PHP ESSENTIALS #9

By WI400 Team

: (O)bject (O)riented (P)rogramming

Concetti OOP

- ideazione e evoluzione
- definizione di classe
- uso di una classe
- PHP e OOP

Ideazione ed evoluzione

- La programmazione ad oggetti nasce negli anni ’60 con l’aumento della complessità dei programmi
- Aumento della complessità implica un aumento della difficoltà di mantenimento
- La programmazione procedurale diviene sempre più inadeguata all’aumentare delle righe di codice
- Il codice procedurale viene suddiviso in unità logiche di programmazione (funzioni)
- Con la programmazione ad oggetti le funzioni vengono eliminate dal programma principale e vengono raggruppate in unità specializzate chiamate classi

Vantaggi

- La programmazione ad oggetti ha tre principali vantaggi:

- **Riusabilità** del codice grazie alla suddivisione di attività complesse in moduli generici

- **Manutenibilità** del codice in quanto la gestione di una singola attività all'interno di un metodo facilita l'identificazione e l'eliminazione degli errori

- **Affidabilità** derivante dalla struttura modulare del codice. Se viene rilevato un problema, sarà necessario risolvere il problema nel solo metodo che gestisce quell'attività

Definizione di Classe

- Il concetto fondamentale di qualsiasi linguaggio ad oggetti è la classe.

- Una **classe** è una collezione di variabili e di funzioni che agiscono su tali variabili

- Una variabile relativa ad una classe prende il nome di **proprietà** della classe

```
echo $myobject->myproperty;
```

- Una funzione relativa ad una classe prende il nome di **metodo** della classe

```
$myobject->mymethod();
```

Uso di una Classe

- Per poter utilizzare una classe precedentemente definita è necessario creare un'istanza della classe da utilizzare.

```php
<?php
class myclass
{
    public $myproperty;
    function mymethod()
    {
        echo 'You just run my method!';
    }
}

$classe= new myclass();
```

- L'istanza di una classe (oggetto) permette l'accesso ai metodi ed alle proprietà definite all'interno della classe.
- L'accesso alle proprietà ed ai metodi di una classe può essere effettuato attraverso l'operatore -> associato all'istanza

  $classe->metodo();

  $classe->proprietà;

PHP e programmazione ad oggetti

- Il PHP non nasce come linguaggio OOP
- Il supporto alla programmazione ad oggetti è stato aggiunto a partire dal PHP 3 con la carenza di molti elementi essenziali
- Con il PHP 4 sono state affrontate le carenze relative al PHP 3 mantenendo però l'attenzione maggiore sulla retrocompatibilità
- Il PHP 5 ha apportato un cambiamento radicale alla gestione della programmazione ad oggetti.
- La maggiore differenza tra il PHP 4 ed il PHP 5 riguardo la programmazione ad oggetti è il passaggio degli oggetti per riferimento e non più per valore

PHP 4 vs PHP 5

- Completa gestione dell'incapsulamento (metodi e proprietà di tipo public, private e protected)
- Unificazione dei nomi dei costruttori. Mentre in PHP 4 il costruttore aveva il nome della classe, in PHP 5 il costruttore ha il nome __construct()
- Supporto alla liberazione esplicita delle risorse attraverso il distruttore: __destruct()
- Classi non estendibili (FINAL)
- Classi, proprietà e metodi statici
- Caricamento automatico delle classi

Creazione di un classe-PHP5

- Ogni proprietà ed ogni metodo devono essere preceduti da una delle seguenti parole chiave:
  - ✓ public: visibile sia dall'interno che dall'esterno della classe
  - ✓ protected: visibile solo dall'interno della classe (e dalle classi che la estendono)
  - ✓ private: visibile solo dall'interno della classe che definiscono il metodo o la proprietà

Creazione di un classe-PHP5

```php
<?php
class myclass
{
    public $myproperty;
    public $other_property;

    function mymethod()
    {
        echo 'You just run my method!';
    }
}

$classe= new myclass();
```

```php
echo $classe->myproperty;
```

fare attenzione a non inserire il simbolo del $ quando si accede ad una proprietà della classe

Imparare la visibilità

- La visibilità è un concetto molto importante per una corretta programmazione ad oggetti

```php
<?php
class Visibilita {
    public $_public = 'pubblica';
    protected $_protected = 'protetta';
    private $_private = 'privata';
}

$istanza = new Visibilita ( );
echo $istanza->_public; // proprietà  pubblica
echo $istanza->_protected; // Fatal error:
echo $istanza->_private;   // Fatal error:
```

- L'accesso alle proprietà private e protected non è consentito e produrrà quindi un fatal error

Imparare la visibilità

- Per accedere alle proprietà di tipo private e protected dall'esterno della classe è necessario definire dei metodi.

```php
<?php
class Dir_Visibilita {
    public $_public = 'pubblica';
    protected $_protected = 'protetta';
    private $_private = 'privata';

    public function getProtectedProperty() {
        return $this->_protected; // $this è questa classe
    }
}

$istanza = new Dir_Visibilita ( );
echo $istanza->getProtectedProperty ();
```

- Il metodo deve essere di tipo public in quanto deve essere visibile all'esterno della classe

Riepilogo

- Una **proprietà** è una **variabile** definita all'interno di una classe
- Tutte le proprietà vanno definite come **public**, **protected** o **private**
- Un **metodo** è una **funzione** definita all'interno di una classe
- La **visibilità** per i metodi è opzionale ma è buona norma non tralasciarla. Nel caso in cui venga tralasciata il metodo viene automaticamente trattato come **public**.
- La parola chiave **new** permette la creazione di una istanza di una classe (**oggetto**)
- Dall'interno della classe è possibile accedere alle proprietà ed ai metodi della classe stessa attraverso la variabile **$this**
- L'operatore **->** associato all'istanza di una classe permette l'accesso alle proprietà ed ai metodi della classe istanziata
- Quando si accede alle **proprietà** di una classe attraverso l'operatore **->** è necessario omettere il simbolo del dollaro prima della proprietà
- L'accesso a proprietà/metodi di tipo **private** o **protected** dall'esterno della classe genera un **fatal error**

Esercizio 17:

- crea una classe myclass con:
  - una proprietà: $myproperty
  - un metodo: mymethod()

- il metodo dovrebbe effettuare l'output della proprietà. La sintassi è:
  echo $this->myproperty;

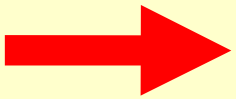
- "testa" il funzionamento della classe attraverso:
  - instance della classe
  - imposta un valore alla proprietà
  - richiamo del metodo

Esercizio 17: solution

- creazione della classe con definizione di una proprietà
- definizione del metodo per effettuare echo della proprietà

```php
<?php
class myclass
{
    public $myproperty;

    function mymethod()
    {
        echo $this->myproperty;
    }
}
$myobject = new myclass;
$myobject->myproperty = 'Hello, World!';
$myobject->mymethod();
```



Hello, World!

Costruttore

- Per maggiore flessibilità è possibile definire le proprietà nel momento in cui viene istanziato l'oggetto
- Per ottenere questo risultato si ha bisogno del costruttore dell'oggetto
- Il costruttore è il metodo che viene richiamato automaticamente nel momento in cui viene istanziato l'oggetto
- Come qualsiasi metodo può ricevere valori in input
- Il costruttore può restituire anche un valore in output *
- I valori saranno passati al momento in cui viene istanziata la classe

```
$istanza = new Classe ( $var1, $var2, $var3 );
```

- Per convenzione il costruttore viene definito subito dopo la definizione delle proprietà della classe

Costruttore – PHP5

- Nel PHP 5 il costruttore è definito attraverso il metodo __construct

Nota: nel PHP 4 il metodo costruttore aveva lo stesso nome della classe

```php
<?php
class Classe {
    public $_val1;
    protected $_val2;
    private $_val3;

    public function __construct($val1, $val2, $val3) {
        $this->_val1 = $val1;
        $this->_val2 = $val2;
        $this->_val3 = $val3;
    }
}

$istanza = new Classe ( $param1, $param2, $param3 );
```

- Per motivi di retrocompatibilità, nel caso in cui non è definito un metodo __construct, il PHP cerca un eventuale metodo che abbia lo stesso nome della classe che si sta istanziando.

Costruttore: nota importante

- Nel PHP 5 se una classe ha sia il costruttore __construct() che il metodo con lo stesso nome della classe definiti, solo il metodo __construct() verrà richiamato in fase di creazione dell'oggetto

- In questo caso verrà generato anche un errore di tipo E_STRICT in quanto più costruttori sono definiti all'interno della classe

Esercizio 18:

- Creare una classe chiamata “Folder”

- Il costruttore della classe deve ricevere in ingresso e salvare in una proprietà il percorso della cartella da gestire

Esercizio 18: solution

```php
<?php
class Folder {
     private $_directory;
     public function __construct($directory) {
     $this->_directory =
     (substr ($directory, -1)==DIRECTORY_SEPARATOR) ? substr($directory, 0, -1) : $directory;
     }
}
```

Metodi Magici

- I metodi magici di PHP possono essere riconosciuti dal fatto che iniziano con il doppio carattere di underscore (__)

- I metodi magici vengono automaticamente invocati al verificarsi di determinati eventi e per questo sono chiamati magici

- Fino ad ora abbiamo visto il metodo magico __construct() che viene invocato al momento in cui viene instanziata la classe alla quale appartiene

__toString()

- Il metodo __toString() permette la conversione di un oggetto in stringa
- Se si prova ad utilizzare echo o print associati ad un oggetto si ottiene un errore
- Potrebbe essere utile avere la possibilità di visualizzare un oggetto come fosse una stringa
- Il metodo magico __toString() permette di definire cosa visualizzare
- Il metodo deve necessariamente restituire una stringa

```php
<?php
class Animale {
    private $_specie;

    public function __toString() {
        return (string)$this->_specie;
    }
}
```

__clone()

- A partire dal PHP 5 gli oggetti non vengono più passati per valore ma per riferimento

```php
<?php
$animale = new fish_class ( 'trota' );
$copiaAnimale = $animale;
```

- Si avrà lo stesso oggetto nelle due variabili.
  Questo significa che entrambe puntano allo stesso oggetto. Ogni modifica ad una proprietà di $animale significherebbe riportare la modifica anche a $copiaAnimale

- Il PHP 5 ha introdotto quindi l'operatore clone che serve a creare una copia di un oggetto:

```php
$copiaAnimale = clone $animale;
```

- Il metodo magico __clone() viene automaticamente richiamato al momento in cui viene eseguita una clonazione di un oggetto tramite l'operatore clone

- E' utile se occorre apportare delle modifiche all'oggetto "copia" lasciando invariato l'oggetto originario

__get()

- Il metodo magico __get() fornisce un meccanismo di accesso virtuale alle proprietà di una classe
- E' molto utile se si vuole evitare di scrivere i metodi per l'accesso in lettura ad ogni singola proprietà di tipo protected o private.
- Permette quindi l'accesso automatico in lettura alle proprietà di una classe
- Il metodo viene richiamato solo se si tenta di accedere ad una proprietà che non esiste
- Il nome passato al metodo __get() è case sensitive

__get()

```php
<?php
class MakePassword {
    public $_public;
    protected $_protected;
    private $_private;

    function __get($name) {
        if ($name == 'md5')
            return substr ( md5 ( rand () ), 0, 8 );
        else if ($name == 'sha1')
            return substr ( sha1 ( rand () ), 0, 8 );
        else
            return rand ();
    }
}

$a = new makePassword ( );
echo $a->md5; // Verrà richiamato il metodo __get()
echo $a->_public; // NON richiamerà  il metodo __get()
echo $a->_protected; // Verrà  richiamato il metodo __get()
echo $a->_private;  // Verrà richiamato il metodo __get()
```

__set()

- Anche il metodo magico __set() fornisce un meccanismo di accesso virtuale alle proprietà di una classe
- E' molto utile se si vuole evitare di scrivere i metodi per l'accesso in scrittura ad ogni singola proprietà di tipo protected o private.
- Permette quindi l'accesso automatico in scrittura alle proprietà di una classe
- Il metodo viene richiamato solo se si tenta di accedere ad una proprietà che non esiste
- Il nome passato al metodo __set() è case sensitive

__set()

- Un esempio di utilizzo del metodo __set() potrebbe essere il seguente:

```php
<?php
class Set_Class {
    function __set($nome, $valore) {
        echo "il valore assegnato a $nome è $valore";
    }
}
$utente = new Set_Class ( );
$utente->password = md5 ( 'miapassword' );
```

__call()

- Il metodo magico __call() fornisce un meccanismo di accesso virtuale ai metodi non definiti all'interno della classe
- Il metodo viene richiamato solo se si tenta di accedere ad un metodo non definito all'interno della classe

```php
<?php
class math {
    function __call($name, $arg) {
        if (count ( $arg ) > 2)
            return FALSE;
        switch ($name) {
            case 'add' :
                return $arg [0] + $arg [1];
                break;
            case 'sub' :
                return $arg [0] - $arg [1];
                break;
            case 'div' :
                return $arg [0] / $arg [1];
                break;
        }
    }
}
```

__sleep()

- La serializzazione è il processo attraverso il quale una variabile PHP viene trasformata in una stringa codificata che può essere successivamente utilizzata per ricreare la variabile originaria
- Il metodo magico __sleep() viene automaticamente richiamato in fase di serializzazione di un oggetto
- La serializzazione si rende necessaria per i dati complessi come oggetti ed array che non possono essere scritti semplicemente in file o memorizzati in database
- Il processo di serializzazione viene eseguito attraverso la funzione serialize()

```php
<?php
$animale = new fish_class ( 'trota' );
$serializzato = serialize ( $animale );
echo $serializzato;
```

→ `O:7:"animale":2:{s:7:"*_pesce";N;s:7:"*_specie";s:5:"trota";}`

- __sleep() viene generalmente utilizzato per specificare le proprietà da serializzare in quanto difficilmente si ha la necessità di serializzare l'intero oggetto

__wakeup()

- Il metodo magico __wakeup() viene automaticamente richiamato in fase di deserializzazione di un oggetto
- La deserializzazione è il processo inverso rispetto alla serializzazione e consiste nel trasformare un oggetto serializzato nell'oggetto originario

```
<?php
$animale = new fish_class ( 'trota' );
$serializzato = serialize ( $animale ); // Serializzazione
$deserializzato = unserialize ( $serializzato );// Deserializzazione
```

- Il processo di deserializzazione viene eseguito attraverso la funzione unserialize()
- __wakeup() viene generalmente utilizzato per ricreare le proprietà che non sono state serializzate

__wakeup()

- Un esempio di utilizzo del metodo __wakeup() potrebbe essere il seguente:

```php
<?php
class test {
    public $foo = 1;
    public $bar;
    public $baz;

    function __construct() {
        $this->bar = $this->foo * 10;
        $this->baz = ($this->bar + 3) / 2;
    }
    function __wakeup() {
        // Ricostruisco le proprietà  non serializzate
        $this->__construct ();
    }
}
```

__destruct()

- Generalmente il PHP distrugge automaticamente variabili ed oggetti dal momento in cui questi non sono più necessari allo script
- Potrebbe essere necessario eseguire alcune operazioni prima che l'oggetto venga distrutto
- Il metodo magico __destruct() viene automaticamente richiamato nel momento in cui viene richiesta la distruzione dell'oggetto

```php
<?php

class myclass {

    public function __destruct() {

        // Operazioni da eseguire
        // prima che l'oggetto venga distrutto
    }
}
```

__destruct()

- Un esempio di utilizzo del metodo __destruct() potrebbe essere il seguente:

```php
<?php
class fileio {
    private $fp;
    function __construct($file) {
        $this->fp = fopen ( $file, "w" );
    }
    function __destruct() {
        // Forzo la scrittura su file
        // di tutti i dati presenti nel buffer
        fflush ( $this->fp );
        // Chiudo il file
        fclose ( $this->fp );
    }
}
```

Esercizio 19:

- Nella classe Folder definita in precedenza:
- implementare il metodo __get() per gestire le proprietà virtuali:
  - ✓ _lastAccess
  - ✓ _lastModified
- Implementare il metodo __toString() in modo che ritorni il nome della cartella in uso senza il percorso

Esercizio 19: solution

```php
<?php
public function __get($name) {
    switch ($name) {
        case '_lastAccess' :
            $this->_lastAccess = fileatime ( $this->_directory );
            return date ( "d-m-Y H:i:s", $this->_lastAccess );
            break;
        case '_lastModified' :
            $this->_lastModified = filemtime ( $this->_directory );
            return date ( "d-m-Y H:i:s", $this->_lastModified );
            break;
        default :
            return FALSE;
            break;
    }
}
public function __toString() {
    $directories = explode ( DIRECTORY_SEPARATOR, $this->_directory );
    return ( string ) array_pop ( $directories );
}

$folder = new Folder ( "path" . DIRECTORY_SEPARATOR ."htdocs");
echo "Ultimo accesso: $folder->_lastAccess<br>";
echo "Ultima modifica: $folder->_lastModified<br>";
```

Ereditarietà

- Uno degli aspetti più importanti della programmazione ad oggetti è la possibilità di costruire una classe sulla base di un'altra classe
- Derivare una classe da un'altra già esistente significa ereditare le proprietà ed i metodi della classe originaria
- E' possibile accedere alle proprietà ed ai metodi della classe padre. Sono quindi visibili i seguenti tipi di proprietà/metodi:
  - ✓ public
  - ✓ protected
- L'ereditarietà è molto importante al fine di una buona organizzazione e un facile riutilizzo del codice
- Per derivare una classe esistente è sufficiente utilizzare la parola extends:

```
<?php
class Child extends Parent {
    //............
}
```

Ereditarietà: esempio

- L'idea è quella di avere una classe (padre) generica “Animale”, e tante classi specifiche (estese) per le singole esigenze

```php
<?php
class Animale {
    private $_specie;
    public function __construct($specie) {
        $this->_specie = $specie;
    }
    public function getSpecie() {
        return $this->_specie;
    }
    public function setSpecie($specie) {
        $this->_specie = $specie;
    }
}
```

```php
class Mammifero extends Animale {
    private $_corna;
    public function __construct($specie) {
        $this->setSpecie ( $specie );
    }
    public function hasCorna() {
        return $this->_corna;
    }
    public function setCorna($corna) {
        $this->_corna = $corna;
    }
}
```

Override

- L'ereditarietà permette quindi di ereditare proprietà e metodi da una classe.
- E' possibile tuttavia ridefinire (override) i metodi e/o le proprietà ereditate.
- Nelle classi precedenti ad esempio è stato eseguito l'override del metodo __construct e cioè del costruttore. Questo significa che il costruttore della classe padre non verrà mai eseguito nel momento in cui vengono istanziate le classi

```
class Mammifero extends Animale {
    private $_corna;
    public function __construct($specie) {
        $this->setSpecie ( $specie );
    }
    public function hasCorna() {
        return $this->_corna;
    }
    public function setCorna($corna) {
        $this->_corna = $corna;
    }
}
```

Controllo dell'Override

- E' possibile controllare l'override sia sulle classi che sui metodi di una classe
- Il controllo dell'override di un metodo o di una classe può essere fatto attraverso l'utilizzo della parola final:
- Impedire l'override di una classe:

```
final class Animale {

}
```

- Impedire l'override di un metodo:

```
class Animale {
    final public function getSpecie() {

    }
}
```

Costanti

- Non è possibile utilizzare la parola chiave final con le proprietà di una classe
- E' possibile tuttavia evitare l'override delle proprietà definendole come costanti
- Anche nella programmazione ad oggetti una costante è un valore che non cambia mai
- Le costanti non iniziano mai con il simbolo del dollaro
- Per convenzione le costanti vengono generalmente definite con caratteri maiuscoli
- E' possibile definire una costante all'interno di una classe tramite la parola chiave **const**

```
class Costante {
    const PGRECO = 3.141592653589793;
}
```

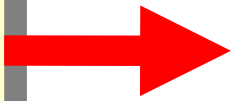
- Le costanti definite all'interno di una classe sono visibili alla classe e a tutte le classi derivate
- `self::PGRECO;` per accedere alla costante PGRECO anche se definita nella classe padre.
- E' possibile accedere ad una costante anche dall'esterno della classe senza instanziare la classe stessa: `classe::PGRECO`

Costanti

- Il valore di una costante non può essere modificato all'interno della classe che la definisce
- Tuttavia è possibile ridefinire la costante all'interno delle classi figlie

```php
<?php
class Costanti {
    const COSTANTE = 5;
}
class Ridefinisce extends Costanti {
    const COSTANTE = 10;
}

echo Costanti::COSTANTE;
echo Ridefinisce::COSTANTE;
```



```
5
10
```

- Se si vuole avere la sicurezza che un valore non venga mai cambiato nel tempo è necessario utilizzare le costanti globali come viene generalmente fatto in PHP:

```php
define('COSTANTE', 5);
```

Riepilogo

- Per estendere (e quindi ereditarne proprietà e metodi) una classe utilizzare la parola chiave extends seguita dal nome della classe da estendere
- Non è possibile estendere le classi definite come final
- Una classe può avere solo un padre (parent) ma può avere più di un figlio (child)
- Una classe figlia ha l'accesso a tutte le proprieta e a tutti i metodi della classe padre ad eccezione di quelli definiti come private
- E' possibile ridefinire proprietà e metodi della classe padre ad eccezione di quelli definiti come final
- E' possibile definire le costanti all'interno di una classe attraverso la parola chiave const

Autoload

- Per poter utilizzare una classe, e quindi crearne un'istanza, è necessario avere l'accesso alla classe. Questo viene fatto includendo la classe all'interno dello script, generalmente attraverso l'istruzione require_once() del PHP
- Il PHP fornisce anche un meccanismo di caricamento automatico delle classi istanziate:

```
function __autoload($classe) {
    // Codice da eseguire per includere la classe istanziata
}
```

- Questo implica l'adozione di una convenzione per quanto riguarda i nomi e le directory di salvataggio delle classi
- Supponendo di salvare ogni classe in un file e di assegnare al file lo stesso nome della classe, la funzione __autoload() potrebbe essere la seguente:

```
function __autoload($classe) {
    require_once $classe . '.php';
}
```

- Questo significa che se abbiamo la classe 'Animale' salvata in un file chiamato 'Animale.php', nel momento in cui viene istanziata la classe 'Animale' verrà fatto prima il require_once del file 'Animale.php' e poi verrà istanziata la classe 'Animale'.

InstanceOf

- Al fine di poter controllare il flusso del programma è necessario sapere di quale classe è un'istanza un oggetto
- Il PHP mette a disposizione l'operatore instanceof() per controllare se un oggetto è un'istanza di una determinata classe:

```
if($animale instanceof Animale)
// restituirà  true se $animale è un'istanza
//della classe Animale altrimenti restituirà false
```

- oppure è possibile utilizzare anche la funzione is_subclass_of() nel seguente modo:

```
is_subclass_of($animale, 'Animale');
```

- Per determinare invece a quale classe un oggetto appartiene è possibile utilizzare la funzione get_class():

```
$animale = new Animale('trota');
echo get_class($animale);
// visualizzerà  Animale
```

- Per determinare la classe parent è possibile utilizzare la funzione get_parent_class():

```
echo get_parent_class($animale);
// visualizzerà  Animale
```

Type Hinting

- Per sua natura il PHP è un linguaggio insensibile ai tipi di dati che possono essere contentuti ad esempio in una variabile
- Per semplificare le ambiguità causate da questa funzionalità il PHP consente di specificare i tipi di oggetti che ci si aspetta di ricevere come argomenti di funzione
- Il passaggio di un oggetto diverso da quello definito genererà un "fatal error"

```
<?php
function NomeFunzione(TipoRichiesto $nomeArgomento)

function TestTypeHintingObject(Mammifero $mammifero) {
    // L'argomento $mammifero dovrà  essere un'istanza
    // della classe Mammifero
}
function TestTypeHintingArray(array $argomento) {
    // $argomento dovrà essere un array
}
```

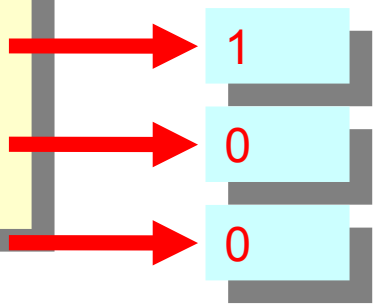
Comparazione di oggetti (==)

- Se si utilizza l'operatore di confronto (==) tra due oggetti, questi sono considerati uguali se hanno gli stessi attributi e valori e sono istanze della stessa classe:

```php
<?php
class A {
    public $foo = 1;
}
$a = new A ( );
$b = new A ( );
$c = new A ( );
$c->foo = 2;

$d = new StdClass ( );

echo ( int ) ($a == $b);
echo ( int ) ($a == $c);
echo ( int ) ($a == $d);
```

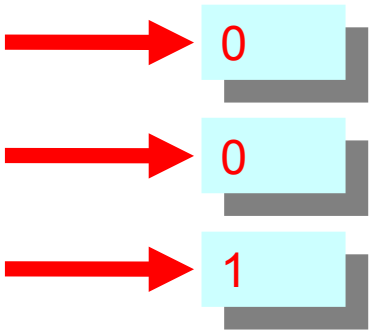


Comparazione di oggetti (===)

- Se si utilizza l'operatore di confronto (===) tra due oggetti, questi sono considerati identici se e solo se fanno riferimento alla stessa istanza di un oggetto:

```php
<?php
class A {
    public $foo = 1;
}
$a = new A ( );
$b = new A ( );
$c = clone $b;
$d = $a;

echo ( int ) ($a === $b);
echo ( int ) ($a === $c);
echo ( int ) ($a === $d);
```



Errori ed Eccezioni

- Il PHP5 ha introdotto un nuovo modo di gestione degli errori noti come eccezioni
- Un'eccezione (exception) viene generata quando qualcosa non funziona correttamente in un blocco di codice
- E' possibile generare un'eccezione attraverso la parola chiave throw

```
if($i <= 0)
 throw new Exception('Messaggio eccezione', [codice eccezione]);
```

- Invece di gestire l'errore nel punto in cui è stato generato, l'eccezione viene passata e gestita in un blocco di codice speciale
- In questo modo si ha il vantaggio di gestire tutti gli errori in un unico punto del codice
- Il PHP ha una classe nativa Exception che può essere utilizzata così, oppure può essere estesa per creare una gestione degli errori personalizzata

Errori ed Eccezioni

- Per catturare eventuali errori che si possono generare all'interno di un blocco di codice, è necessario racchiudere il codice da "controllare" all'interno del costrutto **try**:

```
try {
    // Codice da "controllare"
}
```

- Ogni blocco **try** deve avere almeno un blocco **catch** che si occuperà di catturare l'eccezione rilevata

```
catch ( Exception $e ) {
    // $e sarà  un oggetto di tipo Exception
}
```

- Quindi è possibile gestire eventuali eccezioni con il seguente codice:

```
try {
    // Codice da "controllare"
} catch ( Exception $e ) {
    // $e sarà  un oggetto di tipo Exception
}
```

Errori ed Eccezioni

- Quando viene catturata un'eccezione, il codice immediatamente successivo alla riga di codice che ha generato l'eccezione non viene eseguito
- Viene cercato il primo blocco **catch** "**compatibile**" con l'eccezione che è stata generata
- Se non esiste un blocco catch "**compatibile**" verrà generato un PHP Fatal Error con il messaggio Uncaught Exception a meno che non sia stato definito un gestore delle eccezioni con set_exception_handler()
- Per ogni blocco **try** deve esistere almeno un blocco catch
- Possono essere definiti più blocchi catch in base al tipo di eccezione che si vuole catturare

```
<?php
try {
    // Codice da "controllare"
} catch ( SoapFault $soapFault ) {
    // Gestisco l'errore SOAP
    } catch (SQLiteException $sqLite) {
        // Gestisco l'errore SQLite
        } catch (Exception) {
            // Gestisco l'errore generico
            }
```

Classe nativa: Exception

- Il PHP ha la classe nativa Exception per la gestione degli errori:

```
class Exception {
    protected $message = 'Unknown exception'; // Messaggio
    protected $code = 0; // Codice definito dall'utente
    protected $file; // Nome del file che ha generato l'eccezione
    protected $line; // Numero della riga di codice che ha generato l'eccezione


    function __construct($message = null, $code = 0);
    final function getMessage(); // Messaggio dell'eccezione
    final function getCode(); // Codice dell'eccezione
    final function getFile(); // Nome del file che ha generato l'eccezione
    final function getLine(); // Numero della riga di codice che ha generato l'ecc.
    final function getTrace(); // Array di backtrace
    final function getTraceAsString(); // Array di backtrace restituito come stringa
    function __toString(); // Stringa da visualizzare
}
```

Classe nativa: Exception

- Un esempio di utilizzo della classe nativa Exception potrebbe essere il seguente:

```php
<?php
try {
    $handle = fopen ( 'c:/miofile.txt', 'w' );
    if (! $handle) {
        throw new Exception ( 'Impossibile aprire il file.' );
    }
    if (fwrite ( $handle, abc ) != 3) {
        throw new Exception ( 'Impossibile scrivere sul file.' );
    }
    if (! fclose ( $handle )) {
        throw new Exception ( 'Impossibile chiudere il file.' );
    }
} catch ( Exception $e ) {
    printf ( "Errore in %s:%d %s" . PHP_EOL,
                $e->getFile (), $e->getLine (), $e->getMessage () );
}
```

Estendere la classe Exception

- Il modo migliore di utilizzare la classe Exception è quello di creare una classe che estende la classe nativa
- La classe nativa Exception può essere infatti estesa come tutte le altre classi non definite come FINAL

```php
<?php
class myException extends Exception {
    public function __construct($messaggio, $codice = 0) {
        parent::__construct ( $messaggio, $codice );
    }
    public function __toString() {
        return sprintf ( "Errore in [%s:%d]: %s" . PHP_EOL,
                $this->file, $this->line, $this->message );
    }
}

try {
    throw new myException ( 'Messaggio di errore' );
} catch ( myException $e ) {
    echo $e; // Messaggio di errore
}
```

Exception Handler

- E’ possibile catturare le eccezioni anche senza il costrutto try-catch:

```
function gestioneErrori($eccezione) {
    echo $eccezione;
}
// Imposta gestioneErrori come exception handler
set_exception_handler ( 'gestioneErrori' );
$handle = fopen ( 'c:/miofile.txt', 'w' );
if (! $handle) {
    throw new myException ( 'Impossibile aprire il file.' );
}
if (fwrite ( $handle, 'abc' ) != 3) {
    throw new myException ( 'Impossibile scrivere sul file.' );
}
if (! fclose ( $handle )) {
    throw new myException ( 'Impossibile chiudere il file.' );
}
```

- Se sono stati definiti più exception handler è possibile ripristinare il precedente handler attraverso la funzione

  restore_exception_handler()

```
set_exception_handler ( 'gestioneErrori1' );
set_exception_handler ( 'gestioneErrori2' );
// Ripristina l’exception handler a gestioneErrori1
restore_exception_handler ();
```

QUESTION TIME ?

Nome_____ Cognome_____ Data_____

ARRIVEDERCI

TITOLO