Uso delle classi in php – Introduzione di base

Con questo articolo iniziamo ad addentrarci in uno dei pilastri portanti della programmazione: l'uso degli oggetti di classe. Il php è un linguaggio di programmazione a 360° e come gli altri suoi fratelli più in voga integra un sistema di uso delle classi abbastanza intuitivo.

Una volta conosciute le basi e i primi approfondimenti del php, è quasi indispensabile passare alle classi. Perché? Le motivazioni sono molte (più sotto le vedremo nei dettagli), ma **principalmente è una questione di professionalità e utilità**: il codice è ben ordinato ed è possibile lavorare in più persone contemporaneamente su un progetto, senza temere di intaccare qualche funzionalità fondamentale. L'argomento dell'uso agli oggetti è davvero vasto: con l'esperienza scoprirete quanto è utile e complesso, e che si tratta del vero "cuore" dei linguaggi di programmazione, dove il ragionamento e l'organizzazione fanno la differenza.

Nel tutorial (suddiviso in più articoli) vedremo un utilizzo di base delle classi. A fine spiegazione **dovreste essere in grado di costruirvi una vostra classe rudimentale** e di popolarla con dei metodi. In seguito vedremo, inoltre, un esempio semplice e concreto (che potrete liberamente scaricare) dell'uso di una classe.

In generale: la programmazione a oggetti

Con la parola "oggetti" non si intende altro che una zona di memoria allocata. La "programmazione orientata agli oggetti" (che in genere si abbrevvia con OOP, dall'inglese *Object Oriented Programming*) è idealmente un **insieme di strumenti offerti, nel nostro caso, dal php per permettere agli oggetti di interagire tra loro**, scambiandosi dei messaggi. Restringendo, quindi, la OOP è una serie di strumenti che permette a due o più zone di memoria (oggetti) di comunicare tra loro.

Fin qui la descrizione generale. Andiamo un po' più nel concreto.

Quando lavoriamo con OOP, di qualsiasi linguaggio di programmazione, facciamo uso di quelle che vengono chiamate "classi": si tratta di modelli, che nella pratica possiamo vedere come contenitori di azioni (*metodi*) e di attributi. In pratica la struttura base è la seguente:

```
1  class Nome_classe{
2
3          function metodo1(){
4                  [...]
5          }
6
7          function metodo2(){
8                  [...]
9          }
10
11 }
```

Come notate, i metodi vengono chiamati attraverso l'uso di "function" seguito dal nome: se avete familiarità con le funzioni, vi renderete conto che in effetti i metodi hanno molto in comune con queste (ma non confondete "metodi" con "funzioni": sono usate con delle differenze sostanziali).

La classe è una struttura astratta: di per sé non fa niente, raggruppa semplicemente delle azioni e degli attributi.

Per essere usata, la classe deve essere "istanziata": **dobbiamo cioè creare un oggetto che permetta di accedere a quella classe**. Visivamente si tratta di creare una variabile (che viene chiamata, per l'appunto "istanza"). Grazie a questo oggetto potremmo recuperare tutti i metodi di quella classe. Continuando l'esempio sopra:

```
1 $istanza=new Nome_classe(); //istanzio la classe
2 $istanza->metodo1; //accedo al metodo1 della classe
3 $istanza->metodo2; //accedo al metodo2 della classe
```

Perché usare OOP invece di semplici funzioni

Qualcuno di voi potrebbe chiedersi quale sia la vera utilità della OOP. Innanzitutto, l'uso di classi ci permette **di "spezzettare" il codice in più parti e di recuperare la parte che ci interessa** tralasciando tutte le altre. Per esempio, nel nostro sito possiamo costruire una classe che si occupi esclusivamente di stampare i prodotti e un'altra che esegua il login a un'area riservata. Non potrei costruire tutto usando semplicemente le funzioni?

Questa è una domanda che chiunque si pone quando si trova davanti, per la prima volta, alla OOP. Gran parte delle situazioni, in effetti, potrebbero essere "replicate" dalle funzioni. Ma:

- le funzioni sono meno performanti
- le classi permettono ulteriori funzionalità, come **per esempio la possibilità di sovrascrivere i propri metodi** (cosa non permessa con le funzioni: una volta creata, la funzione rimane così com'è)
- le classi permettono una gestione più ordinata
- le classi permettono a più persone di lavorare su uno stesso progetto senza "pestarsi i piedi", limitando così gli errori solo a un settore del progetto
- una classe può essere "presa" (tecnicamente si dice "estesa") da più classi diverse: in pratica possiamo usare una sola base per creare funzionalità diverse

Queste sono soltanto alcune considerazioni. La vera utilità della OOP sarà chiara soltanto con l'esperienza. Dopotutto, una ragione c'è se viene adoperata in tutti i linguaggi di programmazione più avanzati…

Funzionamento di base delle classi

Sopra abbiamo visto che per richiamare un metodo di classe, **bisogna istanziare quest'ultima in un oggetto**. Dopodiché potremo richiamare i metodi di quella classe facendo:

```
1 $oggetto->metodo();
```

Vediamo cosa possiamo fare all'interno di una classe e all'interno di un metodo. Come punto di riferimento prendiamo la struttura vista a inizio articolo, ovvero la classe *Nome_classe* che contiene al suo interno *metodo1* e *metodo2*. Aggiungiamo qualche altro particolare:

```
1  class Nome_classe{
2
3          public $variabile_a="Prima variabile";
4          plubic $variabile_b=20;
5
6          function metodo1(){
7                  echo $this->variabile_b;
8          }
9
10         function metodo2(){
```

```
11                  $this->metodo1();
12          }
13
14  }
15
16  $istanza=new Nome_classe(); //istanzio la classe
17  $istanza->metodo2(); //accedo al metodo2 della classe
```

Come notiamo qui sopra, **anche all'interno delle classi possiamo dichiarare delle variabili** (che in una classe prendono il nome di *proprietà*). Alle proprietà possiamo assegnare un valore di qualsiasi tipo che conosciamo: stringa, intero, array di valori, ecc. Niente di diverso dalle comuni variabili: l'unica cosa che cambia è l'attributo *public*, di cui ci occuperemo tra poco.

Se dichiariamo una variabile all'interno della classe, questa sarà visibile in tutti i metodi della classe e potrà essere richiamata sempre con "$this->nome_variabile" (notare l'assenza del $, indispensabile quando si inizializza la variabile nella classe ma errato se usato quando si richiama quella stessa variabile all'interno di un metodo).

Non solo: l'attributo ***$this*** **può essere usato per richiamare un altro metodo della stessa classe** (nel nostro esempio, "metodo2" richiama "metodo1").

Nell'esempio, con l'oggetto di classe istanziato noi richiamiamo "metodo2". A sua volta, il metodo2 richiama "metodo1", che si occupa di stampare a video $variabile_b. Il risultato è che a video sarà stampato il valore 20.

Passare un parametro a un metodo

Aggiungiamo adesso un terzo metodo:

```
1   class Nome_classe{
2
3           [...]
4
5           function metodo3($parametro){
6                   echo $parametro;
7           }
8
9   }
10
11  $istanza=new Nome_classe();
12  $istanza->metodo3("Questa è una stringa");
```

In questo caso abbiamo fatto passare un parametro al metodo. Se notate non si usa l'attributo $this: questo perché il parametro appartiene unicamente al metodo, non alla classe intera. $parametro, infatti, è accessibile soltanto all'interno di "metodo3", mentre non sarà visibile o utilizzabile direttamente in "metodo1" e "metodo2". Il risultato sarà la stampa a video di "Questa è una stringa".

Una nota sulle proprietà

Soffermiamoci adesso sull'inizializzazione delle proprietà (cioè delle variabili di classe). Possiamo farlo in diversi modi, per esempio:

```
1  var $a;
2  public $b="Testo";
3  private $c=10;
4  protected $d=20;
```

Nel primo caso ($a) abbiamo semplicemente inizializzato una variabile vuota. Scrivere una cosa del genere sarebbe stato sbagliato:

```
1 var $a="Stringa";
```

Per le altre tre proprietà abbiamo assegnato **tre attributi diversi: public, private, protected**. Questi tre attributi sono utili per stabilire il loro comportamento all'esterno della classe: i metodi di una classe, infatti, possono essere ereditati e usati da altre classi (di questo però ci occuperemo nel prossimo articolo).

Questi stessi attributi possono essere assegnati anche ai metodi, non soltanto alle variabili. Per esempio:

```
1  public function metodo4(){
2          [...]
3  }
4
5  protected function metodo5(){
6          [...]
7  }
8
9  private function metodo6(){
10         [...]
11 }
```

Vediamoli in breve:

- *public*: la proprietà o metodo è visibile dalla classe corrente, dalle sue sottoclassi e quando richiamato all'esterno
- *protected*: visibile solo dalla classe corrente e dalle sue sottoclassi
- *private*: visibile solo dalla classe corrente

Se non specificato, un metodo si considera *public*.

Se cerchiamo, per esempio, di accedere al "metodo6" (di tipo *private*) dall'esterno della classe, ci verrà restituito un errore.

Costanti e metodi statici

Anche **le OOP fanno uso di costanti**. Il loro accesso, però, è diverso da quello a cui siamo abituati con la normale programmazione. Inoltre, le costanti devono essere valori semplici: una stringa, un numero, ecc. Non possono essere, per esempio, il risultato di un'operazione matematica, un'assegnazione di variabile o un array. Per dichiarare la costante "STATO" all'interno di una classe:

```
1 const STATO = "Italia";
2
3 //tutte le seguenti assegnazioni sono errate
4 const RISULTATO = 9*8;
5 const RISULTATO = array(9, 8);
6 const RISULTATO = $variabile;
```

Come le classiche costanti, il loro valore non può cambiare una volta inizializzato. Le costanti non possono essere richiamati come le altre proprietà: non si possono usare quindi $this o ->. Per utilizzarle all'interno di un metodo **utilizzeremo la sintassi "self::_nome_costante"**, mentre per richiamarla al di fuori della classe dovremmo usare "nome_classe::nome_costante". Per esempio l'istanza stamperà a video in entrambi i casi "Italia":

```
1  class Nome_classe{
2
3          [...]
```

```
4
5           function metodo4(){
6                   return self::STATO;
7           }
8
9   }
10
11  $istanza=new Nome_classe();
12  echo $istanza->metodo4();
13  echo Nome_classe::STATO;
```

Per completezza, concludiamo dicendo che esiste anche una forma di **metodi che vengono richiamati allo stesso modo delle costanti: i "metodi statici"**. Perché usarli? A differenza delle costanti viste sopra, i valori di questi metodi statici possono essere modificati come nei metodi comuni. All'interno di questi metodi non è possibile usare $this. Un metodo statico si crea con la clausola "static".

Per esempio:

```
1   class Nome_classe{
2
3           [...]
4
5           static function metodo4(){
6                   echo "prova";
7           }
8
9           function metodo5(){
10                  self::metodo4();
11          }
12
13  }
14
15  $istanza=new Nome_classe();
16  Nome_classe::metodo4();
```

Come vedete, all'esterno della classe è buona norma richiamare il metodo come con le costanti. A dire la verità funziona anche l'uso del solito operatore di deferenziamento (->), ma è una pratica sconsigliata.