

CLASSI

Una classe è una collezione di **variabili** e **funzioni** che utilizzano le variabili definite all'interno della classe stessa per svolgere determinate azioni. Di seguito è riportato il codice semplificato per definire una classe:

```
<?php
class nome_classe
{
    var $nome_variabile; // DEFINIZIONI VARIABILI
    var ...

    function nome_funzione ( $var1, $var2, ... ) // DEFINIZIONI FUNZIONI
    {
        ...
    }
}
?>
```

Un esempio di classe è il seguente:

```
<?php
class Cart
{
    var $items; // Articoli nel carrello

    // Aggiunge $num articoli di $artnr nel carrello

    function add_item ($artnr, $num)
    {
        $this->items[$artnr] += $num;
    }

    // Prende $num articoli di $artnr e li rimuove dal carrello

    function remove_item ($artnr, $num)
    {
        if ($this->items[$artnr] > $num) {
            $this->items[$artnr] -= $num;
            return true;
        } else {
            return false;
        }
    }
}
?>
```

Il codice definisce una classe chiamata Cart composta da un array associativo che archivia gli articoli nel carrello e due funzioni per aggiungere e rimuovere gli articoli dal carrello stesso.

Avvertimento: NON spezzate una definizione di classe in più file o in più blocchi PHP. Il seguente codice non funziona:

```
<?php
class test {
?>
<?php
function test() { print 'OK'; }
}
?>
```

In PHP 4, sono permesse **inizializzazioni di variabili** con valori costanti solamente grazie all'uso di **var**. Per inizializzare variabili con valori non costanti, bisogna creare una funzione di inizializzazione che viene eseguita automaticamente durante la fase di istanza della classe stessa. Questo tipo di funzione si chiama **costruttore**.

```
<?php
/* questo non funziona in PHP 4. */

class Cart
{
    var $todays_date = date("Y-m-d");
    var $name = $firstname;
    var $owner = 'Fred ' . 'Jones';
    var $items = array("VCR", "TV");
}

/* Questo è corretto. */

class Cart
{
    var $todays_date; // Queste sono le PROPERTIES
    var $name;
    var $owner;
    var $items;

    function Cart() // Questo è il COSTRUTTORE
    {
        $this->todays_date = date("Y-m-d");
        $this->name = $GLOBALS['firstname'];
        /* etc ... */
    }
}
?>
```

Si definisce **Property** di una classe la variabile chiamata con il var all'inizio della classe stessa, ed avente un nome qualsiasi.

Istanziare una classe significa assegnare la classe ad una variabile usando la funzione natia **new**. Per creare quindi una variabile oggetto si usa l'operatore **new**.

I **costruttori** sono funzioni che esistono in una classe e che sono chiamate automaticamente quando si crea una nuova istanza di una classe con **new**. In PHP 3, una funzione si trasforma in un costruttore quando ha lo stesso nome di una classe. In PHP 4, una funzione diventa un costruttore quando ha lo stesso nome di una classe ed è definita all'interno della classe stessa; la differenza è sottile, ma cruciale.

```
<?php
class Cart2
{
    var $items;
    function add_item ($artnr, $num)
    {
        $this->items[$artnr] += $num;
    }
    function remove_item ($artnr, $num)
    {
        if ($this->items[$artnr] > $num) {
            $this->items[$artnr] -= $num;
            return true;
        } else {
            return false;
        }
    }
}

$cart2 = new Cart2;
$cart2->add_item("10", 1);
$another_cart2 = new Cart2;
$another_cart2->add_item("0815", 3);

print_r($cart2);
print_r($another_cart2);
?>
```

Il codice sopra, genera gli oggetti \$cart2 e \$another_cart2, dalla classe Cart2. La funzione add_item() dell'oggetto \$cart2 è chiamata per aggiungere una ricorrenza dell'articolo numero 10 a \$cart2 (una sorta di carrello spesa). Ad \$another_cart2 sono aggiunte 3 ricorrenze dell'articolo numero 0815.

L'output è:

```
cart2 Object ( [items] => Array ( [10] => 1 ) )
another_cart2 Object ( [items] => Array ( [0815] => 3 ) )
```

Il separatore del pathname nome_oggetto / funzione_classe è ->.

\$cart2->items e **\$another_cart2->items** sono due diverse variabili che differiscono per il nome.

Si noti che la variabile si chiama \$cart2->items, e non \$cart2->\$items, questo perchè le variabili in PHP si scrivono con un unico simbolo di dollaro.

Per poter accedere alle funzioni e alle variabili interne di una classe si usa la pseudo-variabile **\$this** che può essere letta come 'la mia\il mio' o 'di questo oggetto'.

extends

Spesso si ha bisogno di avere classi con variabili e funzioni simili ad altre classi. È buona norma definire una classe in modo generico, sia per poterla riutilizzare spesso, sia per poterla adattare a scopi specifici. Per facilitare questa operazione, è possibile generare classi per estensione di altre classi. Una classe estesa o derivata ha tutte le variabili e le funzioni della classe di base (questo fenomeno è chiamato *'eredità'*) più tutto ciò che viene aggiunto dall'estensione.

Non è possibile che una sottoclasse, ridefinisca variabili e funzioni di una classe madre.

Una classe estesa dipende sempre da una singola classe di base: l'eredità multipla non è supportata. Le classi si estendono usando la parola chiave 'extends'.

```
<?php
class Named_Cart extends Cart2
{
    var $owner;

    function set_owner ($name)
    {
        $this->owner = $name;
    }
}
$ncart = new Named_Cart; // Crea un carrello con nome
$ncart->set_owner("kris"); // Assegna il nome al carrello
print $ncart->owner; // stampa il nome del proprietario
$ncart->add_item("10", 1); // (funzionalità ereditata da Cart)
?>
```

Qui viene definita una classe Named_Cart che ha tutte le funzioni e variabili di Cart2 più la variabile \$owner e la funzione set_owner(). Viene creato un carrello con nome con il metodo usato in precedenza, in più la classe estesa permette di settare o leggere il nome del carrello.

La relazione mostrata è chiamata relazione "*genitore-figlio*". Si crea una classe di base, poi utilizzando extends si crea una nuova classe basata sulla classe genitore: la classe figlia. Successivamente si può usare la classe figlia come classe base per un'altra classe.

A volte è utile riferirsi alle funzioni ed alle variabili di classi base o riferirsi alle funzioni di classi senza istanziarle. L'operatore :: è usato per questi scopi.

```
<?php
class A
{
    function example()
    {
        echo "Sono la funzione originale A::example()";
    }
}

class B extends A
{
    function example()
    {
        // ...
    }
}
```

```

    {
        echo "Sono la funzione ridefinita B::example()";
        A::example();
    }
}

// non viene istanziato nessun oggetto dalla classe A.
// ma il codice stampa
// Sono la funzione originale A::example()
A::example();

// crea un oggetto dalla classe B.
$b = new B;

// questo codice stampa
// Sono la funzione ridefinita B::example()
// Sono la funzione originale A::example()
$b->example();
?>

```

L'esempio chiama la funzione `example()` della classe A, ma senza creare un'istanza di A, di modo che la funzione non si possa richiamare con `$a->example()`. `example()` è chiamata come 'funzione della classe', e non come funzione di un oggetto della classe.

Si possono usare funzioni della classe, ma non le variabili della classe. Infatti, non esiste nessun oggetto nel momento della chiamata della funzione. Quindi, la funzione della classe non può usare le variabili dell'oggetto (ma può usare le variabili locali e globali) e `$this` non può essere usato. Nel suddetto esempio, la classe B ridefinisce la funzione `example()`. La funzione originale definita nella classe A è adombrata e non più disponibile, a meno che voi non chiamate esplicitamente con l'operatore `::`, scrivendo `A::example()` per richiamare la funzione.

E' possibile ritrovarsi a scrivere classi con codice che si riferisce a variabili e funzioni di classi base. Ciò è particolarmente VERO se una classe derivata è un perfezionamento o una specializzazione di una classe base.

Invece di usare il nome letterale della classe, bisognerebbe usare il nome speciale **parent**, che si riferisce al nome della classe base definita nella dichiarazione di `extends`. Usando questo metodo, si evita di usare il nome della classe base nel codice scritto. Se l'albero di eredità cambiasse durante lo sviluppo della classe, il cambiamento si ridurrebbe semplicemente alla modifica della dichiarazione `extends` della classe.

```

<?php
class A
{
    function example()
    {
        echo "Sono la funzione originale A::example()";
    }
}

class B extends A
{
    function example()
    {
        echo "Sono B::example() e fornisco una funzionalità aggiuntiva.";
        parent::example();
    }
}

$b = new B;

// Il codice chiama B::example(), che a sua volta chiama A::example().
$b->example();
?>

```

Visibilità

La visibilità di una property o di un metodo, può essere definita prefissando la loro dichiarazione con una parola chiave: "*public*", "*protected*" o "*private*". *Public* dichiara un metodo o una variabile che può essere vista da tutti, *Protected* invece limita l'accesso alla classe che la definisce ed alle sue sottoclassi, mentre *Private* consente l'accesso alla sola classe che la definisce.

Esempio:

```

<?php
class MyClass

```

```

{
    public $public = 'Public';
    protected $protected = 'Protected';
    private $private = 'Private';

    function printHello()
    {
        echo $this->public;
        echo $this->protected;
        echo $this->private;
    }
}

$obj = new MyClass();
echo $obj->public; // Funziona
echo $obj->protected; // va in Errore
echo $obj->private; // va in Errore
$obj->printHello(); // Mostra Public, Protected e Private

class MyClass2 extends MyClass
{
    // Si può ridichiarare i metodi public protected, ma non quello privato
    protected $protected = 'Protected2';

    function printHello()
    {
        echo $this->public;
        echo $this->protected;
        echo $this->private;
    }
}

$obj2 = new MyClass2();
echo $obj2->public; // Funziona
echo $obj2->private; // Indefinita
echo $obj2->protected; // va in Errore
$obj2->printHello(); // Mostra Public, Protected2, e non Private
?>

```

Nell'esempio seguente, invece, viene mostrato lo stesso concetto di visibilità applicato però ai metodi di una classe:

```

<?php
class MyClass
{
    // il Costruttore deve essere public
    public function __construct() { }

    // dichiarazione di un metodo public
    public function MyPublic() { }

    // dichiarazione di un metodo protected
    protected function MyProtected() { }

    // dichiarazione di un metodo private
    private function MyPrivate() { }

    // questo è public
    function Foo()
    {
        $this->MyPublic();
        $this->MyProtected();
        $this->MyPrivate();
    }
}

$myclass = new MyClass;
$myclass->MyPublic(); // Funziona
$myclass->MyProtected(); // va in Errore
$myclass->MyPrivate(); // va in Errore
$myclass->Foo(); // Public, Protected e Private funzionano

class MyClass2 extends MyClass
{
    // questo è public
    function Foo2()
    {

```

```

        $this->MyPublic();
        $this->MyProtected();
        $this->MyPrivate(); // va in Errore
    }
}

$myclass2 = new MyClass2;
$myclass2->MyPublic(); // Funziona
$myclass2->Foo2(); // Public e Protected funzionano, ma non Private
?>

```

Variabile o metodo Statico

Di seguito è riportato un esempio di variabile statica:

```

<?php
class foo
{
    static $my_static = 5;
    public $my_prop = "asd";
}

print foo::$my_static;
$obj = new foo;
print $obj->my_prop;
?>

```

La differenza quindi tra una variabile statica ed una non, è che è possibile accederci senza istanziare la classe, ma usando `i::`.

Di seguito, invece, è mostrato un esempio di metodo statico, la cui definizione è la stessa:

```

<?php
class foo
{
    public static function my_static_method()
    {
        ...
    }
}

print foo::$my_static;
$obj = new foo;
print $obj->my_prop;
?>

```

Eccezioni

La gestione delle Eccezioni non è presente in PHP 4, ma è presente nella versione 5, che introduce tale concetto nello stesso modo di quello presente ad esempio in Java. Viene supportata la clausola "catch all" ma non la "finally". L'eccezione può essere gestita anche nei blocchi catch, esattamente come in Java; ma se questa non viene gestita in alcun modo, l'applicazione va in eccezione. Di seguito è riportato il codice php necessario per gestire una generica eccezione:

```

<?php
try
{
    $error = 'throw questo errore';
    throw new Exception($error);

    // Il codice che segue ad una eccezione non viene eseguito.
    echo 'Mai eseguita!';
}
catch (Exception $e)
{
    echo 'Eccezione trovata: ', $e->getMessage(), "\n";
}

// Continua l'esecuzione dell'applicazione
echo 'Hello World';
?>

```

Il blocco *Try* viene usato per gestire parte del codice php dell'applicazione che potrebbe generare un'eccezione, che viene però riportata e quindi gestita dal blocco seguente *catch* attraverso la parola chiave *throw*; in genere il blocco catch printa il tipo di errore. Nel blocco try, le istruzioni che seguono la riga del throw, non vengono eseguite nel caso in cui si genera un'eccezione.

E' sempre possibile definire una calsse madre Exception contenente metodi generici per la gestione di errori:

```
<?php
class Exception
{
    protected $message = 'Unknown exception'; // exception message
    protected $code = 0; // user defined exception code
    protected $file; // source filename of exception
    protected $line; // source line of exception

    function __construct($message = null, $code = 0);

    final function getMessage(); // message of exception
    final function getCode(); // code of exception
    final function getFile(); // source filename
    final function getLine(); // source line
    final function getTrace(); // an array of the backtrace()
    final function getTraceAsString(); // formatted string of trace

    /* Overrideable */
    function __toString(); // formatted string for display
}
?>
```

ed una classe figlia, MyException, che estenda quella madre:

```
<?php
class MyException extends Exception
{
    // Redefine the exception so message isn't optional
    public function __construct($message, $code = 0) {
        // some code

        // make sure everything is assigned properly
        parent::__construct($message, $code);
    }

    // custom string representation of object */
    public function __toString() {
        return __CLASS__ . ": [{"this->code}]: {"this->message}\n";
    }

    public function customFunction() {
        echo "A Custom function for this type of exception\n";
    }
}
?>
```

Di seguito è riportato il codice di una classe esempio per testare le eccezioni di cui sopra:

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

    const THROW_NONE = 0;
    const THROW_CUSTOM = 1;
    const THROW_DEFAULT = 2;

    function __construct($avalue = self::THROW_NONE)
    {
        switch ($avalue) {
            case self::THROW_CUSTOM:
                // throw custom exception
                throw new MyException('1 is an invalid parameter', 5);
        }
    }
}
```

