PHP: OOP

Pasqualetti Veronica

Oggetti

Possiamo pensare ad un **oggetto** come ad un tipo di dato più complesso e personalizzato, non esistente fra i tipi tradizionali di **PHP**, ma creato da noi.

Gli oggetti sono formati principalmente da **attributi** e **metodi**.

Gli **attributi** sono delle variabili proprietarie dell'oggetto, semplici o complesse, quindi anche Array o **Oggetti**.

I **metodi** invece, sono delle funzioni proprietarie dell'oggetto, e fra questi ce ne sono due molto importanti i **Costruttori** e i **Distruttori**.

Oggetti

Se non definirete un **costruttore** e un **distruttore** per la vostra **classe** (oggetto), PHP li rimpiazzerà con dei metodi propri di default.

Il **costruttore** verrà chiamato automaticamente da PHP, ogni volta che verrà creato un oggetto (istanza), mentre il **distruttore** sarà chiamato quando l'oggetto verrà distrutto, ossia quando non esiste più alcun riferimento all'oggetto oppure alla fine dello script.

Per dichiarare un costruttore, è sufficiente creare una funzione all'interno della classe che abbia lo stesso nome di quest'ultima, oppure potete usare la parola chiave **__construct()**.

Oggetti: esempio Persona.php


```
<?php
    class Persona
    {
            public $nome = ""; // attributo
            public $cognome = ""; // attributo
            public $datanascita = ""; // attributo

            public function Persona($n_nome, $n_cognome, $n_data) // costruttore
            {
                    $this->nome = $n_nome;
                    $this->cognome = $n_cognome;
                    $this->datanascita = $n_data;
            }

            public function stampaPersona() // metodo
            {
                    echo "Nome : " . $this->nome . "<br />\n";
                    echo "Cognome : " . $this->cognome . "<br />\n";
                    echo "Data di nascita : " . $this->datanascita . "<br />\n";
            }
    }

?>
```

Persona.php - commenti

- Il nome del file è uguale a quello della classe, così come lo è il nome del costruttore.
  Il costruttore prende in input tre parametri ($n_nome, $n_cognome, $n_data) che andrà a memorizzare rispettivamente nei propri attributi ($nome, $cognome, $datanascita), a cui accederà tramite la parola chiave **this**.

- Con **this** l'oggetto può richiamare i suoi attributi e metodi, in quanto **this** indica l'oggetto stesso.
  Subito dopo **this** segue l'operatore di selezione **->** che punta ad un determinato attributo o metodo alla sua destra, appartenente all'oggetto alla sua sinistra (this).

Persona.php - commenti

- Il simbolo del dollaro **$**, va solo su **this** e non sul nome dell'attributo/metodo.

- Infine troviamo il metodo **stampaPersona()** che semplicemente stampa gli attributi dell'oggetto tramite il costrutto **echo**.

Persona.php - utilizzo

test.php

```
<?php require_once("Persona.php"); ?>
<html>
    <head>
            <title>Test</title>
    </head>
    <body>
    <?php

            $utente = new Persona("Mario", "Rossi", "10-01-1980");
            $utente->stampaPersona();
    ?>
    </body>
</html>
```

Test.php - commenti

- Per creare una nuova istanza della classe **"Persona"**, abbiamo usato la parola chiave **new** seguita dal costruttore della classe con i rispettivi parametri.

- A seguire richiamiamo il metodo **"stampaPersona()"** con l'operatore di selezione **->** descritto in precedenza.

Test.php - commenti

- Osservando la classe **"Persona.php"**, possiamo notare che tutti gli attributi, il costruttore e il metodo, sono dichiarati con la parola chiave **public**.

- Questo ci consente di usarli e richiamarli liberamente nello script attraverso un'istanza della suddetta classe.

  La parola chiave **public** è un modificatore di accesso, che serve a stabilire che tipo di restrizioni si devono avere lavorando su quel dato.

Public, Protected e Private

- **public** - I membri (attributi o metodi) della classe dichiarati **public**, possono essere utilizzati sia all'interno che all'esterno della classe madre e di quelle derivate da essa (**ereditarietà**)
  (**$this->membro** oppure **$oggetto->membro**)

- **protected** - I membri dichiarati **protected**, possono essere utilizzati solo all'interno delle classi madri e derivate
  (**$this->membro**)

- **private** - I membri dichiarati **private**, possono essere utilizzati solo all'interno della classe madre
  (**$this->membro**)

Public, Protected e Private

- Tornando alla classe **"Persona"**, poichè abbiamo dichiarato gli attributi come **public**, il linguaggio ci consente di agire direttamente su di essi, senza dover richiamare il costruttore ogni volta che dobbiamo modificare un attributo dell'istanza (**$utente**) :

```
$utente = new Persona("John", "Doe", "1-1-1970");
$utente->nome = "Santiago";
$utente->cognome = "Arnavisca";
$utente->stampaPersona();
// output = Nome : Santiago / Cognome : Arnavisca / Data di nascita : 1-1-1970
```

Public, Protected e Private

- Se invece avessimo usato la restrizione **private** per gli attributi della classe **"Persona"**, avremmo ricevuto dal server un errore sollevato dalla riga $utente->nome = "Santiago";
- Avremmo ottenuto un errore equivalente anche utilizzando il modificatore di accesso **protected**, in quanto **protected** stabilisce che l'attributo o metodo così dichiarato, può essere utilizzato solo all'interno della classe, ma a differenza di **private**, ne consente l'utilizzo anche nelle classi derivate, argomento che tratteremo più avanti parlando della **Ereditarietà**.

Oggetti e Membri statici

- Prima di introdurre il concetto dei **membri statici**, è necessario sapere che per utilizzare un membro di una **classe**, un **attributo** o un **metodo**, è obbligatorio creare un'istanza di tale classe attraverso la parola chiave **new**, come illustrato in precedenza.

- $utente = new Persona("John", "Doe", "1-1-1970");

- In questo modo, ogni istanza della classe avrà la propria copia di ogni attributo e ogni metodo.

Oggetti e Membri statici

Per capire meglio, prendiamo un esame un'ipotetica classe **"Colore"**, che avrà una serie di colori di base.

Colore.php

```
<?php
        class Colore{
                static $rosso = "#FF0000";
                static $verde = "#00FF00";
                static $blu = "#0000FF";
                /*.. altri attributi*/
                public function Colore()
                { /* codice costruttore */ }
                /*... altri metodi*/
                static public function stampaColore($colore){
                    echo "<font color=\"" . Colore::${$colore} . "\">Il valore esadecimale
del colore $colore è : ";
                    echo Colore::${$colore} . "</font><br />\n";
                }
        }
 ?>
```

Membri statici – usare Colore

Vediamo come è possibile utilizzare i tre membri statici della classe **Colore**.
E' sufficiente specificare il nome della classe seguito dai doppi due punti e il nome del membro :

test.php

```
<?php
        require_once("Colore.php");

        echo "<font color=\"" . Colore::$rosso . "\">Il valore esadecimale del colore rosso è : ";
        echo Colore::$rosso . "</font><br />\n";

        echo "<font color=\"" . Colore::$verde . "\">Il valore esadecimale del colore verde è : ";
        echo Colore::$verde . "</font><br />\n";

        echo "<font color=\"" . Colore::$blu . "\">Il valore esadecimale del colore blu è : ";
        echo Colore::$blu . "</font><br />\n";
 ?>
```

Colore - commenti

Se in un secondo momento decidiamo che per il nostro sito è più adatto un rosso scuro, sarà sufficiente assegnarvi tale valore all'inizio dello script, rendendo così effettiva la modifica in tutto il codice, senza peraltro modificare la classe che potrebbe essere condivisa con altre applicazioni anche esterne al server.

Nell'esempio che segue, utilizzeremo il metodo statico **"stampaColore()"** per visualizzare i colori sul browser.

test.php

```
<?php
        require_once("Colore.php");
        Colore::stampaColore("rosso");

        Colore::$rosso = "#C11C1C"; // Rosso più scuro
        Colore::stampaColore("rosso");
 ?>
```

Self

Per accedere ai membri statici dall'interno della classe, **$this** non è adeguato.
Il modo corretto per accedervi dall'interno è mediante il costrutto **self** :

```
<?php
        class Colore
        {
                static $rosso = "#FFoooo";
                // ...
                static public function stampaColore($colore)
                {
                        self::$rosso = "#FFoooo";
                        echo "<font color=\"" . Colore::${$colore} . "\">Il
valore esadecimale del colore $colore è : ";
                        echo Colore::${$colore} . "</font><br />\n";
                }
        }

?>
```

Oggetti e costanti

Potete pensare alle **costanti** delle classi, come a degli attributi statici che non possono però essere modificati una volta dichiarati e definiti. Prendiamo nuovamente in esempio la classe **"Colore"** esaminandone un utilizzo con le costanti :

Colore.php

```
<?php
        class Colore
        {
                const ROSSO = "#FF0000";
                const VERDE = "#00FF00";
                const BLU = "#0000FF";
                static public function stampaRosso()
                {
                        echo "<font color=\"" . self::ROSSO . "\">Il valore
esadecimale del colore rosso è : ";
                        echo self::ROSSO . "</font><br />\n";
                }
        }
 ?>
```

Utilizzo delle costanti globali

Con le costanti globali è necessario omettere il simbolo del dollaro **$** durante la dichiarazione.
I nomi delle costanti in PHP 5 sono sempre Case Sensitive, ed è buona norma scriverle tutte in maiuscolo per distinguerle immediatamente come costanti, ma non è obbligatorio.

test.php

```
<?php

        require_once("Colore.php");

        echo Colore::ROSSO . "<br />\n";
        Colore::stampaRosso();

?>
```

Oggetti ed Ereditarietà

- Il concetto dell'**ereditarietà**, è uno dei più importanti della programmazione orientata agli **oggetti**, a cui si appoggiano altri metodi avanzati di programmazione come ad esempio il **Polimorfismo** o le **Classi Astratte** che vedremo più avanti.

- L'**ereditarietà** ci consente di creare delle classi (**classi derivate** o **sottoclassi**) basate su classi già esistenti (**classi base** o **superclassi**).

- Un grande vantaggio è quello di poter riutilizzare il codice di una **classe di base** senza doverlo modificare.

Oggetti ed Ereditarietà

- L'**ereditarietà** ci consente quindi di scrivere del codice molto più flessibile, in quanto permette una generalizzazione molto più forte di un concetto, rendendo più facile descrivere una situazione di vita reale.

- Pensate alla **classe base** come ad un oggetto che descrive un concetto generale, e pensate invece alle **sottoclassi** come ad una specializzazione di tale concetto, esteso mediante proprietà e metodi aggiuntivi.

Oggetti e Ereditarietà: esempio (1/3) – classe “madre”

Animale.php

```
<?php

    class Animale // Classe Base
    {
            public $zampe;
            public $ordine;
            public $nome;

            public function Animale($z, $o, $n) // Costruttore
            {
                    $this->zampe = $z;
                    $this->ordine = $o;
                    $this->nome = $n;
            }

            protected function stampaDati() // Metodo protetto : può essere richiamato solo
    dalle classi derivate
            { echo $this->nome . " : Zampe = " . $this->zampe . " / Ordine = " . $this->ordine;
    }
    }

?>
```

Oggetti e Ereditarietà: esempio (2/3) – classe ereditata

Cane.php

```
<?php

    require_once("Animale.php");

    class Cane extends Animale // "Sottoclasse" o "Classe Derivata"{
            public function Cane() // Costruttore classe derivata
            {
                    parent::Animale(4, "Vertebrati", "Cane"); // Chiamata al costruttore
    della classe madre

                    /* oppure più generalizzato

                    parent::__construct(4, "Vertebrati", "Cane"); */
            }

            public function stampaDati($suono)
            {
                    echo "Faccio $suono perchè sono ";
                    parent::stampaDati(); // Chiamata al metodo protetto della classe
    madre
            }
    }
?>
```

Oggetti e Ereditarietà: esempio (3/3) – classe ereditata

Gallina.php

```
<?php

    require_once("Animale.php");

    class Gallina extends Animale // "Sottoclasse" o "Classe Derivata"
    {
            public function Gallina() // Costruttore classe derivata
            {
                    parent::Animale(2, "Vertebrati", "Gallina"); // Chiamata al
    costruttore della classe madre
            }

            public function stampaDati($suono)
            {
                    echo "Faccio $suono perchè sono ";
                    parent::stampaDati(); // Chiamata al metodo protetto della
    classe madre
            }
    }

?>
```

Ereditarietà: commenti all'esempio

- Il metodo **stampaDati** di **Animale** è dichiarato con il modificatore di accesso **protected**, ossia può essere utilizzato solo all'interno della classe madre **Animale** e all'interno di tutte le classi da essa derivate, quindi anche **Cane** e **Gallina**.

- Per richiamare il costruttore della classe madre dalle figlie è sufficiente utilizzare **parent::** invece di **this**, in questo modo accediamo direttamente alla classe base, allo stesso modo con cui **self::** viene usato al posto di **this**.

Ereditarietà: utilizzo delle classi in esempio

test.php

```
<?php

    require_once("Cane.php");
    require_once("Gallina.php");

    $cane = new Cane();
    $cane->stampaDati("bau");

    echo "\n\n<br /><br />\n\n";

    $gallina = new Gallina();
    $gallina->stampaDati("chicchirichì");

?>
```

Sottoclasse derivata da una sottoclasse

Alano.php

```
<?php
    require_once("Cane.php");
    class Alano extends Cane{
            private $suono;
            public function Alano()
            {
                    parent::Cane();
                    $this->suono = "wuoff";
            }
            public function stampaDati()
            {
                    parent::stampaDati($this->suono);
                    echo " / Razza : Alano";
            }
    }
?>
```

Utilizzare Alano

test.php

```
<?php

    require_once("Alano.php");

    $cane = new Alano();
    $cane->stampaDati();

?>
```

Polimorfismo

- Il Polimorfismo è la capacità di utilizzare un unico metodo in grado di comportarsi in modo specifico quando applicato a tipi di dato differenti.
- Questo è reso possibile grazie all'ereditarietà, che consente alle **sottoclassi** di ridefinire i metodi ereditati dalla classe base.
- Il linguaggio può identificare una qualunque istanza della sottoclasse, come un'istanza della classe base, poiché per il principio dell'**ereditarietà**, ogni sottoclasse possiede per natura tutte le proprietà della classe base (attributi e metodi).

Esempio


```
<?php
    class Animale{/* Rendiamo protetto il metodo per obbligare una  ridefinizione in una sottoclasse. Più avanti per ottenere un risultato migliore useremo le Classi Astratte. */
        protected function zampe(){
            echo "Errore : la funzione va obbligatoriamente ridefinita da una sottoclasse!";
        }
    }
    class Cane extends Animale{
        public function zampe(){
            echo "Zampe : 4";
        }
    }
    class Gallina extends Animale{
        public function zampe(){
            echo "Zampe : 2";
        }
    }
    function numeroZampe($oggetto){
        if ($oggetto instanceof Animale) // Condizione = Se oggetto è un istanza di Animale o derivata da essa
        {
            $oggetto->zampe();
        }
        else{
            echo "Tipo di oggetto non riconosciuto!!";
        }
    }
    numeroZampe(new Cane()); // Stampa : "Zampe : 4"
    numeroZampe(new Gallina()); // Stampa : "Zampe : 2"
?>
```

Commenti

- Nell' esempio possiamo notare che abbiamo stabilito un nome univoco per il nostro metodo (**"zampe()"**), ereditato direttamente dalla classe base **Animale**.

- Flessibilità della funzione esterna **numeroZampe()**, che sarà in grado di gestire anche altri classi oltre a **Cane** e **Gallina**, definibili in un secondo momento senza necessità di apportare modifiche alla funzione **numeroZampe()** o alle tre classi (Animale, Cane e Gallina).

Clonare

- In PHP 4, durante la creazione di un oggetto attraverso la parola chiave **new**, veniva restituito l'oggetto stesso e questo veniva memorizzato nella variabile specificata.

- In **PHP 5** invece, quando creiamo una nuova istanza (**$oggetto = new MiaClasse();**), **new** ci restituisce non il nuovo oggetto ma un riferimento ad esso.

Clonare

- In **PHP 5** se assegnate ad una variabile l'istanza di un oggetto, l'assegnazione avverrà per riferimento poichè l'istanza stessa contiene solo un riferimento all'oggetto creato.

- Se volete quindi creare una copia di un'istanza, dovrete clonarla. Per clonare un oggetto è sufficiente mettere la parola chiave **clone** dopo l'operatore di assegnazione **"="**.

Esempio clonazione Oggetti


```
<?php
    class Oggetto{
            public $valore;
            public function Oggetto($v){
                    $this->valore = $v;
            }
    }
    $istanza1 = new Oggetto(5);
    $istanza2 = $istanza1; // Assegnazione per riferimento
    $istanza3 = clone $istanza1; // Clonazione oggetto
    $istanza2->valore = 7; // Modifica anche $istanza1
    $istanza3->valore = 13;

    echo $istanza1->valore; // Non stampa 5 ma 7
    echo $istanza2->valore; // Stampa 7
    echo $istanza3->valore; // Stampa 13
 ?>
```

Clonando un oggetto, per definizione si crea una copia esatta di tale oggetto, quindi i riferimenti contenuti in esso saranno comunque copiati come tali, e dopo la clonazione conterranno ancora il riferimento alla stessa risorsa di prima.

Esempio clonazione Oggetti

Clonando un oggetto, per definizione si crea una copia esatta di tale oggetto, quindi i riferimenti contenuti in esso saranno comunque copiati come tali, e dopo la clonazione conterranno ancora il riferimento alla stessa risorsa di prima.

Classi astratte

- Le **classi astratte** ci permettono di specificare con esattezza quali classi e quali metodi devono obbligatoriamente essere ridefiniti da una sottoclasse per poter essere utilizzati.

- La sottoclasse derivata da una classe astratta, dovrà obbligatoriamente definire tali metodi astratti per far sì che l'ereditarietà venga accettata.

Utilizzo classi astratte


```
<?php
    abstract class Animale{
            protected $zampe;
            protected function Animale($z){
                    $this->zampe = $z;
            }
            public function numeroZampe(){
                    echo $this->zampe;
            }
            abstract protected function suono();
    }
    class Cane extends Animale{
            public function Cane(){
                    parent::Animale(4);
            }
            public function suono()
            { echo "bau!"; }
    }
    $rex = new Cane();
    $rex->numeroZampe(); // Stampa 4
    $rex->suono(); // Stampa "bau!"
?>
```

- Il metodo **astratto "suono()"** della classe astratta **"Animale"** è solamente dichiarato, mentre la definizione avviene direttamente nella classe derivata.
- Dichiarando Cane senza la definizione di suono(), avremmo ottenuto un errore.