

PHP Fundamentals II

Courseware by Perforce 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.

Course Details

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 Details

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

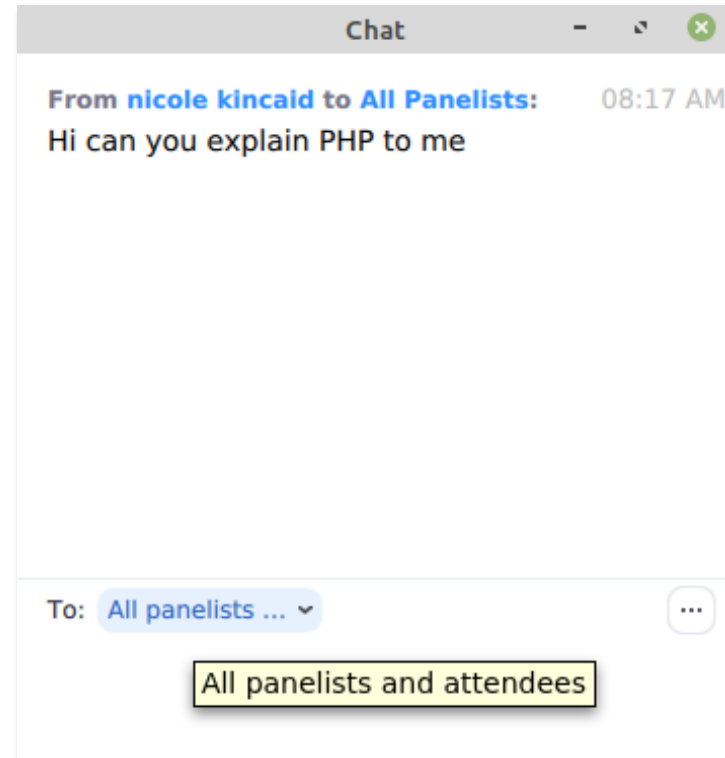
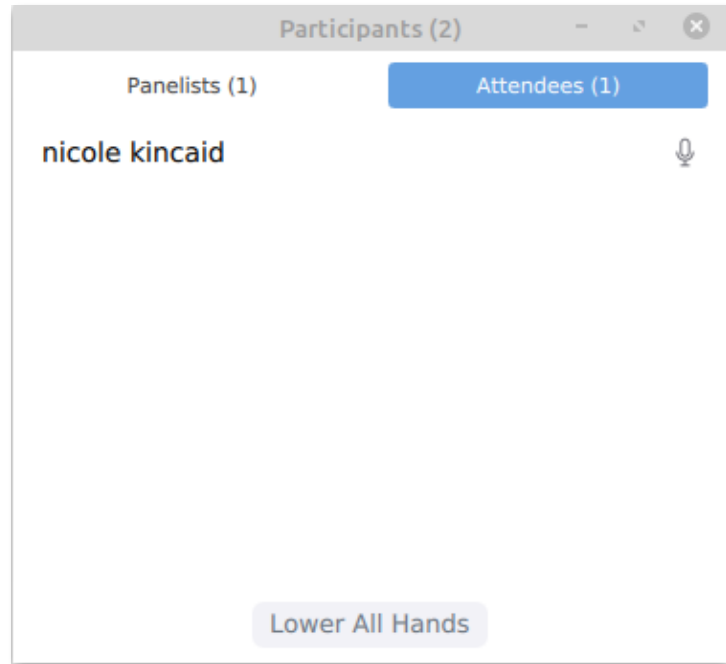
The course uses Zoom web conferencing and includes participant and chat panels.

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.
- A database server.
- The PHP language interpreter.
- Gedit, the text editor.
- An integrated development environment tool.
- A database administration and modeling tool.
- A collaboration tool for API development.
- Course applications.

Course Virtual Machine

Course Applications/Projects

Applications/Projects included:

- OrderApp
- php2
- 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
- Trait
- Object cloning
- Nullsafe operator
- Enumerations

PHP Namespaces

PHP Namespaces

A *namespace* is an encapsulating mechanism for classes, traits, interfaces, constants, and functions. These constructs are susceptible to naming collisions with other code namespaces including the global namespace.

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

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

PHP Namespaces

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;  
    }  
}
```

Grouped Namespace Trailing Comma

A trailing comma can be added to the group-use syntax:

```
namespace Manage\Service;  
use Interop\Http\ServerMiddleware\ {  
    DelegateInterface,  
    MiddlewareInterface,  
};  
...
```

Considerations

Notes:

- Namespaces can encapsulate with curly braces.
- Namespaces sometimes map directly to a directory structure thereby conforming to the *PSR-4* autoloading standard.
- The *namespace* keyword is resolvable to the current namespace name along with 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. PHP attempts to *autoload* a class whose definition has not been loaded into memory. An *autoloader* is a function, anonymous function, a class method, or third party autoloader that is declared and registered with the parser as an autoloader.

To implement one or more autoloaders, the `spl_autoload_register()` function is available, and is used in the class project. All of the following are used as autoloaders:

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

The PSR-4 standard defines the relationship between namespaces and autoloading.

Composer Autoloading

Composer, a PHP package manager, includes a commonly used autoloader.

When available, loading the *Composer* autoloader is required. From that point, and assuming some configuration, Composer's autoloader 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 above, execute the following shell script, at the project root, to configure the auto loader:

```
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, properties, and methods:

```
class UserEntity {  
    public const TABLE = 'user';  
    protected $id, $firstName, $lastName;  
    public function setId(int|string $id) :void {  
        $this->id = $id;  
    }  
    public function setFirstName(string $firstName) :void {  
        $this->firstName = $firstName;  
    }  
    public function setLastName(string $lastName) :void {  
        $this->lastName = $lastName;  
    }  
    public function getId() :int {  
        return $this->id;  
    }  
    public function getFirstName () :string {  
        return $this->firstName;  
    }  
    public function getLastName () :string {  
        return $this->lastName;  
    }  
    public function getFullName () :string {  
        return trim(($this->firstName . ' ' ?? ")  
            . ($this->lastName ?? "));  
    }  
}
```

Classes and Class Members

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) :void {  
        $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 (string, int, float, bool)
- null
- Another object
- An array
- A resource

Classes and Class Members

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:

```
class UserEntity {  
    public const TABLE = 'user';  
  
    // Properties  
    public int $id = 0;  
    protected string $firstName = "";  
    protected string $lastName = "";  
  
    // "new" is allowed in PHP 8.1+  
    public DateTime $time = new DateTime();  
  
    public function getFirstName() :string {  
        return $this->firstName;  
    }  
}
```

As of PHP 7.4, properties can also have a data type assigned. The data type can include scalar, object class, or interface.

Class Property Best Practices

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.
- Define classes with a narrow aspect which makes them easy to extend.

Classes and Class Members

Class Method

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

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

Methods

Class method:

- Is 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 specifies availability of a class constant, property or method to other code. Think of a visibility as a mechanism to control access to class members.

Three visibilities are available:

Public:

Allow access from any scope and does not require accessor methods.

Protected:

Allow direct access from defined and derived class scopes only. Is inheritable.

Private:

Allow access only from defined class scope. Not inheritable.

PERFORCE

Classes and Class Members

\$this

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

\$this is only used within the context of object code. It is necessary to have an object instance that the *\$this* is bound to at runtime.

```
class UserEntity {  
    public const TABLE = 'user';  
    protected $firstName, $lastName;  
  
    // Bound $this reference  
    public function getFirstName() :string {  
        return $this->firstName;  
    }  
}
```

Objects

Objects

An object is an instance of a class, and contains all defined, and inherited, class members. They can have unique properties and methods and represent a unique instance of a class. An instance might be an individual user, employee, invoice, ball, shipping, etc. The `new` keyword is required:

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

Objects

Unique Instances Object Constructor

Objects can and usually do have unique values set at the time of instantiation.

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:

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

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

Objects

Unique Instances

Object Constructor Promotion

The previous example of the object constructor required lots of boilerplate code. Instead, with PHP 8, it is possible to use *constructor promotion* to declare and assign the values within the parameter signature with concise code. Notice the property declarations, and explicit setting within the constructor is no longer necessary:

```
class UserEntity {  
    public function __construct(  
        public string $firstName,  
        public string $lastName  
    ) {}  
}  
  
$user1 = new UserEntity('Jack' , 'Ryan');  
$user2 = new UserEntity('Monte' , 'Python');
```

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.
- Construct 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, or use constructor promotion to cut down on the boilerplate code.
- Use to setup dependent properties.

Class Functions

PHP has several *class functions* tailored for working with classes:

`get_class()`:

Return a class name.

`get_class_methods()`:

Return an array of a classes methods.

`class_exists()`:

Return boolean on check of a loaded class.

`get_parent_class()`:

Return the name of the parent class.

`get_class_vars()`:

Return an array of the classes default properties.

`method_exists()`:

Return boolean on check of a method existence.

`property_exists()`:

Return 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 a cornerstone of object oriented programming. It 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;  
    }  
}
```

Inheritance

A Superclass

A superclass example:

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

A subclass example:

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

Inheritance

Subclass Extending Superclass

Superclass Method Call With `parent::<method>`

A super class method is available from a subclass method by calling the *parent::<method name>* reference. This is a traditional classical inheritance model:

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

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

Inheritance

Subclass Extending Superclass With Constructor Promotion

A super class method is, potentially, no longer required unless needed by other subclasses. In this case, if the *GuestUser* subclass is the only extending subclass, then the parent constructor is not needed:

```
// The superclass without constructor
class UserEntity {
    ...
}

// The subclass with constructor promotion eliminates the parent call. As a result, this, potentially,
// could eliminate the parent constructor depending on subclass use.
class GuestUser extends UserEntity {
    public function __construct(
        protected string $firstName,
        protected string $lastName,
        protected string $role
    ) {}
}
```

Inheritance

Subclasses Extending Superclass With Constructor Promotion

A super class method with two subclasses calling the parent method:

```
// A superclass using constructor promotion
class UserEntity {
    public function __construct(public string $firstName, public string $lastName) {}
}

// The subclass using constructor promotion calling the parent constructor
class GuestUser extends UserEntity {
    public function __construct(string $firstName, string $lastName, protected string $role) {
        parent::__construct($firstName, $lastName);
    }
}

// A subclass using constructor promotion calling the parent constructor
class AdminUser extends UserEntity {
    public function __construct(string $firstName, string $lastName, protected string $role) {
        parent::__construct($firstName, $lastName);
    }
}

$o = new AdminUser('Mark', 'Watney', 'astronaut');
echo $o->lastName;
```

Inheritance

Class Inheritance Considerations

Notes:

- Visibilities allow, limit or prevent inheritance.
- It is necessary to either explicitly load, or autoload a super class before extending.
- Multiple inheritance is not supported 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

Inheritance

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 are needed. They can contain as many methods as desired, including magic methods. They can implement interfaces and extend super classes.

PERFORCE

Anonymous Classes

Use Case

This example tests a *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

Overriding

Overriding happens when a subclass re-declares a method or property defined in a superclass.

Overriding

Method Overriding

Unless otherwise specified, class properties and methods can be overridden as seen previously with a constructor. 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 string $firstName;
    public function setFirstName ($firstName) {
        $this->firstName = $firstName;
    }
}

// The subclass
class GuestUser extends UserEntity {
    protected string $mi;
    public function setFirstName($firstName, $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.

Overriding

Property Overriding

This example shows a superclass that has a generic table name. The subclass overrides this property:

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

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

Overriding

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 UserEntity {
    // **** 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()`
- `__serialize()`
- `__unserialize()`

Magic Methods

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:

Method	Triggered by ...
<code>__get()</code> :	... a get call to an inaccessible property.
<code>__set()</code> :	... a set call to an inaccessible property.
<code>__call()</code> :	... a call to an undefined method.
<code>__callStatic()</code> :	... a static call to an undefined method.
<code>__sleep()</code> :	... a <code>serialize()</code> call, and executes prior to serialization.
<code>__wakeup()</code> :	... an <code>unserialize()</code> call, and executes prior to object reconstitution.
<code>__clone()</code> :	... the <code>clone</code> keyword, and executes prior to duplication.
<code>__toString()</code> :	... treating an object as a string, requires a string return, and allows thrown exceptions.
<code>__destruct()</code> :	... <code>unset()</code> , or runtime completion, and is called last in/first out (LIFO) sequence.
<code>__invoke()</code> :	... treating an object as a function. Makes the object callable; useful when defining event listeners.

Magic Methods

__toString()

The `__toString()` method is called when an object is used as if it were a string. A `__toString()` can only return a string, or throw an exception:

```
class UserEntity {  
    public function __construct(  
        protected string $firstName,  
        protected string $lastName) {  
    }  
  
    public function __toString(): string {  
        return $this->firstName . ' ' . $this->lastName;  
    }  
}
```

```
$userEntity = new UserEntity('Mark', 'Watney');  
echo $userEntity; // outputs: "Mark Watney"
```

Magic Methods

__toString() Exceptions

The `__toString()` method can throw an exception:

```
class UserEntity {  
    public function __construct(  
        protected string $firstName,  
        protected string $lastName) {  
    }  
  
    public function __toString() {  
        throw new Exception('Error message');  
    }  
}  
  
$userEntity = new UserEntity('Mark', 'Watney');  
try{  
    echo $userEntity; // triggers an exception  
} catch (Throwable $e){  
    echo 'something';  
    // handle error  
}
```

Magic Methods

__get()

Here is a magic `__get()` example:

```
class UserEntity {  
    public function __construct(  
        protected string $firstName,  
        protected string $lastName  
    ) {}  
  
    // Returns an inaccessible property  
    public function __get($value) {  
        return $this->$value;  
    }  
}
```

```
$userEntity = new UserEntity('Mark', 'Watney');  
echo $userEntity->firstName; // outputs: "Mark"
```

PERFORCE

Magic Methods

__sleep()

The magic `__sleep()` method is called by the `serialize()` function to create a string representation of an object with an object's properties returned as an array:

```
class Foo{
    public function __construct(protected string $fooValue){}
    public function __sleep(): array {return ['fooValue'];}
}

$foo = new Foo('some stuff');
echo serialize($foo); // O:3:"Foo":1:{s:11:"fooValue";s:10:"some stuff";}
```

A string representation of an object is useful when storing an object's state in an SQL data store.

PERFORCE

Magic Methods

__wakeup()

The magic `__wakeup()` method is called by the `unserialize()` function to re-instate an object, as long as the class is loaded:

```
class Foo{
    public function __construct(protected string $fooValue){}
    public function __sleep(): array {return ['fooValue'];}
    public function __wakeup(): void {}
}

$foo = new Foo('some stuff');
$stringRepresentation = serialize($foo);

$fooReinstated = unserialize($stringRepresentation);
var_dump($fooReinstated); //object(Foo)#2 (1) [{"fooValue":protected}=>string(10) "some stuff"]
```


Magic Methods

__serialize()

The magic `__serialize()` method, similar to the magic `__sleep()` method, will return a serialized string. It is different than `__sleep()` in that it will allow stateful additions that are not part of the object's original properties:

```
class Foo{
    public function __construct(private $fooValue){}
    public function __serialize(): array {
        // Return array of serialized values
        return ["fooValue" => 'some new value'];
    }
}

$instance = new Foo('stuff');
echo serialize($instance); // O:3:"Foo":1:{s:8:"fooValue";s:14:"some new value";}
```

PERFORCE

Magic Methods

__unserialize()

This example shows the magic `__unserialize()` method used to re-instate an object with an addition. It differs from magic `__wakeup()` in an ability to add new state values:

```
class Foo{
    public function __construct(private $fooValue){}
    public function __serialize(): array {
        return ["fooValue" => 'some new value'];
    }
    public function __unserialize(array $data): void {
        $this->fooValue = $data["fooValue"];
        $this->barValue = 'barValue';
    }
}

$instance = new Foo('stuff');
$serializedString = serialize($instance);
var_dump(unserialize($serializedString));
/* object(Foo)#2 (2) {[
    "fooValue":"Foo":private=> string(14)
    "some new value":["barValue"]=> string(8) "barValue"
}*/
```

Best Practices

Best practices:

- 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. *Concrete* subclasses are the objects instantiated, rather than the *abstract* superclass.

Abstract classes cannot themselves instantiate as objects. They are used in inheritance as a superclass.

Classes that have *abstract* methods should be marked *abstract*, due to the extensible requirement of a subclass defining the *abstract* method.

Abstract Classes and Methods

Abstract Form Class

An abstract form class serving as a superclass:

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

    public function addTagAttributes(){...}
}
```


Abstract Method

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

Abstract Classes and Methods

Abstract Method

In the superclass example below, the method *getInput()* is required, but the implementation is unknown and implemented by subclass 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.
- To serve as a class template, if required.

Abstract Class and Method Labs

Abstract Classes and Methods

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. It is similar to an abstract class declaration, but does not serve the inheritance purpose of a superclass. The interfaces' declared methods are often required due to an underlying code base need, and/or a desire to type hint on an interface.

Interfaces

A Service Interface

This interface declares two methods:

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

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

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


Interfaces

Parameter Widening

Parameter types from overridden methods, and from interface implementations, may be omitted:

```
interface Foo {  
    public function test(array $input);  
}  
  
class Bar implements Foo {  
    // type omitted for $input  
    public function test($input) {  
        return 'You requested ' . htmlspecialchars($input);  
    }  
}  
  
$bar = new Bar();  
echo $bar->test('something');
```

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.

Interfaces

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.

You can use a super class (or an abstract class) as a type hint. Any subclass will satisfy the type hint due to PHP's support of [subtype polymorphism](#).

Type hinting on interfaces is even better as it 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.

Required Types

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

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


Type Hinting

Parameter Type Hints

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

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

Type Hinting

Union Type Hints

union types are used when the method argument can accept multiple specified data types.

This class requires either an integer, or string type passed:

```
class Test {  
    protected $data;  
    public function __construct(int | string $data) {  
        $this->data = $data;  
    }  
}
```

This class uses constructor promotion requiring an integer or string type passed.

```
class Test {  
    public function __construct(protected int | string $data)  
    {}  
}
```

This property is defined using union types:

```
class Test {  
    protected int | string $data;  
    public function __construct($data) {  
        $this->data = $data; // only int or string are accepted  
    }  
}
```

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"
```

Type Hinting

Class and Interface

A *class name* type hint in a constructor requiring a service instance:

```
use OrderApp\Core\Service\Services;
class AbstractModel implements ModelInterface {
    protected $services;
    public function __construct(Services $services) {
        $this->services = $services;
        $this->db = $this->services->getDb();
    }
}
```

An instance implementing the interface:

```
use OrderApp\Core\Service\Services;
class AbstractModel implements ModelInterface {
    protected $services
    public function __construct(ServiceInterface $services) {
        $this->services = $services;
        $this->db = $this->services->getDb();
    }
}
```

Type Hinting

Object Type Hint

Object parameter and return type hints define when an object is expected as a parameter, or returned:

```
// Object parameter and return type
function foo(object $bar): object {
    return $bar;
}

class Bar{}
var_dump(foo(new Bar)); // object(Bar)#1 (0) {}

// Handled in try/catch
try{
    $instance = foo('some string');
} catch (Throwable $e){
    echo $e->getMessage(); // foo(): Argument #1 ($bar) must be of type object, string given ...
    ...
}
```

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:

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

Type Hinting

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;  
    }  
}
```

Return Typing Static

A special return type, *static*, allows return of a an instance of a calling class and not of an inherited parent. This is detailed more in the section on static declarations:

```
class Foo {  
    public function bar($param): static {  
        return new static($param);  
    }  
}
```


Type Hinting

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 {
    public function __construct(
        protected string $arg1,
        protected int $arg2
    ) {}
}
```

Type Hinting

Intersection Type Hint

intersection type hints were introduced in PHP 8.1. Generally speaking, you use an *interface* when you wish to guarantee certain functionality. Interface data type hints work if you only want to test for a *single* interface. If you want a method that accepts an argument where you want the functionality of two or more interfaces, prior to PHP 8.1 you needed to create either a new interface that extended the ones with the desired functionality or test the argument directly using `instanceof`.

```
class Test
{
    public function scan($iter)
    {
        if (!$iter instanceof ArrayAccess
            || !$iter instanceof Countable)
            throw new InvalidArgumentException();
        $output = "";
        $max = count($iter);
        for ($x = 0; $x < $max; $x++)
            $output .= $x . ':' . $iter[$x] . "\n";
        return $output;
    }
}

$a = new ArrayIterator(['A','B','C']);
$test = new Test();
echo $test->scan($a);
```

Type Hinting

Intersection Type Hint Usage

In PHP 8.1 you can now use the ampersand ("&") to define what is known as an *intersection* type. In this example, we use an intersection type to test for the combined functionality of the `ArrayAccess` and `Countable` interfaces.

Note that a *union* type would not accomplish the purpose of this example. With a union type you test for either/or, whereas with an intersection type you test for *combined* functionality.

```
// only works in PHP 8.1 and above
class Test {
    public function scan(ArrayAccess & Countable $iter)
    {
        $output = "";
        $max = count($iter);
        for ($x = 0; $x < $max; $x++)
            $output .= $x . ':' . $iter[$x] . "\n";
        return $output;
    }
}
$a = new ArrayIterator(['A','B','C']);
$test = new Test();
echo $test->scan($a);
```

For a thorough discussion on this type hint, have a look at [PHP RFC: Pure intersection types](#).

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 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 within 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 and reference to the global namespace.

Exceptions

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 its constructor if the custom exception defines a constructor.

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

Exceptions

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:

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

Exceptions

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 {
    $pdo = new PDO('mysql:host=localhost;dbname=phpcourse', 'vagrant', 'vagrant');
    if (!$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');
}
```

Exceptions

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 ...  
}
```

Exceptions

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 ...  
}
```

Exceptions

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 the *Error* class, most notably:

- *ParseError*
- *TypeError*

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

Exceptions

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() {  
    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/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:

```
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;
    }
}
$symbol = Currency::$currency; // $symbol == "$"
echo Currency::format(100); // output: "$100.00"
```

NOTE: in some situations, inherited methods bind *eagerly* and might result in unexpected behavior. Have a look here for a discussion on the issue of [Late Static Binding](#).

Trait

Trait

A *trait* is a language construct that allow blocks of code inserted into more than one class at a time. It is best used to avoid duplication.

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

A *trait* has 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

Sharing Code

This trait has code shared by more than one concrete subclass:

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


Trait

Trait Used in a Class

Trait use in a class by the *use* declaration:

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

Trait

Visibilities

With a trait, 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}  
}
```

Traits

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

Traits

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 Cloning

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>

Object Cloning

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.

Nullsafe Operator

Nullsafe Operator

The *nullsafe* operator `?->` provides concise function tests for a null values.

```
// Demo classes
class Model{
class UserEntity{
class AddressEntity{

$model = new Model; // Instantiate empty model
$model->userEntity = new UserEntity; // Inject a UserEntity instance
$model->userEntity->addressEntity = new AddressEntity; // Inject an AddressEntity instance
$model->userEntity->addressEntity->country = 'someCountry'; // Assign a country property value

// Traditional nested null value conditional test structure
$country = null;
if ($model !== null) {
    $userEntity = $model->userEntity;
    if ($userEntity !== null) {
        $addressEntity = $userEntity->addressEntity;
        if ($addressEntity !== null) {
            $country = $addressEntity->country;
        }
    }
}

// Equivalent test with null-safe operator
echo $model?->userEntity?->addressEntity?->country; // someCountry
```

Nullsafe Operator

Short Circuiting

The *nullsafe* operator can short circuit a conditional test:

```
class Model{
    public function process(){
        return 'some expensive processing';
    }
}

$model = new Model;
$foo = null;

// Traditional null test to short circuit call to expensive method
if($foo !== null){
    $result = $model->process();
}

// Equivalent with nullsafe. Expensive process not called on $foo === null
$result = $foo?->$model->process();

// Works with nested nullsafe tests
// $foo?->bar?->baz()?->boo?->bit();
```

Enumerations

Enumerations

PHP 8.1 introduced *Enumerations* (also called *Enums*. Enums are a good way to represent lists of relatively unchanging default values. They also make really great type hints.

- Enums can be defined just like a class, interface or trait using the keyword `enum`.
- Use `case` to define the items within the enumeration. Each case follows the same rules as *constants*.
- Enums include a property named `value` that returns the value of an enum case.
- A method `cases()` returns an array of all cases defined in the enum.
- An enum implements an internal interface `UnitEnum`

Here's a simple example:

```
enum Gender
{
    case MALE;
    case FEMALE;
    case OTHER;
}
```

For more information see the PHP documentation on [Enumerations](#).

Enumerations

Why Use Enumerations?

To illustrate why you might want to use an Enumeration, consider the following class:

```
class Signup
{
    const MALE = 'M';
    const FEMALE = 'F';
    const OTHER = 'X';
    const GENDER = [self::MALE, self::FEMALE, self::OTHER];
    const ERR_GENDER = 'ERROR: gender must be one of M|F|X';
    public function __construct(
        public string $username,
        public string $password,
        public string $dateOfBirth,
        public string $gender)
    {
        // To constrict the value of $this->gender you need to
        // add an extra sanity check inside the body of the __construct() method:
        if (!in_array($gender, self::GENDER))
            throw new Exception(self::ERR_GENDER);
    }
}
```

Enumerations

Why Use Enumerations?

If you pass a string as an argument, the class construction succeeds:

```
$fred = new Signup('fred', 'pass', '1970-01-01', Signup::MALE);
```

If you pass a value other than "M", "F" or "X", an Exception is thrown:

```
$wilma = new Signup('wilma', 'pass', '1970-01-01', 'FEMALE');
```

```
PHP Fatal error: Uncaught Exception: ERROR: gender must be one of M|F|X in oop_before_enum.php:17
```

Enumerations

Using Enumerations

If you rewrite the same example above using Enumerations, notice how many lines of code are saved. You'll also note that there no longer a need to add the extra sanity check inside the `__construct()` method.

```
class Signup
{
    public function __construct(
        public string $username,
        public string $password,
        public string $dateOfBirth,
        // NOTE: using the Gender enum shown in an earlier slide
        public Gender $gender)
    {}
}
```

Enumerations

Using Enumerations

Now, instead of passing a string that represents gender, you pass one of the enum cases

```
$fred = new Signup('fred', 'pass', '1970-01-01', Gender::MALE);
```

If you pass any value other than one of the enum cases, a Fatal Error is thrown:

```
$wilma = new Signup('wilma', 'pass', '1970-01-01', 'FEMALE');  
PHP Fatal error: Uncaught TypeError: Signup::__construct(): Argument #4 ($gender) must be of type Gender,  
string given, called in oop_enum.php on line 37 and defined in oop_enum.php:12
```

Other Considerations

There are a few additional considerations that we mention here.

[Backed Enumerations](#)

A *backed enumeration* is where you assign a value to the case. Use the `value` property to access the default.

[Enumeration Methods](#)

Enumeration may contain constants, regular or static methods, and can also implement interfaces and use traits.

[Cases as Constants](#)

An enum case can be used outside of the enum in the manner that you would use class constants. The only exception is that you can't use array syntax.

Other Considerations

[Not Objects](#)

Although enums are built upon the same internal foundation as PHP classes and objects, the following restrictions apply:

- No magic methods, except for `__call()`, `__callStatic()` and `__invoke()`
- No properties
- No support for inheritance
- No cloning

Enums can be serialized, however you cannot implement serialization-related magic methods.

If you are interested in more information, consider joining the *PHP Fundamentals III* course (shameless plug!), where we cover Enumeration usage in greater detail.

Module Summary

This module included:

- 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
- Trait
- Object cloning
- Nullsafe operator
- Enumerations

OrderApp OOP Implementation

This module covers:

- OrderApp OOP implementation
- File organization
- Layout scripts

OrderApp OOP Implementation

OrderApp OOP Implementation

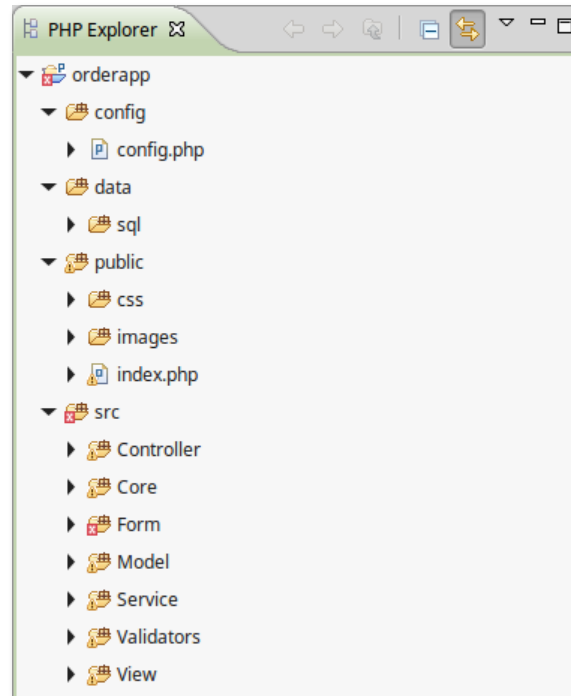
The OrderApp was developed for the PHP Fundamentals courses and covers all basic features of a web-based application such as form validation and database storage.

This module will take the PHP object oriented constructs learned in the last module and apply them to the OrderApp application.

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 that contains an array of configuration.

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

The config array contains top level keys for database configuration. The key *db* key provides configuration for the database server, and a *services* key provides application-wide service configuration.

The Data Directory

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

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: From the application, write data only 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*, like a framework, are classes that are domain agnostic, 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 ng 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

Domain Classes

In addition to domain controllers, these classes relate directly to the domain application being built. These define the domain 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.

Layout Scripts

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:

- Domain and object models
- Data model
- PHP Data Objects (PDO)
- Database operations

Domain and Object Models

Domain and Object Models Definitions

Domain:

A *domain* is any field, area, or discipline. Any enterprise, business, or institution. It is considered a *domain* from a software perspective, and can include, among other things; an airline, hospital, or academic university. Domains may have other domains within called *subdomains*.

Domain Model:

A *domain model* is an expressed software model of a domain. *Domain modeling* is a design practice that architects a software based on *domain* knowledge and requirements.

Object Model:

A software construct *class* that models domain elements.

Domain Model Software Pattern

A *domain model* is also a software pattern that composes other software patterns. This pattern offers a full domain and persistence abstraction in the data modeling layer:

Creational:

Factory, Singleton, Builder.

Structural:

Adapter, Module, Decorator, Proxy.

Behaviorial:

Pub/Sub, Strategy, Iterator.

Concurrency:

Active Object, Lock, Reactor, Scheduler.

One goal of a *domain model* is complete ignorance of the implemented data persistence architectural layer. The *domain model* code should not be concerned with where and how the data is persisted, just that it is persisted.

Domain Model Components

Domain Model components include:

Domain entities:

A domain element that has identification.

Domain value objects:

A domain element that does not have or need identification, and is transient by nature.

Hydrator:

A software class that populates, or extracts data from, an object model.

Mapper:

A software component that maps the difference between an object and the data model. It can include an *abstraction* layer that abstracts away most of data persistence layer query language.

Repository:

A software component that serves as a gateway to a modular system of objects.

Model Components

Domain Entities

Domain 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.

Domain and Object Models

Entities

RegistrantEntity

A *RegistrantEntity* class can consume a *PersonEntity* instance, or that person that registering for an event, like a concert, conference. A *RegistrantEntity* is, perhaps, what the domain calls this entity, so it's named the same:

```
class RegistrantEntity implements HumanInterface{
    protected int $id;
    public function __construct(
        protected HumanInterface $personEntity
    )
    {}
    public function getPersonEntity(): string { return $this->personEntity; }
    public function getId() {return $this->id;}
}
```

Domain and Object Models

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; }
}
```

Domain Value Objects

Domain *value objects* are immutable and transient objects. 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

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

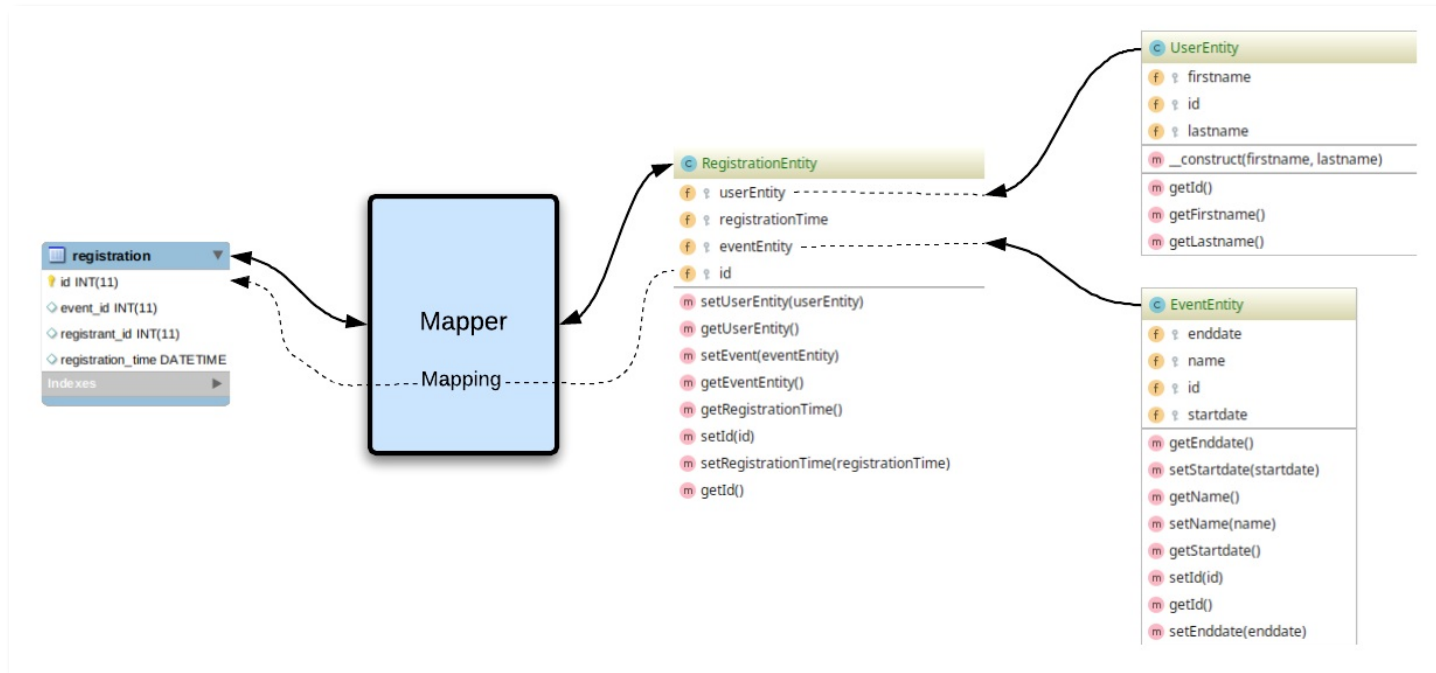
A database might have a column named *first_name* , but the object model property name is *\$firstName*, so they are not directly linkable without a mapper. The mapper maps the two together.

PERFORCE

Domain and Object Models

Mapping

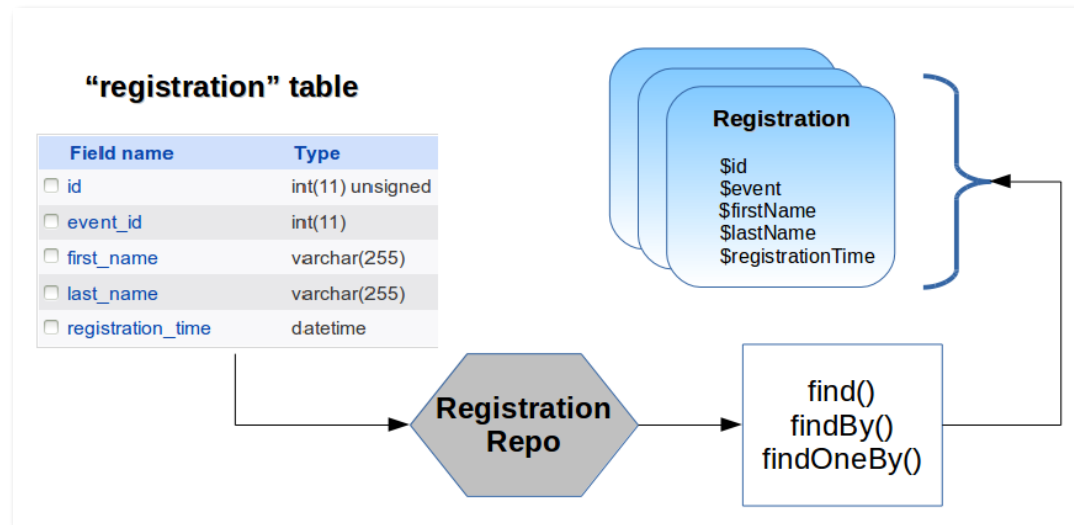
A mapper maps the entities to their associated database table(s).



Domain and Object Models

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:

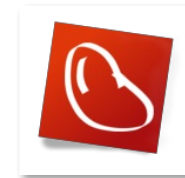
Doctrine



Propel



RedBeanPHP



Data Model

Data Model

A *data model* is an expressed database model of a domain's data. *Data modeling* is a design practice that creates and maps a database system to, and is guided by, a *domain model*.

Relational Data Models and Structured Query Language (SQL)

SQL is a programming language that interface relational databases in order to design schema, or perform Create, Read, Update, and Delete (CRUD) operations on data by query.

Queries are groupings of SQL statements stringed together and terminated with semicolon that perform schema design, or CRUD operations.

SQL is divided into two major SQL groups:

Data Manipulation Language (DML):

Commands that perform CRUD operations on data.

Data Definition Language (DDL):

Commands that perform structural manipulation, like creating databases and tables, or removing them.

Data Model

DML Keywords

The most frequent DML keywords follow:

INSERT INTO:

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.

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:

Create a new table.

DROP TABLE:

Delete a table.

ALTER TABLE:

Alter a table.

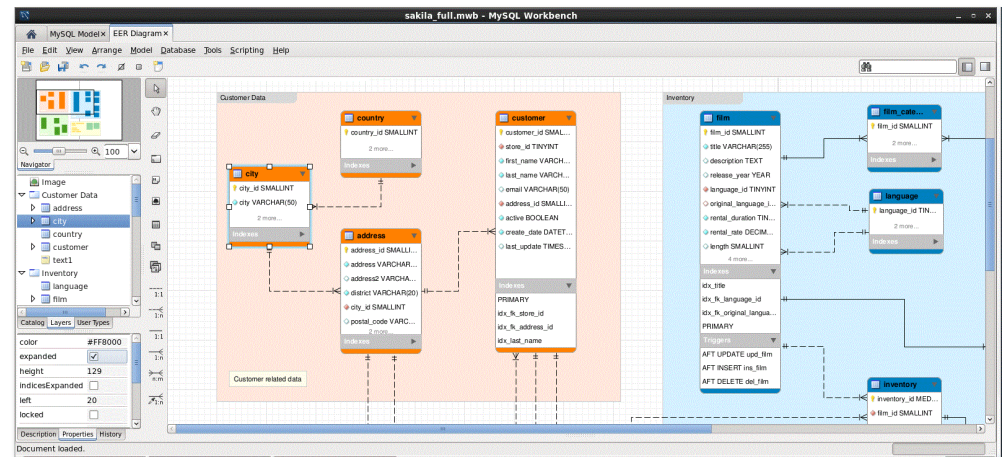
Data Model

Tools

Most RDBMS's have tools written to help with data modeling.

Here's an example of one used to model MySQL databases. It's called *MySQL Workbench*.

MySQL Workbench is free and available on all platforms.



Relationships

Part of an SQL data model includes fundamental relationships:

One-to-One::

How a single table row maps to a single row in another table.

One-to-Many::

How a single table row maps to a single row 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.

SQL Statements Lab

Relations and SQL

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;
```

PHP Data Objects (PDO)

PHP Data Objects (PDO)

PDO is 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.

PERFORCE

PHP Data Objects (PDO)

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_INTO:
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=phpcourse', '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.

PERFORCE

PHP Data Objects (PDO)

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).

Get the connection PDO instance via Unix socket

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

Or

Get the connection PDO instance from a host IP

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

The SELECT Statement

This code assumes a valid PDO connection object:

```
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

This code assumes a valid PDO connection object:

```
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  
}
```

PERFORCE

PHP Data Objects (PDO)

The INSERT INTO Statement

This code assumes a valid PDO connection object:

```
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

This code assumes a valid PDO connection object:

```
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. When the statement is executed, it stores itself on the database engine, which improves security and performance. A parameter for each marker is required, and is the only part of the statement that is exposed to SQL injection attacks.

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=phpcourse','vagrant','vagrant',
        [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=phpcourse','vagrant','vagrant',  
        [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=phpcourse','vagrant','vagrant',
        [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 an array with named parameters:

```
try {
    // Get the connection instance
    $pdo = new PDO('mysql:host=localhost;dbname=phpcourse','vagrant','vagrant',
        [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 includes parameter markers. 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, only parameters. Like prepared statements, benefits include:

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

Stored Procedure Loading

This is a single statement example, but multiple statements using bound parameters are possible within the BEGIN and END statements.

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

```
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 ;
```

Note: The SQL for this stored procedure is in DefaultWorkspace/php2/src/ModDB/*.

Stored Procedure Positional Parameters

Calling with positional parameters:

```
try {  
    // Get the connection instance  
    $pdo = new PDO('mysql:host=localhost;dbname=phpcourse','vagrant','vagrant',  
        [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  
}
```

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

Transaction

A transaction is a 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:

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.

Database Operations

Transactions

Transaction Code

A typical transaction code sequence:

```
try {  
    // Get the connection instance  
    $pdo = new PDO('mysql:host=localhost;dbname=phpcourse','vagrant','vagrant',  
        [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

Complete the following:

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

Complete the following:

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

Lab: Transaction

Complete the following:

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

Module Summary

This module covered:

- Domain and object models
- Data model
- PHP Data Objects (PDO)
- Database operations

Internet Communications

This module covers:

- Output control
- Email

Output Control

Output Control

Output control captures application output, stores it in server memory 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.

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

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 Control

Output Buffering

This code demonstrates calling `header()` after echo statements, which is only possible with output buffering turned on:

```
// Start buffering
ob_start ();

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

// This can execute anywhere
header('Content: Something very cool for you');

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

Nested Buffer Example

This code demonstrates nested buffering:

```
// 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;
```

Output Control

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:

```
// The date/time after which the response is considered stale.  
header('Expires: ' . gmdate ( 'D, d M Y H:i:s' , time() + 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 request and 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 a location header with 304 response and exit.

Output Control

ETag Header Example

This example sets and responds on a received *etag* hash. This helps unload the server.

```
// 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;
$expires = $current + $duration;
$_SESSION['etag'] = $etag;
$_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:

- `$_SERVER['HTTP_USER_AGENT']`
The user agent (browser) identification.
- `$_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

The `mail()` function facilitates sending email. Configuration in the `php.ini` file for a mail transport agent (MTA) is required. It is elementary by design, but other APIs™s are available to enhance the core email functionality, APIs™s like `Laminas\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);
```

Email

Additional Email Headers

Standard headers, defined by internet messaging standards RFC 2822 section 3.6 (Field Definitions), are acceptable.

Headers represent the fourth argument to the `mail()` function, and form a single continuous parameter string separated by CR-LF ("`\r\n`").

```
$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: mark@astronauts.net';  
$headers[] = 'Bcc: cal@ecalevans.com';  
mail($to, $subject, $body, implode("\r\n", $headers));
```

Email Configuration

Configuration directives directly impacting PHP's email function:

`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

Complex email operations are simplified by incorporating third party software. Most frameworks have an email component:

- **Laminas**: Laminas\Mail\Message
- **Code Igniter**: Email Class
- **CakePHP**: Cake\Mailer\Email

In addition, stand-alone libraries are available. An example is [PHPMailer code on github.com](#).

Module Summary

This module covered:

- Output control
- Email

Regular Expressions

This module covers:

- Regular expressions
- Meta characters
- Character classes
- Quantifiers
- Greediness
- Pattern modifiers
- Perl compatible regular expression functions

Regular Expressions

Regular Expressions

A regular expression is a pattern string parsed by the regex parsing engine, and used to find patterns in other strings. It is performance expensive, but powerful. Pattern strings must wrap in a character *delimiter*, which identifies the start and end of a pattern string.

Most commonly used pattern delimiters are `/`, `#`, or `!`. Other options for input validation are recommended first, such as `ctype*`() or `str*`() functions due to the parsing expense. Best uses include:

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

Meta Characters

Meta Characters

Metacharacters are characters that have special meaning to the regex engine, and, as a limited list, include:

- The dot ("."), which matches any character, except new line characters. It requires *escaping* if used literally.
- The pipe symbol ("|") that allows alternates.
- Use parentheses "(xxx)" that designates a sub-pattern.

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

```
/Table.*\.php/
```

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

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

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

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

Meta Characters

Positioning

These characters have special meaning to the parser:

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

Meta Characters

Positioning Examples

Examples:

Matches string which starts with letters A to Z:

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

Matches string which ends with *jpg* or *png*:

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

Matches string which contains exactly the distinct word *ERROR*:

```
/bERROR\b/
```

Character Classes

Character Classes

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.

The regex parser will only match one of the presented characters of a character class.

Character classes are available in two different forms:

- Shortcut character classes
- Custom character classes

For more information, see the [PHP documentation on character classes](#).

NOTE: the documentation mentions *POSIX* notation, however it is no longer supported in PHP 8.

Character Classes

Custom Character Classes

Square brackets ("[]") enclose character classes. A dash ("-") indicates a range of characters or numbers. A caret ("^") negates the character class list.

- [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

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:

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


Character Classes

Shortcut Character Classes

Shortcut versions of standard character classes:

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

Matches string which only contains alpha numerics:

```
/^\w*$/
```

Matches string which only contains digits:

```
/^\d*$/
```

Quantifiers

Quantifiers

Quantifiers quantify the number of single character token matches required on the previous character token:

- ?
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.

Matches 1 or more characters between <h1> tags:

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

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

```
!^http(s)?://\w*!
```

Quantifiers

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.

Matches string a US Social Security Number:

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

Matches a UK or Canadian Postal Code:

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

Greediness

Greediness Definition

By default the regex engine tries to match *all possible* characters that fit a pattern:

```
$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>"
```

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

```
$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

Pattern modifiers, one or more, follow the end delimiter as needed and affect the operation of the entire pattern:

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

Matches a string which starts with case insensitive A to Z:

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

Matches a UTF-8 string:

```
/à,à,£à,°.*/u
```


PERFORCE

PHP Perl Compatible Regular Expressions (PCRE) Functions

Perl Compatible Regular Expressions Functions

Perl-Compatible Regular Expressions (PCRE) functions are available with a companion function set for *mb_string*. The functions process PCRE pattern searches.

`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 found.

`preg_quote()`
Quote regular expression characters.

`preg_replace_callback_array()`, or `preg_replace_callback()`
Perform a regular expression search and replace using callbacks.

`preg_replace()`
Perform a regular expression search and replace.

`preg_split()`
Split string by a regular expression.

For a complete reference on PCRE see: <https://www.pcre.org/pcre.txt>.

PERFORCE

PHP Perl Compatible Regular Expressions Functions

preg_match(), preg_match_all()

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);
```

PERFORCE

PHP Perl Compatible Regular Expressions (PCRE) Functions

preg_replace_callback()

This pulls 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

PHP Perl Compatible Regular Expressions (PCRE) Functions

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.

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

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

To run at a terminal prompt after global installation:

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

Important Files

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.

Important Files

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.

Note: This file is JSON and strict formatting is required.

Important Files

composer.json (cont.)

Require Key

The require key is where the dependency specification lives:

```
"require": {  
    "php": ">=7.4",  
    "zendframework/zendframework": "2.3.*"  
}
```

Subkeys denote the dependency name.

Important Files

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 is could possibly break backward compatibility. Be specific with versioning constraints until test verifies compatibility.

Important Files

composer.json (cont.)

Expanded composer.json File

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

Important Files

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.

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 .

Packagist

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

Data Formats

XML

Responses to web service requests are formatted in a number of different ways. Extensible Markup Language (XML) is a tag-based format:

```
<?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>
```

Data Formats

JSON

JavaScript Object Notation (JSON) is popular due to its simplicity and represents a native JavaScript object:

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

Data Formats

YAML

YAML (Y Ain't Markup Language) is another format:

```
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

A number of parsing extensions for data formats exist:

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.

Data Formats

SimpleXML

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 string. All import techniques shown below return an instance of SimpleXMLElement. The SimpleXMLElement object and associated methods allow for obtaining parts of the XML as either other SimpleXMLElement objects, or as arrays:

The constructor accepts an XML parameter string:

```
SimpleXMLElement::__construct('...')
```

A function that accepts a file path containing XML:

```
simplexml_load_file('path/to/file.xml')
```

A function that accepts a string of XML:

```
simplexml_load_string('...')
```

Data Formats

Parsing with Object Notation

Loading and parsing:

```
// 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 accomplished with two functions:

`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 );
```


Data Formats

YAML Parsing

The YAML functions read or write YAML:

`yaml_parse()`:

Parses an YAML string into an array.

`yaml_emit()`:

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

```
$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 data APIs. Some services are free, some at a cost, some require registration and Application Programming Interface (API) keys, some do not.

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

- Google Maps APIs
- Ebay Developers Program
- Amazon Web Services
- Reddit Developers API

Using Web Services

To use a web service, make a 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:

Laminas\Http\Client

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

Requests

A web service request for data from a third party server will return formatted data that needs parsing:

The common web service tools:

Simple Object Access messaging Protocol (SOAP)

A request tool that includes both client and server classes.

Representational State Transfer (REST):

A service request over HTTP/HTTPS protocol using the HTTP verbs of GET, POST, PUT, PATCH AND DELETE.

XML Services

There are two primary tools for XML web services:

XML-RPC

Extensible Markup Language Remote Procedure Call.

SOAP

Simple Object Access Protocol

XML-RPC

The *XML-RPC* provides a client and server that facilitate remote procedure calls using HTTP as a transport, and XML as the data format.

The *laminas/laminas-xmlrpc* framework provides the two classes, and are available via *Composer* install.

PERFORCE

Web Services

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.

Web Services

Web Service Description Language (WSDL) Document

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>
```

Web Services

A SOAP Client

SOAP clients and servers should execute 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  
}
```

Web Services

SOAP Server

A SOAP server is a software server and not a stand alone web server. It requires a 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
}
```

REST Web Services

REST web services are relatively simple compared to SOAP services. They can be a simple URL request from the PHP function `file_get_contents()`, or from a client request API:

- 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.

PERFORCE

Web Services

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.

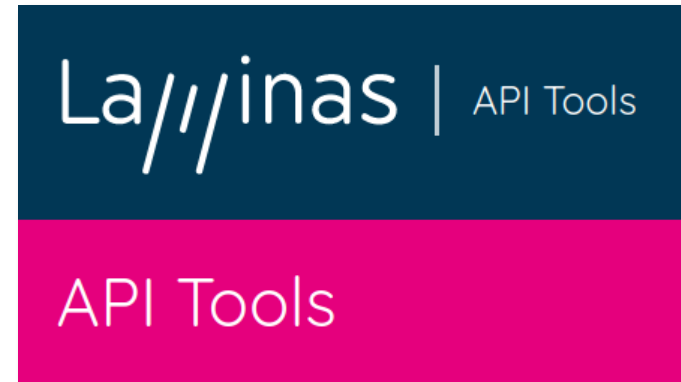
PATCH:

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 APIs™s check out the [Laminas API Tools](#).



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($response));
```

Web Services

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

Web Services

Lab: REST

Complete the following:

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.

This lab is complete.

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:

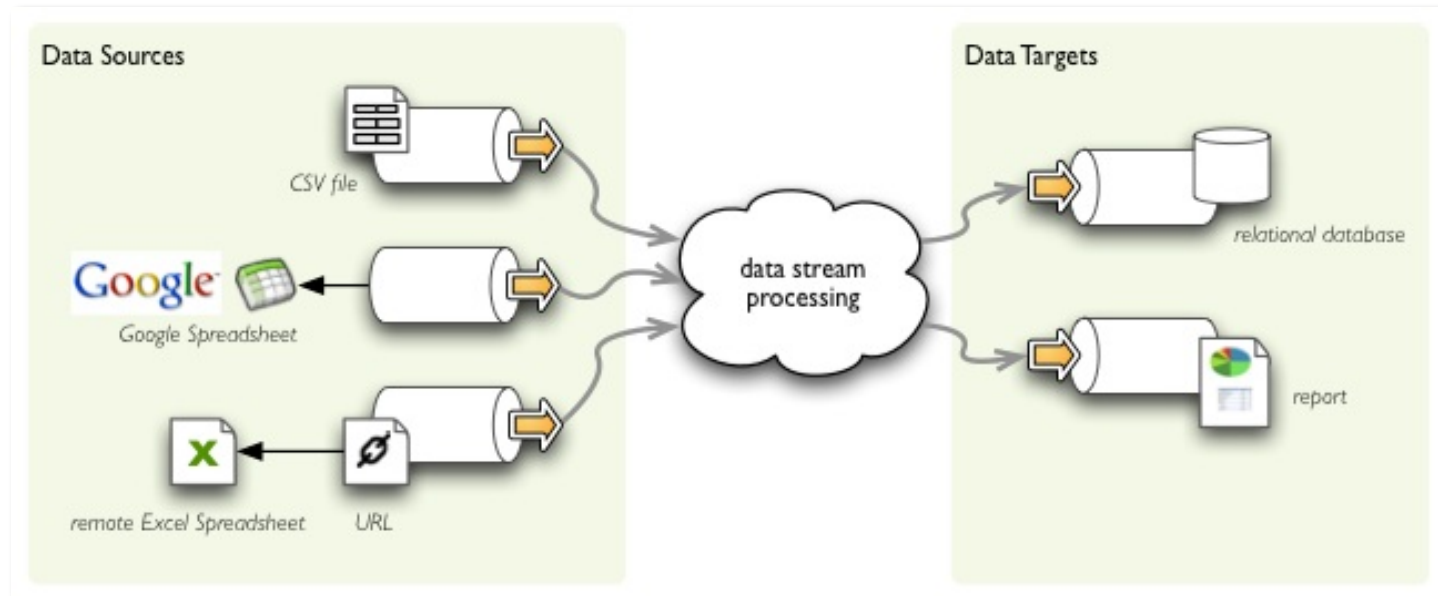
- Files
- Variables
- URLs
- Databases
- Memory

Streams Architecture

Streams 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.

Streams Architecture

Custom Wrapper Class and Runtime

This example shows a limited class functionality, other methods are available:

```
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;
    }
}
```

The runtime:

```
stream_wrapper_register('mystream', 'DataStream');
$data = " ";
$fh = fopen('mystream://data', 'r+');
fwrite($fh, 'some new stuff');
fclose( $fh );
```

Stream Socket Client

This is an example using the `stream_socket_client()` function:

```
// 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

Complete the following:

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.

This lab is complete.

Module Summary

This module covered:

- Data formats
- Web services
- Streams

Final Bindings

This module covers:

- PHP Unit
- Security
- Configuration
- Language Updates

PHP Unit

PHP Unit

PHP Unit is a software unit testing library for testing application code. A full understanding of unit testing and test driven design is beyond the scope of this class, but references are provided along with a simple example.

Unit testing:

- 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.

Test Class

A test class that tests the method unit of a *Domain* class:

```
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');
    }
}
```

Security

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

PERFORCE

Security

Validating Input

This code takes input and validates it with various functions:

```
// 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]];
```

Security

Validating Input

This code validates input, then escapes output:

```
// 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';
}
```

Security

Filtering Input

This code filters 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]
];
```

Security

Filtering Input

This code snippet iterates filters and escapes output:

```
// 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';
}
```

Security

Escaping Output

This code escapes output before sending a response:

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

print_r($goodtogo);
```

Configuration

Configuration

Directives

Directives define PHP behavior. There are a number of configuration directives found in a given *php.ini* file, and there are numerous *php.ini* files on a given server, one for each type of parser.

Each directive links to manual pages detailing the directive:

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: A web server configuration file will vary depending on the web server and OS. In Ubuntu Linux the main configuration is found at */etc/apache2/apache2.conf*. However, in CentOS linux, the main configuration defaults to */etc/httpd/conf/httpd.conf*. The *NGINX* web server configuration file most often default to */etc/nginx/nginx.conf*.



Changing Directives at Runtime

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`

Configuration

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	
	

PERFORCE

Configuration

"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
```

Configuration

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

Language Updates

PHP 8.1

The most significant PHP 8.1 updates have already been incorporated into the PHP Fundamentals I, II and III courses.

[PHP 8.1 Fibers](#) are covered in the PHP Fundamentals III course.

Here is a brief list of other OOP changes in PHP 8.1 not otherwise covered:

- First class *callable* syntax
- Properties and constants changes
- Resource to object migration

For more information see the [PHP 8 to 8.1 Migration Guide](#).

Callable Syntax

`Closure::fromCallable()` was introduced in PHP 7.1 as a way of creating Closure instances (i.e. anonymous functions) from something that's callable. Here's an example of that usage:

```
class Test {
    public function __construct(
        public string $first = "",
        public string $last = "" ) {}
    public function getFullName()
    {
        return $this->first . ' ' . $this->last;
    }
}

$test = new Test('Fred', 'Flintstone');
$anon = Closure::fromCallable([$test, 'getFullName']);
echo $anon();
// output: Fred Flintstone
```

Language Updates > Callable Syntax

First Class Callable Syntax

In PHP 8.1 you can now use the ... operator to achieve the same results as Closure::fromCallable().

```
class Test {  
    public function __construct(  
        public string $first = "",  
        public string $last = "") {}  
    public function getFullName()  
    {  
        return $this->first . ' ' . $this->last;  
    }  
}  
$test = new Test('Fred', 'Flintstone');  
$anon = $test->getFullName(...);  
echo $anon();  
// output: Fred Flintstone
```


Language Updates

Properties and Constants

PHP 8.1 adds support for readonly properties.

Constants can now be designated final.

define() now accepts objects as constants.

```
class Test {  
    define('OBJ', new ArrayObject([1,2,3]));  
    var_dump(OBJ->getArrayCopy());  
    // output:  
    /*  
    array(3) {  
        [0]=> int(1)  
        [1]=> int(2)  
        [2]=> int(3)  
    }  
    */  
}
```

Resource to Object Migration

PHP 8.1 continues the trend first initiated in PHP 8.0. Many extensions now return *objects* instead of *resources*. Affected extensions include: FileInfo, FTP, IMAP, LDAP, PostgreSQL and PSpell.

The list of extensions that now produce objects instead of resources in PHP 8.0 include: CURL, Enchant, GD, OpenSSL, XMLParser and XMLWriter.

This does not affect your code, however if you use `is_resource()` to check to see if the resource was created, your code will fail. Here's an example using CURL

```
$url = 'https://www.google.com';
$ch = curl_init();
if (!is_resource($ch)) exit('Unable to open connection');
curl_setopt($ch, CURLOPT_URL, $url);
curl_setopt($ch, CURLOPT_HEADER, 0);
$result = curl_exec($ch);
var_dump($result);
curl_close($ch);
// PHP 8.1 output: Unable to open connection
```

If you replace `is_resource()` with `!empty()`, the code will work in all versions of PHP.

Module Summary

This module covered:

- PHP Unit
- Security
- Configuration
- Language Updates

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

Perforce 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 Perforce Software courses' materials are provided "as is" and subject to any statutory warranties which cannot be excluded, Perforce 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 Perforce 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.