PHP Object Oriented Programming Quick Reference Guide

[Last Updated: Oct 29, 2018]

PHP supports OOPS programming paradigm which bundles data (properties) and related functions (methods) in independent units called objects.

Features/Syntax	Description/examples

class
A **class** is a blueprint from which we create object instances. A class mainly contains properties and methods.
Variables and functions are called properties and methods respectively in OOP world.
'new' keyword
To create an instance of a class, **new** keyword is used `new MyClass();`
pseudo-variable $this
The reference to the current object, only used within the class.
Object operator `->`
Used when calling a method or accessing a property on an object instance. It is used with `$this` as well.
	`<?php`
	`class MyClass {`
	`   // defining a property`
	`   public $myProperty = 'a property value';`
	
	`   // defining a method`
	`   public function display() {`
	`     echo $this->myProperty;`
	`   }`
	`}`
	
	`$var = new MyClass();`
	`$var->display();`
	`?>`
	Output:
	`a property value`

Visibility
The visibility keywords (*public, protected, private*) determines how an object properties/methods are accessible:
• **public**: can be accessed anywhere.
• **protected**: can be accessed by the class and subclasses only.
• **private**: can be accessed by the class only.
A property must be defined with one of the above visibility keywords.
A method defined without any of them will have public visibility by default.
	`<?php`
	`class MyClass {`
	`  public $var1 = 'public var';`
	`  protected $var2 = 'protected var';`
	`  private $var3 = 'private var';`
	
	`  function printHello() {`
	`    echo $this->var1 . '<br>';`
	`    echo $this->var2 . '<br>';`
	`    echo $this->var3 . '<br>';`
	`  }`
	`}`
	
	`$obj = new MyClass();`
	`echo $obj->var1 . '<br>'; // prints public var`
	`$obj->printHello(); // prints all`
	
	`// if uncommented followings will produce fatal errors`

Core Java Tutorials

- Java 16 Features
- Java 15 Features
- Java 14 Features
- Java 13 Features
- Java 12 Features
- Java 11 Features
- Java 10 Features
- Java 9 Module System
- Java 9 Misc Features
- Java 9 JShell

Recent Tutorials

- Spring - The Use Of PropertyEditors With @V Annotation
- Spring - The Use Of PropertyEditors With XM Configuration
- Spring - Use of PropertyEditors via BeanWra
- Spring - Using BeanUtils for resolving string based method signatures
- Spring - Copying properties using BeanUtils
- Spring - Obtaining BeanInfo And PropertyDescriptors
- Spring - Directly setting fields via DirectFieldAccess
- Spring - BeanWrapper, setting nested beans
- Spring - Using BeanWrapper
- Spring - Transforming events
- Spring - Conditional Event Handling via @EventListener annotation
- Spring - Using @EventListener annotation wi multiple event types
- Spring - Publishing and Consuming Custom Events
- Java - Introduction to ResourceBundle
- Spring - Injecting Resource using @Value annotation
- Spring - Injecting ResourceLoader
- Spring - Resource Loading
- Spring - Using Spring Expression Language v @Value Annotation
- Spring - Using @Value Annotation

```
/*
* echo $obj->var2; // Fatal Error
* echo $obj->var3; // Fatal Error
*/
```

Output:

```
public var
public var
protected var
private var
```

Class constants

Just like defining other constants (outside of a class) using 'const' keyword, we can also define them inside a class. The default visibility of class constants is public.

Constants are allocated once per class, and not for each class instance.

Scope Resolution Operator (::)

Instead of using **->**, double colon allows access to static and constant. This operator is also used to access super class features.

Using 'self'

Instead of using **$this**, the **self** keyword is used to access constants within the class. Generally, for all class level access **self** should be used and for all current object instance access **$this** should be used within the class.

```
<?php
class MyClass {
   const PI = 3.14159;

  function showPI() {
       echo self::PI . "<br>";
  }
}

echo MyClass::PI . "<br>";
$class = new MyClass();
$class->showPI();
echo $class::PI . "<br>";
?>
```

Output:

```
3.14159
3.14159
3.14159
```

static properties and methods

Class properties and methods can be declared with ***static*** keyword which makes them class level features and we don't need a class instance to access those features.

Like class constants we access static properties/methods with double colon (**::**) and to access them within class we use **self** keyword.

$this is not available inside a static method.

By default static features have 'public' accessibility.

For static variable initialization, the same rules apply as const expressions.

```
<?php
class MyClass {
  static $var = 'a static property';

  static function aMethod() {
    return self::$var;
  }
}

echo MyClass::$var . '<br>';
echo MyClass::aMethod();
?>
```

Output:

```
a static property
a static property
```

Constructors and destructors

A **constructor** function `__construct()` is a special (magic) function which is automatically called

```
 <?php
class MyClass {
   private $prop;
```

Spring - Injecting Environment to access properties in beans
Spring - Adding New Property Source to Environment
Spring - Adding user properties by using @PropertySource
Spring - Accessing Environment Properties
Spring - Profiles
Spring - Successfully injecting circular dependencies using @Lazy Annotation
Spring - Successfully injecting circular dependencies using setter injection
Java 16 - Records Features, Quick Walk-thro
Java 16 - Introduction to Records
Spring - Injecting beans into Arrays/Collectic Using @Qualifiers And Specifying the Orderir
Injecting Collections - Injecting Beans Into A And Collections, ordering with Ordered Interf
Spring - Injecting beans Into Arrays and List ordering with @Ordered annotation
Spring - Injecting beans into Arrays and Collections, selecting elements with @Qualifi annotation
Spring - Injecting multiple Beans Into Arrays Collections
Spring - Arrays and Collections As Beans
Spring - Using @ComponentScan#excludeFil to exclude classes from scanning based on annotations
Spring - Using @ComponentScan#includeFilt to scan non component classes based on annotations
Spring - Implementing ApplicationContextAw Interface
Spring - Using excludeFilters attribute of @ComponentScan to exclude component cla
Spring - Using @ComponentScan to scan no component classes via includeFilters attribut
Spring - Using Filters To Customize Scanning @ComponentScan
Spring - Using basePackageClasses Attribute @ComponentScan
Spring - Specifying packages to be scanned basePackages attribute of @ComponentScan
JUnit - How to test user command line Input Java?
Spring - Session based Prototype Bean Exam
Spring - Prototype Bean Example
Spring - Singleton Bean Example
Spring - Receiving lifecycle callbacks by implementing InitializingBean and DisposableBean
Spring - Receiving lifecycle callbacks by usin 'initMethod' and 'destroyMethod' of @Bean annotation
Spring - Implicit Constructor Injection In @Configuration Class
Spring - Using @Autowired annotation on arbitrary methods

when an object instance is created with 'new' keyword. A constructor can have any number of user defined parameters. Constructors should ideally be used to initialize object

A **destructor** is a special function `__destruct()` which is automatically called when object is deleted from memory by garbage collector.

Objects are deleted as soon as no references of them used in the program, or during the shutdown sequence.

Destructors are typically used for cleanup, state persistence, etc.

```
    function __construct($var) {
      echo 'In constructor of ' . __CLASS__ . '<br>';
      $this->prop = $var;
    }

    public function displayProp() {
      echo $this->prop .'<br>';
    }

    function __destruct() {
      echo 'destroying ' . __CLASS__;
    }
}

$a = new MyClass('the property value');
$a->displayProp();
?>
```

Output

```
In constructor of MyClass
the property value
destroying MyClass
```

Inheritance:

Inheritance is the process of extending an existing class (parent class) by a new class (subclass) using **extends** keywords.

A subclass inherits all properties and methods from it's super (parent) class except for private ones.

Inheritance is used for code reuse and polymorphism.

PHP allows single inheritance (at most one super class)

```
<?php
class MyParentClass {
  protected $var = 'a super call property';

  public function display() {
     echo $this->var . '<br>';
  }
}
class MySubclass extends MyParentClass {
  public function getVar() {
     return 'returning ' . $this->var;
  }
}

$a = new MySubClass();
$a->display();
echo $a->getVar();
?>
```

Output:

```
a super call property
returning a super call property
```

Inheritance and Construct/destruct

Parent class constructor/destructor are not implicitly called if the child class defines them as well. In order to run a parent constructor (or destructor), a call to **parent::__construct()** within the child constructor is required. If the child does not define a constructor then it may be inherited from the parent class just like a normal class method.

parent keyword

```
<?php
class A {
    protected static $x = 'value x';
    function __construct() {
      echo 'In constructor of ' . __CLASS__ . '<br>';
    }
}
class B extends A {
    function __construct() {
      parent::__construct();
      echo parent::$x .'<br>';
      echo 'In constructor of ' . __CLASS__ . '<br>';
```

Spring - Using @Inject annotation on setter methods
Spring Core - Using @Autowire annotation O setter method
Spring - Defining Injection point by using @I annotation
Spring - Defining injection point by using @Autowire annotation
Spring - Resolving ambiguity by using @Inje and @Named annotations
Spring - Resolving ambiguity by using @Inje and @Qualifier Annotations
Spring - Autowiring By Name, Using Autowire.BY_NAME
Spring - Autowiring By Type mode, Using Autowire.BY_TYPE
Spring - Default Auto-wiring mode, Autowire Example
Elements of @Bean Annotation
Spring - Using Multiple @Configuration Class
Spring - Dependency Injection In @Configur Classes
Reactor - Programmatically generate Flux via Consumer<SynchronousSink<T>>
Spring Boot - Testing With @SpyBean
Mockito - Creating Spy Of Real Objects
Reactor - Creating Flux Instance From Iterab
Reactor - Create Flux Instance From Array
Reactor - Creating Flux Instance Which Emits Range Of Integer
Reactor - Creating Flux and Mono with empty
Java - Find Files in classpath under a Folder SubFolder
Java - How to find enum by ordinal?
Java 15 - Sealed Classes
Spring Boot Primefaces Integration
Spring Boot JSF Integration
Reactor - Creating an instance using Flux.jus Mono.just()
Spring Boot - Testing With @MockBean
Java - How to delete old files under a folder i number of files are over a specified limit?
Mockito - BDD Style Verification using then() should()
Mockito - BDD Style Stubbing with given() - willReturn()
Cassandra - Mapping Java Objects using Map
Java - How to convert Calendar to LocalDateTime?
Mockito - verifyNoMoreInteractions() and verifyNoInteractions()
Mockito - Verifying Multiple Number of Metho Invocations
Mockito - Verify Method Calls With Argument Matchers
Kafka - ConsumerRebalanceListener Example
Kafka - Understanding Partition Rebalancing

Similar to 'self' keyword which is used to access class level static and constant features, ***parent*** is used to access super class level features from subclass that includes construct/destruct and overridden methods (next topic).

```
  }
}

$b = new B();
```

Output:

```
In constructor of A
value x
In constructor of B
```

Method overriding

Method overriding is the process where a subclass redefine a parent class method to change it's behavior. The signature should exactly be the same.

In case if we want to access parent level features from within a subclass then we will use **parent::**

```
<?php
class A {
  function aMethod() {
    return "aMethod from A";
  }
}

class B extends A {
  function aMethod() {
       return "aMethod from B, ".
       parent::aMethod();
  }
}

$a = new A;
echo($a->aMethod());
echo('<br>');
$b = new B;
echo($b->aMethod());
?>
```

Output

```
aMethod from A
aMethod from B, aMethod from A
```

Abstract classes

An abstract class cannot be instantiated. They provide base class abstract implementation to be extended to provide some specific behavior.

An abstract thing doesn't exist in reality e.g. an animal doesn't exists as a concrete thing but an specialization e.g. dog does exist.

An abstract class definition starts with **abstract** keyword.

An abstract class can defined method(s) without body. In that case, we have to use **abstract** keyword in the method signature.

A subclass must override abstract methods or otherwise should be 'abstract' themselves.

```
<?php
abstract class A{
  abstract protected function aMethod();

  public function doSomething(){
    $this->aMethod();
  }
}
class B extends A{
  protected function aMethod(){
    echo 'aMethod called';
  }
}

$b = new B();
$b->doSomething();
```

Output:

```
aMethod called
```

Kafka Manual Commit - commitSync() Examp

Kafka Manual Commit - CommitAsync With Callback and Specified Offset

Kafka Manual Commit - commitAsync With Callback Example

Java - Introduction to Java 8 Date and Time

Reactor - Understanding Flux/Mono's retryW

Reactor - Retrying Flux/Mono Sequence

Spring - Injecting Prototype Bean Using Java Functions

Mockito - Argument Matchers Example

Mockito - Argument Matchers

Kafka Manual Commit - CommitAsync() Exar

Java - How to Indent multiline String?

Mockito - Verifying Method Calls

Spring MVC - Post Request With Simple Form Submission

Java - Parsing String To Numeric Primitives

Mockito - Stubbing methods with exceptions

Java - Avoiding possible NullPointerExceptior method call chain

Java - Autoboxing And Unboxing, How to avc NullPointerException with unboxing?

Kafka - Auto Committing Offsets

Kafka - Understanding Offset Commits

Kafka - Publishing records With null keys anc assigned partitions

Java Collections - How to find frequency of e element in a collection?

How to convert java.util.Map To Java Bean?

Java - How to repeat a string n number of tir

Git - Merging Branches

Java - How to convert Iterator To List?

Spring Boot - Unit Testing Application Argum

How to find the longest and the shortest Stri Java?

How to find first and last element of Java 8 stream?

Mockito - Stubbing consecutive method calls

Mockito - Stubbing a method's return Value

Kafka - Using Keys For Partition Assignment

Spring Boot - Using @TestConfiguration In st nested class

Spring Boot - Using @TestConfiguration to de beans for tests

Java Collections - Why Arrays.asList() does r work for primitive arrays?

Java Collections - Only put Map key/value if specified key does not exist

Getting Started with Mockito

How to connect a Database server in Intellij Community Edition?

Java HashMap - Understanding equals() and hashCode() methods

Java IO - How to write lines To a file and rea lines from a files?

Interfaces

Like an abstract class, Interface is another way to define abstract type.

Interfaces define methods without implementation. They don't have to use abstract keyword.

To define an interface we use **interface** keyword instead of class.

All methods in an interface should be public.

Class implementing interfaces

A class may implement one or more comma separated interfaces by using **implements** keyword. The class must implement all interface methods or otherwise should be declared abstract.

Interfaces can extend each other by using keyword **extends**

```
<?php
interface Task {
  public function runTask();
  public function anotherMethod();
}

abstract class TaskImpl implements Task {
  public function runTask() {
    echo $this->anotherMethod();
  }
}

class TaskImpl2 extends TaskImpl {
  public function anotherMethod() {
    return "another method running task ";
  }
}

$task = new TaskImpl2();
$task->runTask();
?>
```

Output:

```
another method running task
```

Traits

A Trait is intended to reduce some limitations of PHP single inheritance model. A developer can reuse sets of methods freely in several independent classes living in different class hierarchies..

1. A trait is defined using keyword **trait**. It is similar to a class, but cannot be instantiated it's own.

```
trait MyTrait {
...
}
```

2. A trait can have properties and magic methods. It can also have static properties and methods.
3. To use them in a class use keyword **use** followed by the trait name. A class can have multiple comma separated traits.

```
class MyClass{
 use MyTrait1, MyTrait2;
 ...
}
```

4. In case of conflicts, the class overrides trait functions, but if those functions are defined in a parent class of the current class then traits overrides them.
5. If two traits have a method with the same name, a fatal error is

```
<?php
trait  ATrait{
  private $var;

  function __construct($str) {
   $this->var = $str;
  }

  public function aTraitFunction() {
   echo __METHOD__ . ", from ATrait:  $this-
>var<br>";
  }

  public function aTraitFunction2() {
   echo __METHOD__ . ", from ATrait:  $this-
>var<br>";
  }

  function __toString() {
   return __CLASS__ . ", var=$this->var <br>";
  }
}

trait BTrait{
  public function aTraitFunction2() {
   echo __METHOD__ . ", from BTrait:  $this-
>var<br>";
  }
}

class BClass {
  public function aTraitFunction2() {
   echo __METHOD__ . ", from BClass:  $this-
>var<br>";
```

Java Collections - How to find distinct elemen count in collections and arrays?
Spring Boot - Web Application Testing With Embedded Server And TestRestTemplate
Java - How to find Available Runtime Memory
Kafka - Understanding Consumer Group with examples
Spring Boot - Unit Testing Web Application W Embedded Server
Java - Different ways to Set Nested Field Valu By Reflection
Java - Different ways to Set Field Value by Reflection
Installing Python 2.7 on windows
Installing Cassandra And Intro To CQLSH
Cassandra - Getting Started with Java
Java 14 - Switch Expressions And Statement Examples
Kafka - Manually Assign Partition To A Consu
Kafka - Understanding Topic Partitions
Spring Boot - Unit Testing Web Application W Mock MVC
Kafka - Introduction to Kafka Admin API
Java 14 - Helpful NullPointerException
Java 14 - Pattern Matching for instanceof
Spring Boot - Application Unit Testing with @SpringBootTest
Installing and Running Kafka
Kafka - Getting Started
Spring Boot - Different Ways To Pass Applica Properties
Reactor - Transforming into Publishers and Delaying any Error with flatMapDelayError() flatMapSequentialDelayError()
Reactor - Transforming into Publisher and Maintaining the source order with flatMapSequential()
Reactor - Transforming into Publishers and th flattening by using flatMap operation
Reactor - Using transform Operation
Reactor - Mapping items
Reactor - Getting Started
Java 13 - Text Blocks (JEP 355)
Spring Cloud - Hystrix CircuitBreaker, Thread Local Context Propagation
Spring Cloud - Circuit Breaker Hystrix Event Listener
Spring Cloud - Circuit Breaker Hystrix, Chang Default Thread Pool Properties
Spring Cloud - Circuit Breaker Hystrix, concu requests and default thread pool size
Spring Cloud - Circuit Breaker, Specifying Hy configuration in application.properties file
Spring Cloud - Hystrix Circuit Breaker, Settin Configuration Properties Using @HystrixProp
Spring Cloud - Hystrix Circuit Breaker, gettin failure exception in fallback method

produced. We can resolve the conflict by using **insteadof** operator:

```
class MyClass{
 user MyTrait1, MyTrait2{
  MyTrait2::aMethod insteadof
MyTrait1;
 }
  ...
}
```

6. Since **insteadof** operator excludes one of the method, we can re-include the excluded method by using **as** operator which gives an alias to the method.

```
class MyClass{
   use MyTrait1, MyTrait2{
    MyTrait2::aMethod insteadof
MyTrait1;
    MyTrait1::aMethod as
otherMethod;
   }
}
$a = new MyClass();
$a->aMethod();//calls
MyTrait2::aMethod
$a->otherMethod();//calls
MyTrait1::aMethod
```

7. We can also change the visibility of a method using **as** operator:

```
class MyClass{
 use MyTrait {
 aMethod as private
aPrivateMethod;
 }
}
```

8. A trait can use other traits too:

```
trait MyTrait{
 use MyOtherTrait;
}
```

9. A trait can have abstract methods too. In that case classes using them have to implement the method.

```
  }
}

class AClass extends BClass {
  use ATrait, BTrait{
   BTrait::aTraitFunction2 insteadof ATrait;
  }

  public function aTraitFunction() {
   echo __METHOD__ . ", frome AClass: $this-
>var<br>";
  }
}

$a = new AClass('a string');
echo $a;
$a->aTraitFunction();
$a->aTraitFunction2();
```

Output:

```
AClass, var=a string
AClass::aTraitFunction, frome AClass: a string
BTrait::aTraitFunction2, from BTrait: a string
```

Final keyword

A method declared with **final** keyword cannot be overridden by a subclass.

A class declared with **final** keyword cannot be extended.

```
class A {
  final function display() {
   echo "displaying in " . __CLASS__;
  }
}
class B extends A {
  function display() {
   echo "displaying in " . __CLASS__;
  }
}
```

Spring Cloud - Circuit Breaker Hystrix Basics
TypeScript - Standard JavaScript's built-in ob Support
JavaScript - Async Processing with JavaScrip Promise
TypeScript - Applying Mixins
JPA Criteria API - Modifying and Reusing Que Objects
JPA Criteria API - Delete Operations
JPA Criteria API - Update Operations
JPA Criteria API - Case Expressions with CriteriaBuilder.selectCase()
JPA Criteria API - Calling Database Function CriteriaBuilder.function()
Java 12 - Getting JFileChooser shortcuts pan files with FileSystemView# getChooserShortcutPanelFiles()
Java 12 - CompletionStage's new methods fc exception handling
Java 12 - Comparing files with Files.mismatc
Java 12 - Composing Collectors with Collectors.teeing()
Java 12 - Compact Number Formatting suppo
Java 12 - String Changes
Java 12 - Switch Expression (JEP 325)
Git - Understanding HEAD
Git - Creating and switching branches using 'branch' and 'checkout' commands
TypeScript - Parameter Decorator
TypeScript - Property Decorators
TypeScript - Accessor Decorators
Jackson - JSON to Java tree model
Installing MongoDB On Windows 10 and Gett started with MongoDB Compass
MongoDb - Getting Started with Java
JPA Criteria API - CriteriaBuilder Date Time Operations
JPA Criteria API - Collection Operations size(index()
JPA Criteria API - CriteriaBuilder Arithmetic Operations
JPA Criteria API - CriteriaBuilder String Manipulation Methods
TypeScript - Method Decorator
Obtaining Property Descriptor by using Object.getOwnPropertyDescriptor()
Git - Hosting and Accessing Remote Reposito over SSH
Groovy - Class File Vs Script File
Groovy - Automatic JavaBean properties
Groovy - Using @PackageScope for package-private visibility
Groovy - Default Visibility of Classes and Met
Groovy - Imports
Groovy - Operator Overloading
Groovy Operators - Call Operator

```
$b = new B();
$b->display();
```

Output:

```
Fatal error: Cannot override final method A::display() in
```

```
final class A {
}
class B extends A {
}

$b = new B();
```

Output:

```
Fatal error: Class B may not inherit from final class (A)
```

Objects equality

- The comparison operator `==` compares the objects by their property values and by types. Two objects are equal if they are instances of the same class and have same property values and, even though they are different references.
- The identity operator `===` compares the object by their references. The object variables are identical if and only if they refer to the same instance of the same class.

```
<?php
class MyClass {
}

function toString($bool) {
  if ($bool === false) {
   return 'FALSE';
  } else {
   return 'TRUE';
  }
}

$a = new MyClass();
$b = new MyClass();
$c = $a;

echo '<br>$a==$b : ' . toString($a == $b);
echo '<br>$a===$b : ' . toString($a === $b);
echo '<br>$a==$c : ' . toString($a == $c);
echo '<br>$a===$c : ' . toString($a === $c);
```

Output:

```
$a==$b : TRUE
$a===$b : FALSE
$a==$c : TRUE
$a===$c : TRUE
```

Object iteration

By default all visible properties can be iterated using **foreach** loop.

To control iteration process we can implement built-in Iterator interface.

```
<?php
class TestClass {
  public $str = 'a str property';
  public $int = 3;
  protected $str2 = "a protected str property";
  private $str3 = "a private str property";

  function display(){
    foreach($this as $key => $value){
      echo "$key => $value <br>";
    }
  }
```

Getting Started with GPU programming using
Aparapi Framework

Extract files from Windows 10 Backup image
Mounting/Attaching VHD/VHDX

Jackson JSON - Deep merging with @JsonMe
Annotation

Jackson JSON - Updating Existing objects wit
JSON input during Deserialization

Linux - What is the superuser home dir?

Java CompletableFuture - Understanding
CompletionStage.whenComplete() method

Java - Converting FileTime To Formatted Stri
and vice versa

JPA Criteria API - Using all(), any(), some()
methods of CriteriaBuilder

Git - Remote repository basics

Spring Cloud - Spring Cloud Config with Git
Backend

Groovy Operators - Coercion Operator

Groovy Operators - Identity Operator

Groovy Operators - Membership Operator

Upload Maven Artifacts On Remote Server vi
without any Repository Manager

Java CompletableFuture - Exception Handlin

Java IO - Copy Directories In Parallel

JPA Criteria API - Using CriteriaBuilder.exists
Method

How to apply Java Regex to any Command L
Output?

Git - Using Git in Intellij IDE

Installing Windows On Multiple Computers w
single RETAIL License Key

Java Command Line - Sending Command Inp
To Java via command line pipe

Java Swing - Adjusting Right Click JPopupMe
position when Shift + F10 is Pressed

Java Swing - Prevent JTree from Collapsing

TypeScript - Class Decorators

TypeScript - ES6 Modules

TypeScript - Developing TypeScript with Visu
Studio Code IDE

How to completely uninstall/remove Visual S
Code IDE?

Groovy Operators - Subscript Operator

Groovy Operators - Spaceship Operator

Groovy Operators - Range Operator

Groovy Operators - Spread Operator

Git - Using Git in Eclipse IDE (EGit plugin)

JPA Criteria API - Using Subquery.correlate()
method

JPA Criteria API - Subqueries

JPA Criteria API - GroupBy And Having meth
of CriteriaQuery

JPA Criteria API - Aggregate Methods

Spring Data JPA - Using @CreatedBy And
@LastModified to audit who created and cha
an entity.

```
}

$a = new TestClass();
$a->display();
echo '----<br>';
foreach($a as $key => $value){
        echo "$key => $value <br>";
}
```

Output:

```
str => a str property
int => 3
str2 => a protected str property
str3 => a private str property
----
str => a str property
int => 3
```

Auto-loading Classes

For a typical PHP application, a developer has to use multiple include statements to include multiple files. Per good practice, there should be a single class in a file.

spl_autoload_register function registers any number of autoloaders, enabling for classes and interfaces to be automatically loaded if they are currently not defined. By registering autoloaders, PHP is given a last chance to load the class or interface before it fails with an error.

Assume class A resides in A.php and class B resides in B.php at the same location as this Test.php script:

```
<?php
spl_autoload_register(function ($className) {
        echo "including $className.php <br>";
         include $className . '.php';
});

$a = new A();
$b = new B();
```

```
including A.php
including B.php
```

Magic methods

These special methods get called in response to a particular event or scenario. We just have to implement them in our class. These methods starts with double underscores. We have seen two magic methods above already: **__construct()** and **__destruct()**.

Here's the complete list: **__construct()**, **__destruct()__toString()**, **__set()**, **__get()**, **__isset()**, **__unset()**, **__call()**, **__callStatic() __invoke()**, **__set_state()**, **__clone()**, **__sleep()**, **__wakeup()**, and **__debugInfo()**

we are going to explore them in the following sections.

String representation of objects

__toString()

This method should return a string. It's a way to represent an object as a string.

```
<?php
class MyClass{

  private $var = 'a property';

  public function __toString(){
    return "MyClass: var = '$this->var'";
  }
}

$a = new MyClass();
echo $a;
```

Output:

JPA Criteria API - Downcasting with CriteriaBuilder.treat() methods
TypeScript - Auto Refresh TypeScript project browser With LiveReload
TypeScript - Setting up TypeScript project wi tsconfig.json
JavaScript Modules - Declaring a Module To c Export Other Modules
Java CompletableFuture - Selecting Either Of Two Completion Stages to execute a new Completion Stage
Groovy Operators - Regular Expression Opera
Groovy Operators - Method Pointer Operator
Java Stack Walking - How to find name of th current method?
Groovy Operators - Direct Field Access Opera
Groovy Operators - Safe Navigation Operator
Groovy Operators - Elvis Operator
Groovy Operators - Power Operator
Groovy - Using Collections and Arrays
Git - Ignoring Files
Git - Specifying custom Git Repository and W Tree locations
Git - Configurations
JavaScript Modules - Transitively Re-Exportin Modules
JavaScript Modules - Different ways to Expor
JavaScript Modules - Empty Import for runni global code
JavaScript Modules - Combining Default impo With Namespace Or Named Imports
Spring Data JPA - Implementing Basic Auditi using @CreatedDate and @LastModifiedDate
JavaScript Modules - Single Default Export
JavaScript Modules - Named Imports
Java Swing - JMenu Search Highlighting
JPA Criteria API - Applying in() Predicate
JPA Criteria API - Applying isNull() and isNotl Predicate
Jackson JSON - Using @ConstructorPropertie deserialize JSON via constructor
Git - Using Bash Shell for Windows
Spring Data JPA - Applying Pessimistic Lockir with @Lock Annotation
Spring - Understanding the correct use of Declarative Transaction
Spring Data JPA - Optimistic Locking
JavaScript - ES6 Generator Functions
JavaScript - Creating Custom Iterable by app ES6 Iterable and Iterator Protocols
JavaScript - ES6 for-of loop
JavaScript - ES6 Enhanced Object Literals
JavaScript - Symbol Type
TypeScript - Discriminated Unions
Spring MVC - Data Binding with Immutable E Classes in Spring 5

```
MyClass: var = 'a property'
```

Property Overloading

__set(), __get(), __isset(), __unset()

This is a way to dynamically create object properties which do not exist or are inaccessible.

- **__set()**: is triggered when an inaccessible property is assigned a value.
- **__get()**: is triggered when an inaccessible property is accessed.
- **__isset()**: is triggered when calling built-in functions isset() or empty() on inaccessible properties..
- **__unset()**: is triggered when calling built-in function unset() on inaccessible properties.

```
<?php
class MyClass {
  private $arr;

  public function __set($name, $value) {
   $this->arr[$name] = $value;
  }

  public function __get($key) {
   if (array_key_exists($key, $this->arr)) {
    return $this->arr[$key];
   }
   return null;
  }

  public function __isset($name) {
   if (isset($this->arr[$name])) {
    echo "Property $name is set.<br>";
   } else {
    echo "Property $name is not set.<br>";
   }
  }

  public function __unset($name) {
   unset($this->arr[$name]);
   echo "$name is unset <br>";
  }
}
$myObj = new MyClass();
$myObj->name="joe";// will trigger __set
var_dump($myObj);
echo '<br>';
echo $myObj->name; //will trigger __get
echo '<br>';
isset($myObj->name);//will trigger __isset
unset($myObj->name);//will trigger __unset
var_dump($myObj);
?>
```

Output:

```
object(MyClass)#1 (1) { ["arr":"MyClass":private]=> array(
joe
Property name is set.
name is unset
object(MyClass)#1 (1) { ["arr":"MyClass":private]=> array(
```

Method overloading

__call(), __callStatic

This is a way to dynamically create object methods which do not exist or are inaccessible.

- **__call()**: is triggered when calling an inaccessible method of an object.

```
<?php
class MyClass {

  public function __call($name, $paramArr) {
    // do something based on method name and params
    return "<br>returning result of method: $name ,
params: "
        . print_r($paramArr, true) . "<br>";
        }
```

- Using Spring 5 BeanDefinitionCustomizer and new methods in GenericApplicationContext to programmatically register beans
- Spring - Using ObjectProvider to Inject short lived Scoped Bean into longer-lived Scoped B
- Groovy with Intellij
- Groovy with Eclipse
- Groovy - Using Numbers
- Groovy - String Quotes
- Groovy - Data Type and Declaring Variables
- Groovy - Optional Features
- Groovy - Methods In Scripts
- Getting Started with Groovy
- JPA Criteria API - Applying like() and notLike Predicate
- Gradle - Getting Started
- Java CompletableFuture - Composing Comple Stages
- Spring - Java 8 Stream support in ObjectProv to retrieve beans stream
- Spring - Java 8 functions support in ObjectProvider to provide callbacks
- Spring - Programmatic Lookup of dependenc via ObjectProvider
- Spring Boot - Profile Specific Properties with spring.profiles.include
- Spring MVC - Setting Active Profiles Programmatically
- Spring MVC - Setting Active Profile
- Spring Boot - Programmatically Setting Profil
- Spring Boot - Adding active profiles unconditionally by using spring.profiles.inclu property
- Git - Quick Start
- Installing Git on Windows
- TypeScript - Using never Type
- TypeScript - Generic Type Aliases
- TypeScript - Type Aliases
- TypeScript - Non-Null Assertion Operator !

- **_callStatic()**: is triggered when calling an inaccessible static method methods of a class.

```
        public static function __callStatic($name,
$paramArr) {
    // do something based on method name and params
    return "<br>returning result of static method:
$name , params: "
      . print_r($paramArr, true) . "<br>";
        }
}
$a = new MyClass();
echo $a->someMethod("some arg");
echo MyClass::someStaticMethod("some arg for static
method");
?>
```

Output:

```
returning result of method: someMethod , params: Array ( [

returning result of static method: someStaticMethod , para
```

Calling object as a function

__invoke() method is triggered when a script tries to call the target object as an function.

It can be used to pass a class instance that can act as a closure, or simply as a function that we can pass around, and can invoke it without knowing if it's an object or without knowing what object it is.

```
<?php
class MyClass {

  public function __invoke() {
   //do something on being invoked as a function
    echo "invoked triggered<br>";
  }
}

function myFunction(Callable $func) {
  $func();
  return "returning from myFunction";
}

$a = new MyClass();
//$a() this also works
echo myFunction($a);
```

Output:

```
invoked triggered
returning from myFunction
```

Object Cloning

__clone() method of a newly cloned object is triggered just after it has been cloned by using **clone** keyword.

__clone() method allows any necessary properties that need to be changed after cloning.

When an object is cloned by using **clone** keyword, PHP performs a shallow copy of all of the object's properties. Any properties that are internal object references will not be cloned and will remain references. Using __clone(), we can explicitly

```
<?php
class MyClass {
  public $var;

  public function __construct() {
   $this->var = new MyOtherClass();
  }

  public function __clone() {
   $this->var = clone $this->var;
  }
}
class MyOtherClass {
  private $str = 'some string';

  function getStr() {
```

cloned internal objects too. There can be an recursive _clone() method calls to achieve deep cloning.

```
    return $this->str;
  }
}

function toString($bool) {
  if ($bool === false) {
   return 'FALSE';
  } else {
   return 'TRUE';
  }
}
$a = new MyClass();
$b = clone $a;
//using identity operator === to see two '$var' have the same
references
echo 'are internal objects equal: ' . toString($a-
>var === $b->var);
```

Output:

```
are internal objects equal: FALSE
```

removing __clone() method of MyClass will give output of TRUE, confirming that two $var are still pointing to same reference.

Serialization

__sleep() and __wakeup()

__sleep() method triggered when serialize() function is invoke on the target object.

serialize generates a storable string representation of a value.

To deserialize string to PHP type we use unserialize() function. In that case the magic method **__wakeup()** is invoked on the target object.

The purpose of these two magic methods is to control the ongoing serialization process. __sleep() method returns an array of property names to be serialized. If this method is not defined then all properties are serialized.

```
<?php
class MyClass {
  public $date;
  public $str;
  function __sleep() {
    return array('date');
  }

  function __wakeup() {
    echo "wakeup triggered <br>";
  }
}

date_default_timezone_set("America/Chicago");
$a = new MyClass();
$a->date = date("y/m/d");
$s = serialize($a);

echo "serialized: <br>";
echo $s;
echo "<br>";

echo "unserializing<br>";
$b = unserialize($s);
echo 'unserialized object:';
echo var_dump($b);
echo "<br>";
echo 'unserialized date: ' . $b->date;
```

Output:

```
serialized:
O:7:"MyClass":1:{s:4:"date";s:8:"16/10/25";}
unserializing
wakeup triggered
```

```
unserialized object:object(MyClass)#2 (2) { ["date"]=> str
unserialized date: 16/10/25
```

Exporting variables

The magic method **__set_state()** of a class is produced in the exported string when we use **var_export()** function on the instance of the target object.

A variable can be exported to a parsable string representation by using var_export() function. This method is similar to var_dump() but the difference is that var_export() returns a valid PHP code. Which we can import in our program and can reuse it later. For example:

```php
<?php
class A{
  public $prop = 4;
}
echo var_export(new A());
```

Output:

```
A::__set_state(array(
   'prop' => 4,
))
```

Exported var returns a form which would expect a static method call **__set_state()** on object A and pass the exported object property array. That happens when we try to convert it to PHP variable again. We are supposed to define __set_state() method in our class and recreate the object instance from the array.

One way to reuse the exported string to manually copy paste the code and use it. Another way is to use the function eval() , which is not recommended.

```php
<?php
class TestClass {
  public $var = 4;

  public static function __set_state($arr) {
    $a = new TestClass();
    $a->var = $arr['var'];
    return $a;
 }
}

$a = new TestClass();
$v = var_export($a, true);
echo "exported variable:<br> $v <br>";
echo "<br>importing variable:<br>";
eval('$e=' . $v . ';');
var_dump($e);
```

Output:

```
exported variable:
TestClass::__set_state(array( 'var' => 4, ))

importing variable:
object(TestClass)#2 (1) { ["var"]=> int(4) }
```

Controlling debug info

The magic method **__debugInfo()** is triggered when we use var_dump() on the target object.

If this method isn't defined on an object, then all public, protected and private properties will be shown. If it is defined, then it should return an associative array of properties which we want to show in var_dump() call.

```php
class TestClass {
         private $var = 10;
         private $str = 'a string';

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

$a = new TestClass();
var_dump($a);
```

Output:

```
object(TestClass)#1 (1) { ["string"]=> string(8) "a string
```

See Also

PHP Language Reference	PHP language Quick Reference Guide
Classes/Object Functions	PHP language Quick Reference Guide 2

Core Java Tutorials

Java 16 Features	Java 11 Features
Java 15 Features	Java 10 Features
Java 14 Features	Java 9 Module System
Java 13 Features	Java 9 Misc Features
Java 12 Features	Java 9 JShell

Recent Tutorials

Spring - The Use Of PropertyEditors With @Value Annotation	Spring - Transforming events
Spring - The Use Of PropertyEditors With XML Configuration	Spring - Conditional Event Handling via @EventListener annotation
Spring - Use of PropertyEditors via BeanWrapper	Spring - Using @EventListener annotation with multiple event types
Spring - Using BeanUtils for resolving string based method signatures	Spring - Publishing and Consuming Custom Events
Spring - Copying properties using BeanUtils	Java - Introduction to ResourceBundle
Spring - Obtaining BeanInfo And PropertyDescriptors	Spring - Injecting Resource using @Value annotation
Spring - Directly setting fields via DirectFieldAccess	Spring - Injecting ResourceLoader
Spring - BeanWrapper, setting nested beans	Spring - Resource Loading
Spring - Using BeanWrapper	

Share

Previous Page Next Pag