented constructs
- Database with PDO
- Database CRUD operations
- Data formats and parsing libraries
- Using web services
- Working with output control and browser caching
- Understand PHP system configuration

Before We End

Final class poll

Certificate of completion

Congratulations! You are now an intermediate PHP programmer!



Acceptance of and Conditions for Code Use

Rogue Wave Software grants you a nonexclusive copyright license to use all programming code examples from which you can generate similar function tailored to your own specific needs.

The Rogue Wave Software courses' materials are provided "as is" and subject to any statutory warranties which cannot be excluded, Rogue Wave Software, its officers, directors, employees, program developers and training partners make no warranties or conditions either express or implied, including but not limited to, the implied warranties or conditions of merchantability, fitness for a particular purpose, and non-infringement, regarding the courses, Rogue Wave Software courses' materials or programs provided, if any.

Under no circumstances is Rogue Wave Software, its officers, directors, employees, program developers or training partners liable for any of the following, even if informed of their possibility:

- Loss of, or damage to, data;
- Direct, special, incidental, or indirect damages, or for any economic consequential damages; or
- Lost profits, business, revenue, goodwill, or anticipated savings.
- Accuracy or completeness of the Rogue Wave Software courses' materials.

Some jurisdictions do not allow the exclusion or limitation of direct, incidental, or consequential damages, so some or all of the above limitations or exclusions may not apply to you.