

LINGUAGGIO PHP

Storia

PHP e' l'acronimo di **Hypertext Preprocessor** è un **linguaggio di scripting** general-purpose **Open Source** molto utilizzato, è specialmente indicato per lo sviluppo Web e può essere integrato nell'HTML.

La sua sintassi è basata su quella del **linguaggio C** ed è molto semplice da imparare.

L'obiettivo principale del linguaggio è quello di permettere agli sviluppatori web di **scrivere velocemente pagine web dinamiche**.

La storia di PHP comincia nel 1994 con il PHP/FI (Personal Home Page / Form Interpreter) scritto da *Rasmus Lerdorf* (nella foto).

A seguito del successo riscosso nel popolo della rete, Rasmus scrisse una seconda versione che offriva un maggior numero di funzioni ed il supporto per interfacciare il database miniSQL.

Nel 1997 la costituzione di un team di lavoro porta alla riscrittura completa del codice con la versione php.

Con la versione 4 il team di lavoro ha potenziato l'implementazione dell'impalcatura object-oriented e con il prodotto **ZEND** (PHP4) offre un notevole incremento delle prestazioni.

Attualmente il team di lavoro *[23-Sep-2004]* ha rilasciato la versione 5.0.2

[Sito Ufficiale PHP Group](#)

Licenze e copyright [Leggi](#)

Caratteristiche

- Open Source software;
- Linguaggio di scripting server-side;
- linguaggio HTML - embedded : una pagina in PHP e' composta sia da tag HTML, sia da parti in codice PHP delimitate da particolari tag;
- fornisce un modo per inserire istruzioni nelle pagine HTML per creare contenuto dinamico;
- sintassi simile al linguaggio C, al Java ed al Perl;
- utilizzabile su piattaforme Unix e Windows 9X/Me/NT/2000;
- meccanismo di esecuzione degli script simile al linguaggio ASP in ambiente Windows;
- fornisce un'ampia varietà di funzioni (array, grafiche, matematiche, stringhe, HTTP, IMAP, XML ecc.);
- supporta un gran numero di funzioni Open Database Connectivity (ODBC) per la connessione ai più importati DBMS.

Installazione

Per poter utilizzare PHP occorrono tre cose:

- *i pacchetti di installazione del PHP*

Pacchetto per Windows 98/Me/NT/2000/XP/2003.

- VERSIONE 4.03
 - [PHP 4.3.9 zip package](#) [6,953Kb] - 22 Sep 2004
md5: c4a4c349e729b339a27c72abe833c58d
 - [PHP 4.3.9 installer](#) [1,045Kb] - 22 Sep 2004
md5: 4a2af567ba607e40e4ca4e58adba5cc0
- VERSIONE 5.02
 - [PHP 5.0.2 zip package](#) [7,538Kb] - 23 Sep 2004
md5: dd74b477b3b9cabf8c448e333fda2c6
 - [PHP 5.0.2 installer](#) [2,257Kb] - 24 Sep 2004
md5: 2528dcc6a8b6c6e5027e262ba35c5d12

- *un server web*
 - **PWS personal Web server** sul cd di Windows 98 nella cartella ADDONS
 - **IIS Internet Information Server** nelle versioni Windows 2000 / XP Professional (per la versione XP Home consulta la lezione su ASP)
 - **APACHE** <http://www.apache.org/>
- *un browser web*

se stai guardando questa lezione ne hai già uno

Steps per l'installazione su Linux:

- `gzip -d php-4.3.9.tar.gz` (unzip del file compresso)
- `tar -C /usr/local -xvf php-4.3.9.tar` (installazione del codice sorgente nella directory php-4.3.9)
- `cd php-4.3.9.` (posizionamento sulla directory di installazione)
Esecuzione dello script di configurazione:
- `./configure --with-apache=/usr/local/apache_1.3.9` (indica la posizione di Apache) `--with-mysql=/usr/local/mysql` (abilitazione utilizzo librerie per interfacciare il database MySQL) `--with-gd=path gd library` (utilizzo della libreria grafica GD)
- `make` (compilazione)
- `make install` (installazione)
- `cp php.ini-dist /usr/local/lib/php.ini` (copia il file che contiene le direttive per l'interprete)

Per informazioni dettagliate su sistemi Unix/Linux [Consulta la Guida Ufficiale](#)

Per l'istallazione su Windows seguite questi semplici passi:

- scarica i file `Zip_package` e `Installer` (vedi link sopra)
- Scomparta il file .zip scaricato nella directory `C:\php`
- Lancia il file installer

Per informazioni dettagliate fate riferimento alla guida [Consulta la Guida Ufficiale](#)

Esecuzione di una pagina in PHP

Per poter utilizzare PHP occorrono tre cose:

- *i pacchetti di installazione del PHP*

Pacchetto per Windows 98/Me/NT/2000/XP/2003.

- VERSIONE 4.03
 - [PHP 4.3.9 zip package](#) [6,953Kb] - 22 Sep 2004
md5: c4a4c349e729b339a27c72abe833c58d
 - [PHP 4.3.9 installer](#) [1,045Kb] - 22 Sep 2004
md5: 4a2af567ba607e40e4ca4e58adba5cc0
- VERSIONE 5.02
 - [PHP 5.0.2 zip package](#) [7,538Kb] - 23 Sep 2004
md5: dd74b477b3b9cabf8c448e333fda2c6
 - [PHP 5.0.2 installer](#) [2,257Kb] - 24 Sep 2004
md5: 2528dcc6a8b6c6e5027e262ba35c5d12

- *un server web*
 - **PWS personal Web server** sul cd di Windows 98 nella cartella ADDONS
 - **IIS Internet Information Server** nelle versioni Windows 2000 / XP Professional (per la versione XP Home consulta la lezione su ASP)
 - **APACHE** <http://www.apache.org/>
- *un browser web*

se stai guardando questa lezione ne hai già uno

Steps per l'installazione su Linux:

- `gzip -d php-4.3.9.tar.gzip` (unzip del file compresso)
- `tar -C /usr/local -xvf php-4.3.9.tar` (installazione del codice sorgente nella directory php-4.3.9)
- `cd php-4.3.9.` (posizionamento sulla directory di installazione)
Esecuzione dello script di configurazione:
- `./configure --with-apache=/usr/local/apache_1.3.9` (indica la posizione di Apache) `--with-mysql=/usr/local/mysql` (abilitazione utilizzo librerie per interfacciare il database MySQL) `--with-gd=path gd library` (utilizzo della libreria grafica GD)
- `make` (compilazione)
- `make install` (installazione)
- `cp php.ini-dist /usr/local/lib/php.ini` (copia il file che contiene le direttive per l'interprete)

Per informazioni dettagliate su sistemi Unix/Linux [Consulta la Guida Ufficiale](#)

Per l'istallazione su Windows seguite questi semplici passi:

- scarica i file `Zip_package` e `Installer` (vedi link sopra)
- Scomparta il file .zip scaricato nella directory `C:\php`
- Lancia il file installer

Per informazioni dettagliate fate riferimento alla guida [Consulta la Guida Ufficiale](#)

PHP.ini

Le impostazioni di PHP possono essere manipolate attraverso questo file. Diciamo subito che le impostazioni possibili sono molto numerose, e che non saranno considerate tutte, ma solo quelle che possono rivestire un certo interesse per chi è agli inizi con PHP.

Possiamo localizzare il file `php.ini` in questo modo: creiamo un semplice file `php` e chiamiamolo `Configurazione.php` con questa sola istruzione

```
<? phpinfo() ?>
```

Quando richiamiamo dal server il file otterremo una lunga lista di parametri che ci raccontano praticamente tutto della nostra installazione di PHP. In particolare, in una delle prime righe troveremo la voce 'Configuration File (php.ini) Path', di fianco alla quale potremo vedere dove si trova il 'nostro' `php.ini`.

Il file è piuttosto 'voluminoso', perché contiene dettagliate spiegazioni (in inglese) del significato di ogni impostazione. Dal punto di vista operativo, però, tutte le righe che iniziano con un punto e virgola sono totalmente insignificanti, perché vengono considerate commenti. Ricordiamoci quindi, quando vogliamo impostare determinate variabili, di controllare che non ci sia il punto e virgola all'inizio della riga interessata. Vediamo allora quali sono le impostazioni più interessanti per chi sta cominciando ad usare PHP:

- **short_open_tag**: impostato di default a 'on', ci permette, di aprire e chiudere PHP con i tag brevi `<? ... ?>`. Se lo impostiamo a 'off', questi tag non saranno più riconosciuti.
- **asp_tags**: se impostato a 'on', ci permette di utilizzare i tag di apertura e chiusura in stile ASP `<% ... %>`. Di default è comunque impostato a 'off'.
- **max_execution_time**: è il tempo limite concesso a PHP per l'esecuzione di uno script. E' espresso in secondi, ed è impostato di default a 30. Normalmente non c'è motivo di modificarlo, perché 30 secondi sono un tempo più che sufficiente per eseguire uno script PHP anche se deve elaborare quantità molto consistenti di dati.
- **error_reporting**: rappresenta il 'livello' di errori che vengono mostrati da PHP. Il nostro consiglio è di utilizzare l'impostazione 'E_ALL', che mostra tutti gli errori, compresi quelli meno gravi di tipo 'notice': in questo modo impareremo a creare un codice pulito e corretto nel quale sarà anche più facile individuare gli errori di logica.
- **display_errors**: stabilisce se gli errori devono essere mostrati sul browser. Se lo impostiamo ad 'off', non vedremo più gli errori sul browser e, per trovarli, dovremo abilitare la loro registrazione su un file di log. Per chi comincia è comunque consigliabile mantenere la visualizzazione sul browser, purché si lavori su macchine non accessibili pubblicamente.
- **log_errors** ed **error_log**: si usano nel caso in cui non vogliamo mostrare gli errori sul browser. In questo caso dovremo impostare `log_errors` ad 'on' ed indicare in `error_log` il nome del file in cui vogliamo registrare gli errori (con percorso completo sulla nostra macchina, ad esempio 'C:\Php\errori.txt'). Ricordiamoci che dobbiamo anche creare questo file, altrimenti PHP non riuscirà a memorizzare i suoi errori! E' anche importante che il web server abbia il permesso per scrivere su questo file.
- **register_globals**: questo è il parametro di cui si parla di più da quando, con la versione 4.2.0 di PHP, la sua impostazione di default è diventata 'off'. Tale impostazione non ci permette di utilizzare come variabili globali quelle che arrivano dalla query string o dai moduli inviati dagli utenti. Chi era abituato ad utilizzare PHP da prima di questa versione può trovarsi in difficoltà quando passa ad una versione successiva ignorando questo particolare. Un altro caso in cui si possono avere problemi è se si utilizzano script, scaricati da qualche risorsa su internet, che sono stati concepiti per utilizzare queste variabili come globali. In questo caso può diventare necessario modificare questa impostazione a 'on', ma il nostro consiglio è quello di evitarlo, fin quando possibile. Infatti l'utilizzo degli array `$_GET` e `$_POST` invece delle variabili globali conferisce maggiore chiarezza ai nostri script, e può eliminare qualche problema di sicurezza.

- **extension_dir**: questo parametro va impostato quando vogliamo utilizzare le estensioni di PHP, ad esempio le librerie gd per lavorare con le immagini, oppure le librerie pdf per manipolare documenti pdf, o ancora quando vogliamo collegarci (non attraverso ODBC) a database diversi da MySQL, come PostgreSQL, Oracle, Sql Server ecc. Tali funzioni richiedono infatti la presenza di apposite librerie (in Windows si tratta di files .dll che vengono distribuiti insieme al pacchetto PHP e si trovano nella cartella 'extensions'), ed in questo parametro dobbiamo indicare il percorso sulla nostra macchina della directory in cui le abbiamo sistemate.
- **extension**: questo parametro è collegato al precedente, perché ci serve per indicare quali sono le estensioni che PHP deve caricare. Dovremo valorizzarne uno per ogni estensione che ci interessa. Il file php.ini possiede già una lista precompilata di queste estensioni, che però sono tutte 'commentate' (cioè precedute da punto e virgola): ci basterà quindi togliere il punto e virgola da quelle che vogliamo caricare. Ricordiamoci che, una volta tolto il punto e virgola, PHP pretenderà di trovare la libreria corrispondente nella cartella che abbiamo indicato con 'extension_dir': nel caso in cui non la trovi, avremo una segnalazione di errore all'avvio del web server.
- **session.save_path**: questo è l'ostacolo sul quale si infrangono molti di coloro che tentano per la prima volta di utilizzare le sessioni. Qui infatti dobbiamo indicare la cartella nella quale PHP andrà a salvare i files di sessione: di default vi è impostato il valore '/tmp', ma possiamo indicare quello che ci fa più comodo (ad es. 'C:\Php\sessioni'). In alternativa, se non vogliamo modificarlo e lavoriamo sotto Windows, dovremo creare una cartella di nome 'tmp' sotto la directory principale del disco C. E' importante tenere presente che il web server dovrà avere il permesso di scrivere in questa cartella.

Questi sono solo alcuni dei parametri configurabili attraverso php.ini. Man mano che matureremo esperienza nella programmazione in PHP, impareremo ad capirne ed utilizzarne anche altri. E' però importante ricordare che, quando effettuiamo una modifica a questo file, essa non avrà effetti fino a quando non avremo riavviato il web server.

Inserire codice PHP in HTML (Hypertext Markup Language)

PHP è un linguaggio la cui funzione fondamentale è quella di produrre codice HTML, che è quello dal quale sono formate le pagine web. Ma visto che PHP è un linguaggio di programmazione, abbiamo la possibilità di analizzare diverse situazioni (l'input degli utenti, i dati contenuti in un database) e di decidere, di conseguenza, di produrre codice HTML condizionato ai risultati dell'elaborazione.

Questo è, in parole povere, il Web dinamico.

Quando il server riceve una richiesta per una pagina php, la fa analizzare dall'interprete del PHP, il quale restituisce un file contenente solo il codice che deve essere inviato al browser (in linea di massima HTML, ma può esserci anche codice JavaScript, fogli di stile css eccetera).

La prima cosa da sapere è come fa PHP (inteso come interprete) a riconoscere il codice php contenuto nel file che sta analizzando. Il codice php infatti deve essere compreso fra appositi tag di apertura e di chiusura, che sono i seguenti:

```
<? //tag di apertura ..... ?> //tag di chiusura  
<?php //tag di apertura ..... ?> //tag di chiusura  
<SCRIPT LANGUAGE="php"> //tag di apertura ..... </SCRIPT> //tag di chiusura
```

La configurazione dei tag può essere modificata impostando opportunamente il *file di configurazione* **PHP.INI** utilizzando la direttiva **short_open_tag**

Tutto ciò che è contenuto fra questi tag deve corrispondere alle regole sintattiche del PHP, ed è codice che sarà eseguito dall'interprete e **non sarà inviato al browser**. Per generare il codice da inviare al browser abbiamo due costrutti del linguaggio, che possiamo considerare equivalenti, che sono: print() e echo().

Vediamo un primo esempio:

Esempio n.1 :

```
<html>  
<body>  
  
<?echo ("Questa frase sarà inviata al browser"); ?>  
<?print ("Anche questa frase sarà inviata al browser"); ?>  
  
</body>  
</html>
```

Esempio n.2 :

```
.....  
<?php echo "Visualizza prima riga"; echo "Visualizza seconda riga"; ?>
```

Avrete forse notato che in fondo ad ogni istruzione php abbiamo messo un punto e virgola; infatti la sintassi del PHP prevede che il punto e virgola debba obbligatoriamente chiudere ogni istruzione.

Da quanto abbiamo detto finora emerge una realtà molto importante: chi vuole avvicinarsi al PHP deve già avere una conoscenza approfondita di HTML e di tutto quanto può far parte di una pagina web. Questo perché lo scopo di PHP è proprio la produzione di questi codici.

Commenti

Chi ha esperienza, anche minima, di programmazione, sa bene che molte volte i commenti, sebbene spesso sottovalutati, si rivelano di importanza decisiva quando si tratta di mettere le mani su un programma realizzato da qualcun altro, e anche quando il programma è stato scritto da noi stessi, soprattutto se è passato qualche tempo dalla realizzazione.

Abbiamo tre diverse possibilità per posizionare i commenti all'interno del nostro codice: la prima è l'uso dei commenti in stile C++, caratterizzati da due barre:

```
// Commento in stile C++
```

La seconda sono i commenti in stile Perl, contraddistinti dall'uso del cancelletto:

```
# Commento in stile Perl
```

Entrambi questi tipi di commenti sono limitati ad una sola riga: l'interprete PHP, quando trova le barre o il cancelletto, salta tutto ciò che si trova da quel punto fino al termine della riga. Questo ci permette di porre il commento anche sulla stessa riga del codice commentato, così:

```
print ('Buongiorno a tutti<br>'); //stampo un messaggio di saluto  
print ('Esclusi quelli antipatici'); # faccio una precisazione  
// Questa riga contiene solo commento
```

Il terzo tipo di commento (in stile C) prevede un codice di apertura e un codice di chiusura, e ci permette quindi di estendere il commento su più righe: attenzione però a non dimenticare il codice di chiusura, anche se il commento occupa una sola riga, altrimenti tutto il nostro codice php successivo verrà considerato commento e non eseguito!

Il commento in stile C inizia con una barra seguita da un asterisco, e termina con un asterisco seguito da una barra:

```
print ('Buonasera<br>'); /* Questo è un commento in stile C,  
che può occupare più righe ma deve essere chiuso  
con l'apposito simbolo */
```

La scelta di quale tipo di commento utilizzare è puramente soggettiva, e dipende esclusivamente dalle nostre preferenze personali; evidentemente potrà essere influenzata da eventuali precedenti esperienze di programmazione e dalle relative abitudini.

Variabili

Le variabili sono alcuni dei componenti fondamentali di qualsiasi linguaggio di programmazione, in quanto ci consentono di trattare i dati del nostro programma senza sapere a priori quale sarà il loro valore. Possiamo immaginare una variabile come una specie di contenitore all'interno del quale viene conservato il valore che ci interessa, e che può cambiare di volta in volta.

In PHP possiamo scegliere il nome delle variabili usando lettere, numeri ed il trattino di sottolineatura, o *underscore* (`_`). Il primo carattere del nome deve essere però una lettera o un underscore (non un numero). Dobbiamo inoltre ricordarci che il nome delle variabili è sensibile all'uso delle maiuscole e delle minuscole: di conseguenza, se scriviamo due volte un nome di variabile usando le maiuscole in maniera differente, per PHP si tratterà di due variabili distinte! Nello script PHP il nome delle variabili è preceduto dal simbolo del dollaro (`$`).

PHP ha una caratteristica che lo rende molto più flessibile rispetto ad altri linguaggi di programmazione: non richiede, infatti, che le variabili vengano dichiarate prima del loro uso. Possiamo quindi permetterci di riferirci ad una variabile direttamente con la sua valorizzazione:

```
$a = 5;
```

Questo semplicissimo codice definisce la variabile 'a', assegnandole il valore 5. In fondo all'istruzione abbiamo il punto e virgola, che (ricordate?) deve chiudere tutte le istruzioni PHP.

Tipi di dato

Valore booleano. Le variabili booleane sono le più semplici: il loro valore può essere TRUE o FALSE (vero o falso). Vediamo un rapido esempio:

```
$vero = TRUE;  
$falso = FALSE;
```

Intero. Un numero intero, positivo o negativo, il cui valore massimo (assoluto) può variare in base al sistema operativo su cui gira PHP, ma che generalmente si può considerare, per ricordarlo facilmente, di circa 2 miliardi (2 elevato alla 31esima potenza).

```
$int1 = 129;  
$int2 = -715;  
$int3 = 5 * 8; // $int3 vale 40
```

Virgola mobile. Un numero decimale (a volte citato come "double" o "real"). Attenzione: **per indicare i decimali non si usa la virgola, ma il punto!** Anche in questo caso la dimensione massima dipende dalla piattaforma. Normalmente comunque si considera un massimo di circa $1.8e308$ con una precisione di 14 cifre decimali. Si possono utilizzare le seguenti sintassi:

```
$vm1 = 4.153; // 4,153  
$vm2 = 3.2e5; //  $3,2 * 10^5$ , cioè 320.000  
$vm3 = 4E-8; //  $4 * 10^{-8}$ , cioè  $4/100.000.000 = 0,00000004$ 
```

Stringa. Una stringa è un qualsiasi insieme di caratteri, senza limitazione. Le stringhe possono essere espresse in tre maniere:

- Delimitate da apici (singoli)
- Delimitate da virgolette (doppie)
- Con la sintassi heredoc

Le stringhe delimitate da apici sono la forma più semplice, consigliata quando all'interno della stringa non vi sono variabili di cui vogliamo ricavare il valore:

```
$frase = 'Anna disse: "Ciao a tutti!" ma nessuno rispose';  
print $frase;
```

Questo codice stamperà la frase: 'Anna disse: "Ciao a tutti!" ma nessuno rispose'.

Le virgolette ci consentono di usare le stringhe in una maniera più sofisticata, in quanto, se all'interno della stringa delimitata da virgolette PHP riconosce un nome di variabile, lo sostituisce con il valore della variabile stessa (si dice in questo caso che la variabile viene risolta).

```
$nome = 'Anna';  
print "$nome è simpatica... a pochi"; /*stampa: Anna è simpatica... a pochi*/  
print '$nome è simpatica... a pochi'; /*stampa: $nome è simpatica... a pochi*/
```

Come vedete, l'uso delle virgolette permette di risolvere le variabili, mentre con gli apici i nomi di variabile rimangono... invariati.

Ci sono un paio di regole molto importanti da ricordare quando si usano le stringhe delimitate da apici o virgolette: siccome può capitare che una stringa debba contenere a sua volta un apice o un paio di virgolette, abbiamo bisogno di un sistema per far capire a PHP che quel carattere fa parte della stringa e non è il suo delimitatore. In questo caso si usa il cosiddetto 'carattere di escape', cioè la barra rovesciata (backslash: \). Vediamo alcuni esempi:

```
print 'Torniamo un'altra volta'; //stampa: Torniamo un'altra volta
print "Torniamo un'altra volta"; //stampa: Torniamo un'altra volta
print "Torniamo un'altra volta"; //stampa: Torniamo un'altra volta
print 'Torniamo un'altra volta'; /*causa un errore, perché l'apostrofo viene scambiato per l'apice di chiusura*/
print 'Anna disse "Ciao" e se ne andò'; /*stampa: Anna disse "Ciao" e se ne andò*/
print "Anna disse \"Ciao\" e se ne andò"; /*stampa: Anna disse "Ciao" e se ne andò*/
print 'Anna disse \"Ciao\" e se ne andò'; /*stampa: Anna disse \"Ciao\" e se ne andò*/
print "Anna disse "Ciao" e se ne andò"; //errore
```

Da questi esempi si può capire che il backslash deve essere utilizzato come carattere di escape quando vogliamo includere nella stringa lo stesso tipo di carattere che la delimita; se mettiamo un backslash davanti alle virgolette in una stringa delimitata da apici (o viceversa), anche il backslash entrerà a far parte della stringa stessa, come si vede nel terzo e nel settimo esempio. Il backslash viene usato anche come 'escape di sé stesso', nei casi in cui vogliamo esplicitamente includerlo nella stringa:

```
print ("Questo: '\\\\' è un backslash"); /*stampa: Questo: "\" è un backslash*/
print ('Questo: '\\\' è un backslash'); /*stampa: Questo: \' è un backslash*/
print ("Questo: \' è un backslash"); /*stampa: Questo: \' è un backslash*/
print ("Questo: '\\ è un backslash"); /*stampa: Questo: \' è un backslash*/
```

Analizziamo il primo esempio: il primo backslash fa l'escape del primo paio di virgolette; il secondo backslash fa l'escape del terzo, che quindi viene incluso nella stringa; il quarto fa l'escape del secondo paio di virgolette. Il secondo esempio equivale al primo, con l'uso degli apici al posto delle virgolette.

Negli ultimi due casi non è necessario fare l'escape del backslash, in quanto il backslash che vogliamo stampare non può essere scambiato per un carattere di escape (infatti vicino ad esso ci sono degli apici, che in una stringa delimitata da virgolette non hanno bisogno di escape). Di conseguenza, fare o non fare l'escape del backslash in questa situazione è la stessa cosa, e difatti i due esempi forniscono lo stesso risultato.

Passiamo ad esaminare l'ultimo modo di rappresentare le stringhe: la sintassi heredoc. Questa ci consente di delimitare una stringa con i caratteri <<< seguiti da un identificatore (in genere si usa 'EOD', ma è solo una convenzione: è possibile utilizzare qualsiasi stringa composta di caratteri alfanumerici e underscore, di cui il primo carattere deve essere non numerico: la stessa regola dei nomi di variabile). Tutto ciò che segue questo delimitatore viene considerato parte della stringa, fino a quando non viene ripetuto l'identificatore seguito da un punto e virgola. **Attenzione:** l'identificatore di chiusura deve occupare una riga a sé stante, deve iniziare a colonna 1 e non deve contenere nessun altro carattere (nemmeno spazi vuoti) dopo il punto e virgola.

```
$nome = "Paolo";  
$stringa = <<<EOD  
Il mio nome è $nome  
EOD;  
print $stringa;
```

Questo codice stamperà 'Il mio nome è Paolo'. Infatti la sintassi heredoc risolve i nomi di variabile così come le virgolette. Rispetto a queste ultime, con questa sintassi abbiamo il vantaggio di poter includere delle virgolette nella stringa senza farne l'escape:

```
$frase = "ciao a tutti";  
$stringa = <<<EOT  
Il mio saluto è "$frase"  
EOT;  
print $stringa;
```

In questo caso stamperemo 'Il mio saluto è "ciao a tutti"'.

Array. Possiamo considerare un array (vettore) come una variabile complessa, che contiene cioè non un solo valore, ma una serie di valori, ciascuno dei quali caratterizzato da una chiave, o indice. Facciamo un primo esempio, definendo un array composto di cinque valori:

```
$colori = array('bianco', 'nero', 'giallo', 'verde', 'rosso');
```

A questo punto ciascuno dei nostri cinque colori è caratterizzato da un indice numerico, che PHP assegna automaticamente a partire da 0. L'indice viene indicato fra parentesi quadre dietro al nome dell'array:

```
print $colori[1]; //stampa 'nero'  
print $colori[4]; //stampa 'rosso'
```

Oggetto. L'uso degli oggetti si rifà alla programmazione ad oggetti, e va al di là degli scopi di questo corso.

Eliminare una variabile. In alcune situazioni ci può capitare di avere la necessità di eliminare una variabile: in questo caso PHP ci mette a disposizione l'istruzione unset():

```
unset($variabile);
```

Dopo l'istruzione unset, sarà come se la variabile non fosse mai esistita.

Gli Array

Nella lezione precedente abbiamo accennato al fatto che un array (o vettore) è una variabile "complessa", cioè che contiene molteplici valori invece di uno solo. Abbiamo anche visto una prima maniera per definire un array, elencando i suoi valori separati da virgole:

```
$colori = array('bianco', 'nero', 'giallo', 'verde', 'rosso');
```

Usando questo tipo di definizione, PHP associa a ciascuno dei valori che abbiamo elencato un indice numerico, a partire da 0. Quindi, in questo caso, 'bianco' assumerà l'indice 0, 'nero' l'indice 1, e così via fino a 'rosso' che avrà indice 4. Per riferirsi ad un singolo elemento dell'array si indica il nome dell'array seguito dall'indice contenuto fra parentesi quadre:

```
print $colori[2]; //stampa 'giallo'
```

Esiste poi un metodo per aggiungere un valore all'array; questo metodo può essere usato anche, come alternativa al precedente, per definire l'array:

```
$colori[] = 'blu';
```

Con questo codice verrà creato un nuovo elemento nell'array \$colori, che avrà l'indice 5. Questa sintassi infatti può essere "tradotta" come "aggiungi un elemento in fondo all'array \$colori". Come abbiamo detto, questa sintassi è valida anche per definire un array, in alternativa a quella usata prima: infatti, se ipotizziamo che l'array \$colori non fosse ancora definito, questa istruzione lo avrebbe definito creando l'elemento con indice 0. E' naturalmente possibile anche indicare direttamente l'indice: ad esempio

```
$colori[3] = 'arancio';  
$colori[7] = 'viola';
```

Dopo questa istruzione, l'elemento con indice 3, che prima valeva 'verde', avrà il valore cambiato in 'arancio'. Inoltre avremo un nuovo elemento, con indice 7, con il valore 'viola'. E' da notare che, dopo queste istruzioni, il nostro array ha un "buco", perché dal codice 5 si salta direttamente al codice 7: successivamente, se useremo di nuovo l'istruzione di "incremento" con le parentesi quadre vuote, il nuovo elemento prenderà l'indice 8. Infatti PHP, quando gli proponiamo un'istruzione di quel tipo, va a cercare l'elemento con l'indice più alto, e lo aumenta di 1 per creare quello nuovo.

Ma l'argomento array non si limita a questo: infatti gli indici degli elementi non sono necessariamente numerici. Possono essere anche delle stringhe:

```
$persona['nome'] = 'Mario';
```

Con questa istruzione abbiamo definito un array di nome \$persona, creando un elemento la cui chiave è 'nome' ed il cui valore è 'Mario'.

Attenzione: capita qualche volta di leggere, o sentir dire, che PHP gestisce due tipi di array: numerici ed associativi. Per array associativi si intendono quelli che hanno chiavi (o indici) in formato stringa, come nell'ultimo esempio che abbiamo visto. In realtà però questo **non è esatto**: infatti PHP gestisce **un unico** tipo di array, le cui chiavi possono essere numeriche o associative. La differenza è sottile, ma significativa: infatti le chiavi numeriche ed associative possono

coesistere **nello stesso array**. Vediamo un esempio banale, ipotizzando la formazione di una squadra di calcio:

```
$formazione[1] ='Buffon';  
$formazione[2] ='Panucci';  
$formazione[3] ='Nesta';  
$formazione[4] ='Cannavaro';  
$formazione[5] ='Coco';  
$formazione[6] ='Ambrosini';  
$formazione[7] ='Tacchinardi';  
$formazione[8] ='Perrotta';  
$formazione[9] ='Totti';  
$formazione[10] ='Inzaghi';  
$formazione[11] ='Vieri';  
$formazione['ct'] = 'Trapattoni';
```

In questo caso abbiamo creato un array con dodici elementi, di cui undici con chiavi numeriche, ed uno con chiave associativa. Se in seguito volessimo aggiungere un elemento usando le parentesi quadre vuote, il nuovo elemento prenderà l'indice 12. Avremmo potuto creare lo stesso array usando l'istruzione di dichiarazione dell'array, così:

```
$formazione = array(1 => 'Buffon', 'Panucci', 'Nesta', 'Cannavaro', 'Coco', 'Ambrosini',  
'Tacchinardi', 'Perrotta', 'Totti', 'Inzaghi', 'Vieri', 'ct' => 'Trapattoni');
```

Analizziamo il formato di questa istruzione: per prima cosa abbiamo creato il primo elemento, assegnandogli esplicitamente la chiave 1. Come possiamo vedere, il sistema per fare ciò è di indicare la chiave, seguita dal simbolo "=>" (uguale + maggiore) e dal valore dell'elemento. Se non avessimo indicato 1 come indice, PHP avrebbe assegnato al primo elemento l'indice 0. Per gli elementi successivi, ci siamo limitati ad elencare i valori, in quanto PHP, per ciascuno di essi, crea la chiave numerica aumentando di 1 la più alta già esistente. Quindi 'Panucci' prende l'indice 2, 'Nesta' il 3 e così via. Arrivati all'ultimo elemento, siccome vogliamo assegnargli una chiave associativa, siamo obbligati ad indicarla esplicitamente.

E' da notare che quando abbiamo usato le chiavi associative le abbiamo indicate fra apici: ciò è necessario per mantenere la 'pulizia' del codice, in quanto, se non usassimo gli apici (come spesso si vede fare), PHP genererebbe un errore di tipo 'notice', anche se lo script funzionerebbe ugualmente. L'eccezione si ha quando l'elemento di array viene indicato fra virgolette:

```
$persona['nome'] = 'Mario'; //corretto  
$persona[cognome] = 'Rossi'; /*funziona, ma genera un errore 'notice'*/  
print $persona[cognome]; //stampa 'Rossi': corretto  
print "ciao $persona[nome]"; /*stampa 'ciao Mario': corretto (niente apici fra virgolette)*/  
print "ciao $persona['nome']"; //NON FUNZIONA, GENERA ERRORE  
print "ciao {$persona[nome]}"; /*corretto: per usare gli apici fra virgolette dobbiamo comprendere  
il tutto fra parentesi graffe*/  
print "ciao " . $persona[nome]; /*corretto: come alternativa, usiamo il punto per concatenare (v.  
lez.10 sugli operatori)*/
```

Abbiamo così visto in quale maniera possiamo creare ed assegnare valori agli array, usando indici numerici o associativi, impostando esplicitamente le chiavi o lasciando che sia PHP ad occuparsene. Vediamo ora in che modo possiamo creare strutture complesse di dati, attraverso gli array a più dimensioni.

Un array a più dimensioni è un array nel quale uno o più elementi sono degli array a loro volta. Supponiamo di voler raccogliere in un array i dati anagrafici di più persone: per ogni persona registreremo nome, cognome, data di nascita e città di residenza

```
$persone = array( array('nome' => 'Mario', 'cognome' => 'Rossi', 'data_nascita' => '1973/06/15',  
'residenza' => 'Roma'), array('nome' => 'Paolo', 'cognome' => 'Bianchi', 'data_nascita' =>  
'1968/04/05', 'residenza' => 'Torino'), array('nome' => 'Luca', 'cognome' => 'Verdi', 'data_nascita' =>  
'1964/11/26', 'residenza' => 'Napoli'));  
print $persone[0]['cognome']; // stampa 'Rossi'  
print $persone[1]['residenza']; // stampa 'Torino'  
print $persone[2]['nome']; // stampa 'Luca'
```

Con questo codice abbiamo definito un array formato a sua volta da tre array, che sono stati elencati separati da virgole, per cui ciascuno di essi ha ricevuto l'indice numerico a partire da 0. All'interno dei singoli array, invece, tutte le chiavi sono state indicate come associative. Da notare che, sebbene in questo caso ciascuno dei tre array 'interni' abbia la stessa struttura, in realtà è possibile dare a ciascun array una struttura autonoma. Vediamo un altro esempio:

```
$persone = array( 1 => array('nome' => 'Mario Rossi', 'residenza' => 'Roma', 'ruolo' => 'impiegato'),  
2 => array('nome' => 'Paolo Bianchi', 'data_nascita' => '1968/04/05', 'residenza' => 'Torino'),  
'totale_elementi' => 2);  
print $persone[1]['residenza']; // stampa 'Roma'  
print $persone['totale_elementi']; // stampa '2'
```

In questo caso il nostro array è formato da due array, ai quali abbiamo assegnato gli indici 1 e 2, e da un terzo elemento, che non è un array ma una variabile intera, con chiave associativa 'totale_elementi'. I due array che costituiscono i primi due elementi hanno una struttura diversa: mentre il primo è formato dagli elementi 'nome', 'residenza' e 'ruolo', il secondo è formato da 'nome', 'data_nascita' e 'residenza'.

Contare gli elementi di un array. Se vogliamo sapere di quanti elementi è composto un array, possiamo utilizzare la funzione count():

```
$numero_elementi = count($formazione);
```

Nell'esempio visto prima, la variabile \$numero_elementi assumerà il valore 12: infatti abbiamo 11 elementi con chiavi numeriche e 1 ('ct') con la chiave associativa.

Concludiamo questa lezione sugli array vedendo, come già abbiamo fatto per le variabili, in che modo possiamo eliminare un intero array o soltanto un suo elemento.

```
unset($colori[2]); // elimina l'elemento 'giallo'  
unset($formazione); // elimina l'intero array $formazione
```

Con la prima di queste due istruzioni abbiamo eliminato l'elemento con chiave 2 dall'array \$colori. Questo creerà un 'buco' nell'array, che passerà così dall'elemento 1 all'elemento 3. Con la seconda istruzione invece eliminiamo l'intero array \$formazione.

Gli operatori

Gli operatori sono un altro degli elementi di base di qualsiasi linguaggio di programmazione, in quanto ci consentono non solo di effettuare le tradizionali operazioni aritmetiche, ma più in generale di manipolare il contenuto delle nostre variabili. Il più classico e conosciuto degli operatori è quello di assegnazione:

```
$nome = 'Giorgio';
```

Il simbolo '=' serve infatti ad **assegnare** alla variabile \$nome il valore 'Giorgio'. In generale, possiamo dire che con l'operatore di assegnazione prendiamo ciò che sta alla destra del segno '=' ed assegniamo lo stesso valore a ciò che sta a sinistra. Potremmo ad esempio assegnare ad una variabile il valore di un'altra variabile:

```
$a = 5;  
$b = $a;
```

Con la prima istruzione assegniamo ad \$a il valore 5, con la seconda assegniamo a \$b lo stesso valore di \$a.

Altri operatori molto facili da comprendere sono quelli aritmetici: addizione, sottrazione, divisione, moltiplicazione, modulo.

```
$a = 3 + 7; //addizione  
$b = 5 - 2; //sottrazione  
$c = 9 * 6; //moltiplicazione  
$d = 8 / 2; //divisione  
$e = 7 % 4; /*modulo (il modulo è il resto della divisione, quindi in questo caso 3)*/
```

Uno degli operatori più utilizzati è quello che serve per concatenare le stringhe: il punto.

```
$nome = 'pippo';  
$stringa1 = 'ciao ' . $nome; //stringa1 vale 'ciao pippo'
```

Con l'operatore di assegnazione si può anche usare una variabile per effettuare un calcolo il cui risultato deve essere assegnato alla variabile stessa. Ad esempio, supponiamo di avere una variabile di cui vogliamo aumentare il valore:

```
$a = $a + 10; //il valore di $a aumenta di 10
```

Con questa istruzione, viene eseguito il calcolo che sta alla destra del segno '=' ed il risultato viene memorizzato nella variabile indicata a sinistra. Quindi è chiaro che il valore di tale variabile prima dell'istruzione viene utilizzato per il calcolo, ma dopo che l'istruzione è stata eseguita questo valore è cambiato. Un risultato di questo tipo si può ottenere anche con gli operatori di assegnazione combinati, che ci permettono di 'risparmiare' sul codice:

```
$x += 4; //incrementa $x di 4 (equivale a $x = $x + 4)  
$x -= 3; //decrementa $x di 3 (equivale a $x = $x - 3)  
$x .= $a; /*il valore della stringa $a viene concatenato a $x (equivale a $x = $x . $a)*/  
$x /= 5; //equivale a $x = $x / 5  
$x *= 4; //equivale a $x = $x * 4  
$x %= 2; //equivale a $x = $x % 2
```

Userete gli ultimi tre molto di rado. Invece il terzo (l'operatore di concatenamento) è probabilmente il più usato di tutti, perché risulta molto utile quando abbiamo bisogno di costruire stringhe piuttosto lunghe. Anche i primi due possono essere utilizzati in determinate situazioni. Fra l'altro,

per quanto riguarda incremento e decremento abbiamo altri operatori ancora più sintetici che si possono utilizzare per incrementare o decrementare i valori di una unità:

```
$a++; /*incrementa di 1: equivale ad $a = $a + 1, o $a += 1*/  
++$a; /*stessa cosa, ma con una differenza nella valutazione dell'espressione (v. lezione 11 sulle  
espressioni)*/  
$a--; //decrementa di 1: equivale ad $a = $a - 1, o $a -= 1  
--$a; //anche qui stessa cosa, con la stessa differenza di cui sopra
```

Gli operatori di confronto. Gli operatori di confronto sono fondamentali perché ci permettono, effettuando dei confronti fra valori, di prendere delle decisioni, cioè di far svolgere al nostro script determinate operazioni invece di altre. Quando utilizziamo gli operatori di confronto, confrontiamo i due valori posti a sinistra e a destra dell'operatore stesso. Il risultato di questa operazione sarà, ogni volta, vero (true) o falso (false). Questi operatori sono:

- == : uguale
- != : diverso
- === : identico (cioè uguale e dello stesso tipo: ad esempio per due variabili di tipo intero)
- > : maggiore
- >= : maggiore o uguale
- < : minore
- <= : minore o uguale

Vediamo alcuni esempi:

```
$a = 7; $b = 7.0; $c = 4; //assegniamo valori a tre variabili  
$a == $b; //vero  
$a == $c; //falso  
$a === $b; //falso, perché $a è intero mentre $b è float  
$a > $c; //vero  
$c >= $a; //falso, $c è minore di $a  
$a < $b; //falso, hanno lo stesso valore  
$c <= $b; //vero
```

Una piccola osservazione sul terzo confronto: siccome abbiamo assegnato il valore di \$b usando la notazione col punto decimale, per PHP \$b è una variabile del tipo in virgola mobile, anche se in realtà il suo valore è intero. Per questo il confronto di identità restituisce falso.

Fino a qui abbiamo visto comunque casi molto semplici, perché tutte le variabili avevano valori numerici. Gli stessi confronti però si possono fare anche con altri tipi di variabili, ed in particolare con le stringhe. In questo caso il confronto viene fatto basandosi sull'ordine alfabetico dei caratteri: vale a dire che vengono considerati 'minori' i caratteri che 'vengono prima' nell'ordine alfabetico. Quindi 'a' è minore di 'b', 'b' è minore di 'c', eccetera. Inoltre tutte le lettere minuscole sono 'maggiori' delle lettere maiuscole, e tutte, maiuscole e minuscole, sono 'maggiori' delle cifre da 0 a 9:

```
$a = 'Mario'; $b = 'Giorgio'; $c = 'Giovanni'; $d = 'antonio'; $e = '4 gatti';  
$a < $b; //falso, la 'G' precede la 'M'  
$b < $c; //vero, la 'r' ('Gior') precede la 'v' ('Giov')  
$d > $a; /*vero, la 'a' minuscola è 'maggior' di qualsiasi lettera maiuscola*/  
$c > $e; //vero, ogni lettera è 'maggior' di qualsiasi cifra
```

In realtà confronti di questo tipo hanno un significato abbastanza relativo, e difficilmente vi capiterà di utilizzarli. Un caso al quale invece bisogna prestare attenzione, soprattutto perché può essere fonte di confusione, è quello nel quale vengono confrontati una variabile numerica ed una stringa. In questo caso, infatti, PHP, per rendere i valori confrontabili, assegna alla stringa un valore

numerico, e così effettua il confronto fra due valori numerici, restituendo il risultato di conseguenza. Per assegnare questo valore numerico, PHP controlla se **all'inizio** della stringa ci sono dei numeri: se ne trova, considererà tutti i numeri che trova inizialmente come il valore numerico di quella stringa. Se non ne trova, il valore della stringa sarà 0:

```
$a = 7; $b = 5; $c='molte persone'; $d='7 persone'; $e='5';  
$a == $d; //vero, $d vale 7  
$a === $d; /*falso, valgono entrambi 7 ma $a è un intero mentre $d è una stringa*/  
$b > $c; //vero, $b vale 5 mentre $c vale 0  
$e > $c; /*falso: questo è un confronto fra due stringhe, quindi valgono le regole viste prima*/
```

Prestiamo attenzione all'ultimo esempio: il valore di \$e era stato assegnato usando gli apici, e questo fa sì che PHP lo consideri una stringa anche se il contenuto è un numero. Il confronto fra un numero e una stringa può avvenire in maniera voluta, ma è più probabile che avvenga per caso, quando cioè una variabile che pensavamo contenesse un numero contiene in realtà una stringa. E' evidente che in questo caso potremo facilmente ottenere un risultato diverso da quello che ci aspettavamo, o, viceversa, potremmo ottenere casualmente il risultato atteso: in quest'ultima situazione è possibile che risultati inaspettati arrivino più avanti nello script, se utilizzeremo di nuovo la stessa variabile. In tutte queste situazioni, tener presente il modo in cui PHP tratta questi confronti può essere di aiuto per spiegarci comportamenti apparentemente bizzarri del nostro script.

Gli operatori logici. Con gli operatori logici possiamo combinare più valori booleani, oppure negarne uno (nel caso di NOT). Questi valori sono:

- **or**: valuta se almeno uno dei due operatori è vero; si può indicare con 'Or' oppure '||'
- **and**: valuta se entrambi gli operatori sono veri; si indica con 'And' o '&&'
- **xor**: viene chiamato anche 'or esclusivo', e valuta se **uno solo** dei due operatori è vero: l'altro deve essere falso; si indica con 'Xor'
- **not**: vale come negazione e si usa con un solo operatore: in pratica è vero quando l'operatore è falso, e viceversa; si indica con '!'

Anche in questa occasione vediamo qualche esempio:

```
10 > 8 And 7 < 6; /*falso, perché la prima condizione è vera ma la seconda è falsa*/  
10 > 8 Or 7 < 6; //vero  
9 > 5 And 5 == 5; //vero: entrambe le condizioni sono vere  
9 > 5 Xor 5 == 5; /*falso: solo una delle due deve essere vera perché si verifichi lo 'Xor'*/  
4 < 3 || 7 > 9; //falso: nessuna delle due condizioni è vera  
6 == 6 && 1 > 4; //falso: solo la prima condizione è vera
```

Per quanto riguarda gli operatori 'and' e 'or', le due diverse notazioni differiscono per il livello di precedenza in caso di espressioni complesse. Infatti, siccome è possibile combinare molti operatori in espressioni anche assai complicate, è necessario sapere con quale ordine PHP valuta i diversi operatori. Queste regole ricalcano le regole algebriche in base alle quali moltiplicazioni e divisioni hanno la precedenza su addizioni e sottrazioni, ma sono più complesse perché devono considerare anche gli altri operatori. Vediamo quindi qual è l'ordine di priorità dei diversi operatori, iniziando da quelli che hanno la priorità maggiore:

1. Operatori di incremento e decremento (++ --)
2. Moltiplicazione, divisione, modulo (* / %)
3. Addizione e sottrazione (+ -)
4. Operatori di confronto per minore e maggiore (< <= > >=)
5. Operatori di confronto per uguaglianza e disuguaglianza (== === !=)
6. Operatore logico 'and', nella notazione col simbolo (&&)
7. Operatore logico 'or', nella notazione col simbolo (||)
8. Operatori di assegnazione, compresi quelli 'sintetici' (= += -= /= *= %= .=)

9. Operatore logico 'and', nella notazione letterale (And)
10. Operatore logico 'xor' (Xor)
11. Operatore logico 'or', nella notazione letterale (Or)

Abbiamo già visto prima, in occasione degli esempi sugli operatori logici, l'applicazione di questi principi di precedenza: infatti in tutte quelle espressioni venivano valutati prima gli operatori di confronto e, solo dopo, quelli logici. Un'altra classica rappresentazione di esempio è quella dell'espressione algebrica:

$5 + 4 * 2$; /*questa espressione vale 13 e non 18, perché la moltiplicazione viene eseguita prima*/
 $(5 + 4) * 2$; /*questa invece vale 18, perché le parentesi modificano l'ordine di esecuzione*/

Come abbiamo visto, così come avviene in algebra, usando le parentesi possiamo determinare a nostro piacere quali operatori devono essere valutati per primi. Per questo motivo, sebbene sia possibile imparare a memoria l'ordine di precedenza che abbiamo visto poco fa, il nostro consiglio è quello di non tenerne conto, e di utilizzare sempre le parentesi quando abbiamo bisogno di costruire un'espressione un po' complessa: in questo modo ridurremo il rischio di errori, e soprattutto renderemo il nostro codice molto più leggibile. Infatti leggere un'espressione regolata dalle parentesi è molto più immediato che non doversi ricordare quali degli operatori hanno la precedenza sugli altri.

Comandi Condizionale

Con le strutture di controllo andiamo ad analizzare un altro degli aspetti fondamentali della programmazione: la possibilità cioè di eseguire operazioni diverse, ed eventualmente di eseguirle più volte, in base alla valutazione di determinate condizioni. In questa lezione esamineremo le istruzioni che ci permettono di eseguire o non eseguire certe porzioni di codice.

La principale di queste istruzioni è la 'if', la cui sintassi più elementare è la seguente:

```
if ($nome == 'Luca')  
    print "ciao Luca!<br>";
```

Dopo l'if, deve essere indicata fra parentesi un'espressione da valutare (condizione). Questa espressione verrà valutata in senso booleano, cioè il suo valore sarà considerato vero o falso. Nel caso in cui l'espressione non abbia un valore booleano, PHP convertirà comunque questo valore in booleano (fra poco vedremo come). Dunque, se la condizione è vera, l'istruzione successiva viene eseguita. In caso contrario, viene ignorata. Potremmo anche avere la necessità di eseguire, nel caso in cui la condizione sia vera, non una sola ma più istruzioni. Questo è perfettamente possibile, ma dobbiamo ricordarci di comprendere questo blocco di istruzioni fra due parentesi graffe, ad esempio così:

```
if ($nome == 'Luca') {  
    print "ciao Luca!<br>";  
    print "dove sono i tuoi amici?<br>";  
}  
print "ciao a tutti voi";
```

In questo modo, se la variabile \$nome ha effettivamente il valore 'Luca' vengono eseguite le due istruzioni successive, comprese fra le parentesi graffe. Dopo avere valutato la condizione ed eventualmente eseguito le due istruzioni previste, lo script proseguirà con ciò che sta fuori dalle parentesi graffe. Quindi nel nostro esempio la frase "ciao a tutti voi" viene prodotta in ogni caso. E' buona norma usare comunque le parentesi graffe per delimitare il codice condizionato, anche quando è costituito da una sola istruzione: infatti questo rende il codice più leggibile, ed inoltre potrebbe evitarci degli errori se ci dovesse capitare di voler aggiungere delle istruzioni al blocco condizionato dimenticando di aggiungere le graffe.

Dobbiamo notare un particolare, riguardo alla sintassi di questa istruzione: per la prima volta, vediamo che in questo costrutto mancano il punto e virgola. Infatti la 'if' e la condizione espressa fra parentesi non devono averli; continuiamo invece a metterli nel blocco di codice condizionato.

Suggerimento: in molte tastiere non figurano le parentesi graffe. Per ottenerle, abbiamo due possibilità: una è quella di tenere premuto il tasto 'ALT' e formare, **con la tastiera numerica**, il codice 123 (per la parentesi di apertura) o 125 (per quella di chiusura). Un altro modo è quello di usare i tasti sui quali sono raffigurate le parentesi quadre (in genere a destra della P sulla tastiera), tenendo premuto ALT GR + il tasto per le maiuscole.

Abbiamo detto poco fa che la condizione espressa fra parentesi potrebbe non avere un valore booleano. PHP però è in grado di considerare booleano qualsiasi valore, in base ad una regola molto semplice. Facciamo un altro esempio banale:

```
if (5) {  
    print "ciao Luca!";  
}
```

Questo codice, da un punto di vista logico, non ha nessun senso, ma ci permette di capire come PHP interpreta le nostre espressioni. Infatti in questo caso la stringa "ciao Luca!" verrà sempre stampata.

Questo perché, per PHP, il valore 5, così come qualsiasi numero diverso da 0, è considerato 'vero'. In sostanza, PHP considera come falsi:

- il valore numerico 0, nonché una stringa che contiene '0'
- una stringa vuota
- un array con zero elementi
- un valore NULL, cioè una variabile che non è stata definita o che è stata eliminata con unset(), oppure a cui è stato assegnato il valore NULL stesso

Qualsiasi altro valore, per PHP è un valore vero. Quindi qualsiasi numero, intero o decimale purché diverso da 0, qualsiasi stringa non vuota, se usati come espressione condizionale saranno considerati veri. Quindi anche la banale espressione che abbiamo utilizzato nell'esempio di prima.

Analizziamo ora un caso che può essere fonte di grossi mal di testa: vediamo questo codice

```
$a = 7;  
if ($a = 4)  
    print '$a è uguale a 4!';
```

A prima vista, qualcuno potrebbe essere ingannato da queste istruzioni, soprattutto chi ha precedenti esperienze di programmazione in linguaggi strutturati diversamente, nei quali un'espressione come "\$a = 4" potrebbe essere sia un'assegnazione che un test. Costoro infatti si aspetteranno che questo blocco di codice non stampi niente, e rimarrebbero molto sorpresi, qualora lo provassero, di vedere invece sul browser la frase '\$a è uguale a 4!'. Questo succederebbe perché l'espressione che abbiamo messo fra parentesi è un'assegnazione: cioè essa **assegna** alla variabile \$a il valore 4. L'istruzione seguente viene quindi eseguita, per il motivo che abbiamo visto prima: il valore della nostra espressione è 4, che per PHP è un valore vero. Se invece che "if (\$a = 4)" avessimo scritto "if (\$a = 0)" l'istruzione seguente sarebbe stata saltata, perché la nostra condizione avrebbe preso valore 0, cioè falso.

Ricordiamoci quindi, quando vogliamo verificare se il valore di una variabile è **uguale** a qualcosa, di utilizzare l'operatore condizionale di uguaglianza, cioè il doppio uguale ("=="). In caso contrario, non solo otterremo molto spesso risultati diversi da quelli che ci saremmo aspettati, ma avremo anche **modificato** il valore di una variabile che volevamo soltanto verificare!

Andiamo ora un po' più a fondo nell'analisi dell'istruzione 'if': essa infatti ci permette non solo di indicare quali istruzioni vogliamo eseguire se la condizione è vera, ma anche di esprimere un blocco di codice da eseguire quando la condizione è falsa. Ecco come:

```
if ($nome == 'Luca') {  
    print "bentornato Luca!";  
} else {  
    print "ciao $nome!";  
}
```

La parola chiave 'else', che significa 'altrimenti', deve essere posizionata subito dopo la parentesi graffa di chiusura del codice previsto per il caso 'vero' (o dopo l'unica istruzione prevista, se non abbiamo usato le graffe). Anche per 'else' valgono le stesse regole: niente punto e virgola, parentesi graffe obbligatorie se dobbiamo esprimere più di un'istruzione, altrimenti facoltative. Nel nostro caso di esempio, se la variabile \$nome ha il valore 'Luca' stamperemo la frase 'bentornato Luca!', altrimenti verrà stampato 'ciao ' seguito dal nome del nostro visitatore. Ovviamente il blocco di codice specificato per 'else' viene ignorato quando la condizione è vera, mentre viene eseguito se la condizione è falsa.

Le istruzioni 'if' possono essere nidificate una dentro l'altra, consentendoci così di creare codice di una certa complessità. Esempio:

```

if ($nome == 'Luca') {
    if ($cognome == 'Rossi') {
        print "Luca Rossi è di nuovo fra noi";
    } else {
        print "Abbiamo un nuovo Luca!";
    }
} else {
    print "ciao $nome!";
}

```

In questo caso, abbiamo nidificato un ulteriore test all'interno del primo caso, quello in cui \$nome ha il valore 'Luca'. Abbiamo infatti previsto un messaggio diverso, a seconda del valore della variabile \$cognome.

Un'ulteriore possibilità che ci fornisce l'istruzione 'if' in PHP è quella di utilizzare la parola chiave 'elseif'. Attraverso questa possiamo indicare una seconda condizione, da valutare solo nel caso in cui la prima sia falsa. Indicheremo quindi, di seguito, il codice da eseguire nel caso in cui questa condizione sia vera, ed eventualmente, con 'else', il codice previsto per il caso in cui anche la seconda condizione è falsa.

```

if ($nome == 'Luca') {
    print "bentornato Luca!";
} elseif ($cognome == 'Verdi') {
    print "Buongiorno, signor Verdi";
} else {
    print "ciao $nome!";
}

```

In questo caso, abbiamo un'istruzione da eseguire quando \$nome vale 'Luca'; nel caso in cui ciò non sia vero, è prevista una seconda istruzione se \$cognome è 'Verdi'; se nemmeno questo è vero, allora verrà eseguita la terza istruzione. Da notare che, se \$nome è 'Luca' e \$cognome è 'Verdi', viene comunque eseguita solo la prima istruzione, perché dopo avere verificato la condizione, tutti gli altri casi vengono saltati.

Passiamo ora a verificare una seconda istruzione che ci permette di prevedere diversi valori possibili per un'espressione:

```

switch ($nome) {
    case 'Luca':
        print "E' tornato Luca!";
        break;
    case 'Mario':
        print "Ciao, Mario!";
        break;
    case 'Paolo':
        print "Finalmente, Paolo!";
        break;
    default:
        print "Benvenuto, chiunque tu sia";
}

```

L'istruzione 'switch' prevede che indichiamo, fra parentesi, un'espressione che verrà valutata per il suo valore (questa volta non si tratta necessariamente di un valore booleano). Di seguito, tra parentesi graffe, esprimeremo una serie di espressioni da confrontare con quella indicata prima: dal momento in cui ne trova una il cui valore è uguale, PHP esegue il codice indicato di seguito, **fino a quando non incontra un'istruzione 'break'**. Come possiamo vedere dall'esempio, le espressioni da confrontare con la prima vengono precedute dalla parola chiave 'case' e seguite dai due punti.

L'istruzione 'default' può essere indicata come 'ultimo caso', che si considera verificato quando nessuno dei casi precedenti è risultato vero. L'indicazione 'default' può anche essere assente, ma quando c'è deve essere l'ultima della switch.

Nel nostro esempio abbiamo indicato una frase di saluto per ciascuno dei valori previsti per la variabile \$nome, più una frase generica per il caso in cui il valore di \$nome non sia nessuno di quelli indicati.

E' molto importante comprendere la funzione dell'istruzione break in questa situazione: infatti, quando PHP verifica uno dei casi, esegue non solo il codice che trova subito dopo, ma anche tutto quello che trova di seguito, compreso quello relativo ai casi seguenti. Questo fino a quando non trova, appunto, un'istruzione break. Nel nostro esempio, se non avessimo messo i break e il valore di \$nome fosse stato 'Luca', avremmo ottenuto in output tutti e quattro i messaggi. Questo comportamento, apparentemente bizzarro, ci permette però di prevedere un unico comportamento per più valori dell'espressione sotto esame:

```
switch ($nome) {
  case 'Luca':
  case 'Giorgio':
  case 'Franco':
    print "Ciao, vecchio amico!";
    break;
  case 'Mario':
    print "Ciao, Mario!";
    break;
  case 'Paolo':
    print "Finalmente, Paolo!";
    break;
  default:
    print "Benvenuto, chiunque tu sia";
}
```

In questo caso, abbiamo previsto un unico messaggio per il caso in cui la variabile \$nome valga 'Luca', 'Giorgio' o 'Franco'.

L'operatore ternario. L'operatore ternario è così chiamato perché è formato da tre espressioni: il valore restituito è quello della seconda o della terza di queste espressioni, a seconda che la prima sia vera o falsa. In pratica, si può considerare, in certi casi, una maniera molto sintetica di effettuare una 'if'.

```
($altezza >= 180) ? 'alto' : 'normale' ;
```

Questa espressione prenderà il valore 'alto' se la variabile \$altezza è maggiore o uguale a 180, e 'normale' nel caso opposto. Come vediamo, l'espressione condizionale è contenuta fra parentesi e seguita da un punto interrogativo, mentre due punti separano la seconda espressione dalla terza. Questo costrutto può essere utilizzato, ad esempio, per valorizzare velocemente una variabile senza ricorrere all'if:

```
$tipologia = ($altezza >= 180) ? 'alto' : 'normale';
```

equivale a scrivere

```
if ($altezza >= 180)
  $tipologia = 'alto' ;
else
  $tipologia = 'normale';
```

Come potete vedere, in termini di spazio il risparmio è abbastanza significativo. Bisogna però stare attenti ad abusare di questa forma, perché a volte può rendere più difficoltosa la leggibilità del nostro script.

Iterazioni

I cicli sono un altro degli elementi fondamentali di qualsiasi linguaggio di programmazione, in quanto ci permettono di eseguire determinate operazioni in maniera ripetitiva. E' una necessità che si presenta molto spesso: infatti non è raro che un programma o uno script debbano elaborare quantità anche molto grosse di dati; in questa situazione, si prevederà il trattamento da utilizzare per ogni singolo dato (o gruppo di dati correlati), che sarà poi ripetuto per tutte le ricorrenze di tali dati.

Iniziamo subito con un esempio (come al solito abbastanza banale): supponiamo di voler mostrare i multipli da 1 a 10 di un numero, ad esempio 5. La prima soluzione è quella di usare il ciclo **for**:

```
for ($mul = 1; $mul <= 10; $mul++) {  
    $ris = 5 * $mul;  
    print("5 * $mul = $ris<br>");  
}
```

Questo costrutto, simile a quello usato in altri linguaggi, utilizza la parola chiave 'for', seguita, fra parentesi, dalle istruzioni per definire il ciclo; di seguito, si racchiudono fra parentesi graffe tutte le istruzioni che devono essere eseguite ripetutamente. Le tre istruzioni inserite fra le parentesi tonde vengono trattate in questo modo: **la prima** viene eseguita una sola volta, all'inizio del ciclo; **la terza** viene eseguita alla fine di **ogni** iterazione del ciclo; **la seconda** deve essere una condizione, e viene valutata **prima** di ogni iterazione del ciclo: quando risulta falsa, l'esecuzione del ciclo viene interrotta, ed il controllo passa alle istruzioni dopo le parentesi graffe. Quando invece è vera, le istruzioni fra parentesi graffe vengono eseguite. Ovviamente è possibile che tale condizione risulti falsa fin dal **primo** test: in questo caso, le istruzioni contenute fra le parentesi graffe non saranno eseguite nemmeno una volta.

Il formato standard di una if è quindi quello che abbiamo visto sopra, che utilizza le parentesi tonde per definire un 'contatore': con la prima istruzione lo si inizializza, con la seconda lo si confronta con un valore limite oltre il quale il ciclo deve terminare, con la terza lo si incrementa dopo ogni esecuzione.

Con il ciclo 'for', così come con tutti gli altri cicli, è molto importante stare attenti a non creare una situazione in cui il ciclo non raggiunge mai una via d'uscita (il cosiddetto 'loop'): in questo caso, infatti, lo script rieseguirebbe il nostro ciclo all'infinito. Nel nostro esempio di prima potrebbe succedere, ad esempio, se sbagliassimo a scrivere il nome della variabile \$mul nell'istruzione di incremento: se avessimo scritto '\$mus++', avremmo incrementato questa variabile, che non viene utilizzata nel ciclo, dopo ciascuna delle sue esecuzioni; in questo modo \$mul rimarrebbe sempre uguale ad 1, ed il nostro ciclo stamperebbe all'infinito "5 * 1 = 5", perché \$mul non diventerebbe mai maggiore di 10.

In molte situazioni 'classiche' di programmazione, un errore di questo genere potrebbe costringerci a forzare la chiusura del programma o addirittura a spegnere la macchina: nel caso di PHP, questo di solito non succede, in quanto gli script PHP hanno un limite di tempo per la loro esecuzione, oltre il quale si arrestano. Tale limite è normalmente di 30 secondi, ed è comunque impostabile attraverso il file php.ini (v. lez. 14).

Vediamo ora un altro tipo di ciclo, più semplice nella sua costruzione: il ciclo 'while'. Questo si può considerare come una specie di 'if' ripetuta più volte: infatti la sua sintassi prevede che alla parola chiave 'while' segua, fra parentesi, la condizione da valutare, e fra parentesi graffe, il codice da rieseguire fino a quando tale condizione rimane vera. Vediamo con un esempio come ottenere lo stesso risultato dell'esempio precedente:

```
$mul = 1;
while ($mul <= 10) {
    $ris = 5 * $mul;
    print("5 * $mul = $ris<br>");
    $mul++;
}
```

Il ciclo 'while', rispetto al 'for', non ci mette a disposizione le istruzioni per inizializzare e per incrementare il contatore: quindi dobbiamo inserire queste istruzioni nel flusso generale del codice, per cui mettiamo l'inizializzazione prima del ciclo, e l'incremento all'interno del ciclo stesso, in fondo. Anche in questa situazione, comunque, il concetto fondamentale è che l'esecuzione del ciclo termina quando la condizione fra parentesi non è più verificata: ancora una volta, quindi, è possibile che il ciclo non sia eseguito mai, nel caso in cui la condizione risulti falsa fin da subito.

Esiste invece un'altra forma, simile al 'while', con la quale possiamo assicurarci che il codice indicato tra le parentesi graffe venga eseguito almeno una volta: si tratta del 'do...while'

```
$mul = 11;
do {
    $ris = 5 * $mul;
    print("5 * $mul = $ris<br>");
    $mul++;
} while ($mul <= 10)
```

Con questa sintassi, il 'while' viene spostato **dopo** il codice da ripetere, ad indicare che la valutazione della condizione viene eseguita solo dopo l'esecuzione del codice fra parentesi graffe. Nel caso del nostro esempio, abbiamo inizializzato la variabile \$mul col valore 11, per creare una situazione nella quale, con i cicli visti prima, non avremmo ottenuto alcun output, mentre con l'uso del 'do...while' il codice viene eseguito una volta nonostante la condizione indicata fra parentesi sia falsa fin dall'inizio. L'esempio stamperà quindi "5 * 11 = 55".

Uscire da un ciclo. Abbiamo visto che PHP termina l'esecuzione di un ciclo quando la condizione a cui è sottoposto non è più verificata. Abbiamo però a disposizione altri strumenti per modificare il comportamento del nostro script dall'interno del ciclo: infatti possiamo dire a PHP di non completare la presente iterazione e passare alla successiva (con 'continue') oppure di interrompere definitivamente l'esecuzione del ciclo (con 'break'). Vediamo un esempio:

```
for ($ind = 1; $ind < 500; $ind++) {
    if ($ind % 100 == 0) {
        break;
    }elseif ($ind % 25 == 0) {
        continue;
    }
    print("valore: $ind<br>");
}
```

Questo codice imposta un ciclo per essere eseguito 500 volte, con valori di \$ind che vanno da 1 a 500. In realtà, le istruzioni che abbiamo posto al suo interno fanno sì che la stampa del valore di \$ind non venga eseguita ogni volta che \$ind corrisponde ad un multiplo di 25 (infatti l'istruzione 'continue' fa sì che PHP salti la 'print' e passi direttamente all'iterazione successiva, incrementando la variabile), e che il ciclo si interrompa del tutto quando \$ind raggiunge il valore 100.

Esiste un ultimo tipo di ciclo, ed è un ciclo particolare perché è costruito appositamente per il trattamento degli array: si tratta del 'foreach'. Questo ci permette di costruire un ciclo che viene ripetuto per ogni elemento dell'array che gli passiamo. La sintassi è la seguente:

```
foreach ($arr as $chiave => $valore) {  
    ....istruzioni da ripetere....  
}
```

All'interno delle parentesi graffe avremo a disposizione, nelle due variabili che abbiamo definito \$chiave e \$valore, l'indice e il contenuto dell'elemento che stiamo trattando. E' da notare che, nel caso ci interessi soltanto il valore e non la chiave (ad esempio perché le chiavi sono numeriche), possiamo anche evitare di estrarre la chiave stessa:

```
foreach ($arr as $valore) {  
    ....istruzioni da ripetere....  
}
```

Le funzioni

Vediamo ora come **definire** una funzione.

Immaginiamo, facendo il solito esempio banale, di voler costruire una funzione che, dati tre numeri, ci restituisca il maggiore dei tre. Vediamo il codice relativo:

```
function il_maggiore($num1, $num2, $num3) {
  if (! is_numeric($num1) {
    return false;
  }
  if (! is_numeric($num2) {
    return false;
  }
  if (! is_numeric($num3) {
    return false;
  }
  if ($num1 > $num2) {
    if ($num1 > $num3) {
      return $num1;
    } else {
      return $num3;
    }
  } else {
    if ($num2 > $num3) {
      return $num2;
    } else {
      return $num3;
    }
  }
}
```

Come vediamo, la definizione della funzione avviene attraverso la parola chiave **function**, seguita dal nome che abbiamo individuato per la funzione, e dalle parentesi che contengono i parametri che devono essere passati alla funzione. Di seguito, contenuto fra parentesi graffe, c'è codice che viene eseguito ogni volta che la funzione viene richiamata.

All'interno della funzione noterete l'istruzione **return...**; questa istruzione segna il termine della funzione, cioè restituisce il controllo allo script nel punto in cui la funzione è stata chiamata, e contemporaneamente determina anche il **valore restituito** dalla funzione.

Quando sarà il momento di eseguire questa funzione potremo usare questo codice:

```
$a = 9;
$b = 8;
$c = 15;
$m = il_maggiore($a, $b, $c); // $m diventa 15
```

Avete visto che abbiamo chiamato la funzione usando dei nomi di variabile **diversi** da quelli usati nella funzione stessa: la funzione infatti usa \$num1, \$num2, \$num3 (parametri attuali), mentre lo script che la richiama utilizza \$a, \$b, \$c (parametri formali). Ciò che determina la corrispondenza fra gli argomenti è semplicemente **la posizione** in cui vengono indicati: nel nostro caso, quindi, \$a diventerà \$num1 all'interno della funzione, \$b diventerà \$num2 e \$c diventerà \$num3.

Avremmo potuto richiamare la nostra funzione anche passando direttamente i valori interessati, senza utilizzare le variabili:

```
$m = il_maggiore(9, 8, 15); // $m diventa 15
$m = il_maggiore(9, 8, 'ciao'); /* $m diventa FALSE, perché il terzo argomento non è numerico e
quindi i controlli all'inizio della funzione bloccano l'esecuzione */
```

Passiamo ora ad un altro argomento molto importante, che è l'**ambito** (o scope) delle variabili. Infatti, le variabili utilizzate in una funzione **esistono** solo **all'interno** della funzione stessa, mentre non sono definite né per le altre funzioni, né nello script principale. Allo stesso modo, le variabili usate dallo script principale **non** vengono viste dalle funzioni. Questa è una caratteristica molto importante del linguaggio, perché ci permette di definire le funzioni con la massima libertà nel dare i nomi alle variabili al loro interno, senza doverci preoccupare che nel nostro script (o in qualche altra funzione) ci siano variabili con lo stesso nome il cui valore potrebbe risultare alterato. Questo significa, per tornare all'esempio precedente, che se avessimo scritto **print \$num2** all'esterno della funzione `il_maggiore()`, non avremmo ottenuto alcun risultato, e anzi avremmo ricevuto un errore di tipo notice, in quanto la variabile `$num2`, in quell'ambito, non è definita.

Le variabili utilizzate all'interno di una funzione si chiamano variabili **locali**. Le variabili utilizzate dallo script principale, invece, sono dette variabili **globali**.

Normalmente, quindi, se una funzione ha bisogno di un certo dato, è sufficiente includerlo tra i parametri che le dovranno essere passati. Tuttavia, esiste una possibilità per consentire ad una funzione di vedere una variabile globale: si tratta di dichiararla attraverso l'istruzione *global*.

```
function stampa($var1, $var2) {
    global $a;
    print $a;
}
$a = 'ciao a tutti';
$b = 'buongiorno';
$c = 'arrivederci';
stampa($b, $c);
```

Questo codice, dopo avere definito la funzione `stampa()`, assegna un valore a tre variabili globali: `$a`, `$b` e `$c`. Viene poi chiamata la funzione `stampa()`, passandole i valori di `$b` e `$c`, che nella funzione si chiamano `$var1` e `$var2` ma che al suo interno non vengono utilizzati. Viene invece dichiarata la variabile globale `$a`, che è valorizzata con la stringa 'ciao a tutti', e quindi questo è ciò che viene stampato a video dall'istruzione `print`. Se non ci fosse l'istruzione "global `$a`", la variabile `$a` risulterebbe non definita all'interno della funzione, e quindi la `print` genererebbe un errore.

Noi comunque **sconsigliamo** di utilizzare le variabili globali in questo modo, perché fanno perdere chiarezza allo script e alla funzione stessa.

Termine della funzione e valore restituito. Abbiamo visto in precedenza che la funzione termina con l'istruzione `return`, la quale può anche essere utilizzata per restituire un valore. Nel caso in cui non sia scopo della funzione quello di restituire un valore, utilizzeremo semplicemente **return**;. Viceversa, nel caso in cui volessimo restituire **più di un** valore, siccome la sintassi di PHP ci consente di restituire una sola variabile, potremo utilizzare un array. Vediamo un esempio, immaginando una funzione il cui scopo sia quello di ricevere un numero e di restituire il suo doppio, il suo triplo e il suo quintuplo:

```
function multipli($num) {
    $doppio = $num * 2;
    $triplo = $num * 3;
    $quintuplo = $num * 5;
    $ris = array($doppio, $triplo, $quintuplo);
    return $ris;
}
```

Con questo sistema siamo riusciti a restituire tre valori da una funzione, pur rispettando la sintassi che ne prevede uno solo. Ovviamente quando richiameremo la funzione dovremo sapere che il risultato riceverà un array:

```
$a = 7;  
$mul = multipli($a); // $mul sarà un array a 3 elementi
```

Se ci interessa, possiamo anche usare un costrutto del linguaggio per distribuire immediatamente i tre valori su tre variabili distinte:

```
list($doppio, $striplo, $quintuplo) = multipli($a);
```

Il costrutto **list()** serve ad assegnare i valori di un array (quello indicato a destra dell'uguale) ad una serie di variabili che gli passiamo fra parentesi. Ovviamente è possibile utilizzarlo non solo per "raccogliere" il risultato di una funzione, ma con qualsiasi array:

```
$arr = array('Marco','Paolo','Luca');  
list($primo, $secondo, $terzo) = $arr;
```

In questo caso, \$primo prenderà il valore 'Marco', \$secondo il valore 'Paolo', \$terzo il valore 'Luca'.

Un'ulteriore precisazione: poco fa abbiamo detto che la funzione termina con l'istruzione return. In realtà ciò non è necessario: infatti, nel caso in cui **non** ci siano valori da restituire, possiamo anche omettere l'istruzione return, e l'esecuzione della funzione terminerà quando arriverà in fondo al codice relativo, restituendo il controllo allo script principale (o ad un'altra funzione che eventualmente l'ha chiamata).

Parametri passati per riferimento. Normalmente, si dice che i parametri ad una funzione vengono passati **per valore**. Questo significa che alla funzione viene in realtà passata non la variabile indicata, ma **una sua copia**, per cui la variabile originale non subirà alcun mutamento, indipendentemente da ciò che succede all'interno della funzione.

Passando il parametro **per riferimento** invece che per valore la dichiarazione della funzione, sarà del tipo:

```
function auguri($nome, &$eta) {
```

Quel **&**, aggiunto davanti al nome della variabile \$eta, fa sì che questa variabile venga passata per riferimento (in pratica, significa che la funzione utilizzerà la variabile vera e propria, e non una sua copia). Quindi, eseguendo il della funzione, il parametro attuale passato (che dentro la funzione si chiama \$eta) dopo la chiamata cambierà il suo valore anche in ambito globale.

Un'ultima annotazione: è possibile passare una variabile per riferimento anche quando ciò non è previsto nella definizione della funzione. Si può infatti utilizzare il simbolo **&** anche nel momento in cui si **chiama** la funzione ma per avere questa possibilità è necessario modificare un valore di php.ini (allow_call_time_pass_reference). Si tratta comunque di una prassi **sconsigliata**: se si ha necessità di passare una variabile per riferimento, è bene tenerlo presente nel momento in cui si **progetta** la funzione.

Le sessioni

La gestione delle sessioni è una delle novità più importanti introdotte con la versione 4 di PHP. Essa infatti ci permette di stabilire un "dialogo" con l'utente del sito web, superando uno dei limiti del protocollo HTTP, che è quello di "non avere stato". Infatti, per HTTP ogni richiesta è "unica", e una volta soddisfatta non è più in grado di "ricordarla". Quindi, quando lo stesso utente presenterà una nuova richiesta, il server web non sarà in grado di rendersi conto che si tratta dello stesso utente della richiesta precedente, e la tratterà quindi come tutte le altre.

Ovviamente, non sempre questo è un problema: la normale navigazione sui siti internet può tranquillamente avvenire in questo modo. Ma ci sono, invece, delle situazioni in cui è importante stabilire questa specie di dialogo tra l'utente ed il server, in quanto ogni operazione che l'utente compie può influenzare le successive ed essere influenzata dalle precedenti.

Un esempio è quello dei siti con aree riservate: se un certo numero di pagine è visibile solo agli utenti registrati, questi dovranno presentarsi per poterle vedere, ma una volta effettuata la presentazione (login) dovranno essere messi in grado di vedere **tutte** le pagine dell'area riservata, senza, ovviamente, doversi ripresentare per ciascuna pagina. Questo sarà possibile solo se il server "ricorda" che quell'utente si è già presentato.

La **prima cosa** da fare quando vogliamo lavorare con le sessioni è di dire a PHP dove deve salvare i files relativi. Le sessioni infatti non sono altro che files, in ognuno dei quali vengono memorizzati i dati di un utente. Quindi dobbiamo impostare, in **php.ini**, la direttiva `session.save_path`, indicando la cartella dentro la quale verranno salvati questi files. Di default, questa direttiva è impostata a `"/tmp"`: se vogliamo lasciarla così, dobbiamo creare nella directory principale del nostro disco fisso una cartella di nome "tmp". Altrimenti indicheremo il percorso completo, ad esempio `"C:\php\sessioni"`. L'importante è che questa cartella esista, altrimenti PHP non riuscirà a creare i files di sessione.

Quando creiamo una sessione, PHP le assegna un nome, generando una sequenza casuale di lettere e numeri. Dopodiché crea un file con quel nome nella cartella che abbiamo visto sopra, e quello sarà il file di sessione del nostro utente. Infine, per essere in grado di riconoscere l'utente, spedisce un cookie al suo browser, nel quale viene indicato il nome di questo file. In questo modo, quando l'utente farà una nuova chiamata al sito, il suo browser rispedirà il cookie con il nome della sessione, e PHP saprà in quale file recuperare i dati dell'utente. Tutto questo avviene solo fino a quando l'utente non chiude il browser: infatti a quel punto il cookie non è più valido, e la volta successiva l'utente dovrà ripresentarsi per essere riconosciuto.

Vediamo quindi ora come ottenere tutto questo: la funzione che ci permette di aprire una sessione è **`session_start()`**, che non prevede parametri. C'è da dire una cosa molto importante su questa funzione: essa si deve trovare in un punto del nostro script nel quale nessun output è ancora stato inviato al browser. Infatti, come abbiamo detto prima, quando la sessione deve essere creata PHP invia un cookie al browser. Il cookie è un header, cioè un tipo di dato che deve essere trasmesso **prima** del contenuto HTML della pagina. Quindi nessun contenuto HTML deve essere stato creato nel momento in cui apriamo la sessione. Questo significa che nella parte precedente del file (rispetto alla chiamata di `session_start()`) non deve esserci nulla all'esterno dei tag che delimitano il codice PHP, e all'interno del codice non deve essere stata eseguita alcuna `print()` o `echo()`, che avrebbero creato HTML. Se non rispettiamo questa regola, nel momento in cui PHP crea la sessione non riuscirà a spedire il cookie, e riceveremo un messaggio di errore che ci avvisa di questo. Si tratta di un errore di tipo warning, per cui lo script continuerà, ma non essendo stato inviato il cookie, la successiva chiamata del browser non sarà riconosciuta come appartenente alla stessa sessione.

Quando viene aperta la sessione, PHP crea un array superglobale di nome `$_SESSION`. Tutti i dati che mettiamo dentro questo array vengono salvati sul file di sessione. Quindi, supponiamo di voler salvare il nome e il cognome dell'utente che si è collegato:

```
session_start();
$_SESSION['nome'] = 'Giovanni';
$_SESSION['cognome'] = 'Bianchi';
```

I dati 'nome' e 'cognome' saranno quindi salvati sul file di sessione, insieme ai valori relativi. In una chiamata successiva eseguita dallo stesso utente, dopo avere effettuato la `session_start()` avremo di nuovo a disposizione, nell'array `$_SESSION`, gli stessi dati.

Vediamo quindi quali sono le due possibilità che si presentano quando eseguiamo una `session_start()`:

- Se il browser non ha spedito alcun cookie, la sessione deve essere creata, per cui PHP dà il nome alla sessione, crea il file e spedisce il cookie
- Se invece il browser ha spedito il cookie col nome della sessione, significa che la sessione è già aperta, e quindi PHP usa il nome della sessione per leggere il file relativo e rimetterci a disposizione, nell'array `$_SESSION`, i dati che avevamo salvato

Per eliminare le variabili di sessione, ci comporteremo come per qualsiasi altra variabile:

```
unset($_SESSION['nome']); //elimina la variabile di sessione 'nome'
$_SESSION = array(); // elimina tutte le variabili di sessione
```

Esiste poi un'altra funzione, **`session_destroy()`**, che elimina il file di sessione. Bisogna però stare attenti, perché, anche se il file viene eliminato, il contenuto dell'array `$_SESSION` rimane disponibile per lo script in corso. Quindi, se si vuole che il resto dello script tenga conto che la sessione non è più attiva, è necessario ricordarsi di svuotare l'array nel modo che abbiamo visto prima.

In chiusura, dobbiamo ricordare che in questa lezione abbiamo esposto il funzionamento delle sessioni in maniera molto semplificata: infatti, sono molti gli elementi che possono avere un comportamento diverso da quello che abbiamo visto. Ad esempio, esistono altre funzioni come `session_register()` e `session_unregister()` che venivano utilizzate con `register_globals` a "on" e che ora sono sconsigliate; inoltre, quando il browser dell'utente rifiuta i cookie, PHP può usare un sistema alternativo per trasmettere il nome della sessione; la durata del cookie di sessione può essere allungata oltre la chiusura del browser; i dati delle sessioni possono essere salvati su database invece che su file.

Form – POST E GET

La principale peculiarità del 'web dinamico' è la possibilità di variare i contenuti delle pagine in base alle richieste degli utenti. Questa possibilità si materializza attraverso i meccanismi che permettono agli utenti, oltre che di richiedere una pagina ad un web server, anche di specificare determinati parametri che saranno utilizzati dallo script php per determinare quali contenuti la pagina dovrà mostrare.

In alcuni casi, i dati che devono essere trasmessi allo script sono piuttosto numerosi: pensiamo ad esempio ad un modulo di registrazione per utenti, nel quale vengono indicati nome, cognome, indirizzo, telefono, casella e-mail ed altri dati personali. In questo caso lo script, dopo averli ricevuti, andrà a **salvarli** in un file o in un database.

In questa lezione non ci occuperemo di come vengono salvati o recuperati i dati, ma del modo in cui PHP li riceve dall'utente. Esistono due sistemi per passare dati ad uno script: il metodo GET e il metodo POST.

Il **metodo GET** consiste nell'accodare i dati all'indirizzo della pagina richiesta, facendo seguire il nome della pagina da un punto interrogativo e dalle coppie nome/valore dei dati che ci interessano. Nome e valore sono separati da un segno di uguale. Le diverse coppie nome/valore sono separate dal segno '&'. Quindi, immaginando di avere la pagina prodotto.php che mostra le caratteristiche di un prodotto passandole il codice e la categoria del prodotto stesso, diciamo che, per visualizzare i dati del prodotto A7 della categoria 2, dovremo richiamare la pagina in questo modo:

```
<a href="prodotto.php?cod=a7&cat=2">
```

La stringa che si trova dopo il punto interrogativo, contenente nomi e valori dei parametri, viene detta **query string**. Quando la pagina prodotto.php viene richiamata in questo modo, essa avrà a disposizione, al suo interno, le variabili \$_GET['cod'] (con valore 'a7') e \$_GET['cat'] (con valore '2'). Infatti i valori contenuti nella query string vengono memorizzati da PHP nell'array \$_GET, che è un array **superglobale** in quanto è disponibile anche all'interno delle funzioni.

Quindi, per tornare all'esempio del catalogo, possiamo immaginare di avere una pagina nella quale mostriamo una tabella con il nome di ogni prodotto su una riga, e, di fianco, il link che ci permette di visualizzare le caratteristiche di quel prodotto. In ogni riga, quindi, questo link richiamerà sempre la pagina prodotto.php, valorizzando ogni volta i diversi valori di 'cod' e 'cat'.

Il **metodo POST** viene utilizzato con i moduli: quando una pagina HTML contiene un tag <form>, uno dei suoi attributi è "method", che può valere GET o POST. Se il metodo è GET, i dati vengono passati nella query string, come abbiamo visto prima. Se il metodo è POST, i dati vengono invece inviati in maniera da non essere direttamente visibili per l'utente, attraverso la richiesta HTTP che il browser invia al server.

I dati che vengono passati attraverso il metodo POST sono memorizzati nell'array \$_POST. Anche questo array, come \$_GET, è un array **superglobale**. Quindi, per fare un esempio attraverso un piccolo modulo:

```
<form action="elabora.php" method="post">
<input type="text" name="nome">
<input type="checkbox" name="nuovo" value="si">
<input type="submit" name="submit" value="invia">
</form>
```

Questo modulo contiene semplicemente una casella di testo che si chiama 'nome' e una checkbox che si chiama 'nuovo', il cui valore è definito come 'si'. Poi c'è il tasto che invia i dati, attraverso il metodo POST, alla pagina elabora.php.

Questa pagina si troverà a disposizione la variabile `$_POST['nome']`, contenente il valore che l'utente ha digitato nel campo di testo; inoltre, **se è stata selezionata la checkbox**, riceverà la variabile `$_POST['nuovo']` con valore 'si'. Attenzione però: se la checkbox non viene selezionata dall'utente, la variabile corrispondente risulterà non definita.

Abbiamo visto quindi, brevemente, in che modo recuperare i dati che gli utenti ci possono trasmettere. C'è da dire che, modificando l'impostazione **register_globals** su `php.ini`, sarebbe possibile anche recuperare i dati in maniera più semplice. Infatti, se `register_globals` è on, oltre agli array visti sopra avremo anche delle variabili globali che contengono direttamente i valori corrispondenti. Ad esempio, nel primo esempio avremmo a disposizione la variabile `$cod` e la variabile `$cat`, nel secondo avremmo la variabile `$nome` e la variabile (eventuale) `$nuovo`. Fino a qualche tempo fa, erano in molti a lavorare in questo modo, perché il valore di `register_globals`, di default, era ad on, e quindi buona parte dei programmatori PHP, soprattutto agli inizi, trovavano più naturale utilizzare il sistema più immediato. A partire dalla versione **4.2.0** di PHP, però, il valore di default di `register_globals` è stato cambiato in off, e gli sviluppatori di PHP **sconsigliano** di rimmetterlo ad on. Questo perché, utilizzando gli array superglobali `$_GET` e `$_POST`, si rende il codice più chiaro e anche più sicuro. Se vi dovesse capitare di utilizzare degli script già fatti e doveste notare dei malfunzionamenti, potrebbe dipendere dal fatto che tali script utilizzano le variabili globali invece degli array superglobali, ed il vostro `register_globals` è a off.

Un'ultima annotazione: gli array `$_GET` e `$_POST` sono stati introdotti nella versione 4.1.0 di PHP. In precedenza, gli stessi valori venivano memorizzati negli array corrispondenti `$HTTP_GET_VARS` e `$HTTP_POST_VARS`, che però non erano superglobali. Questi array sono disponibili anche nelle versioni attuali di PHP, ma il loro uso è sconsigliato, ed è presumibile che in futuro scompariranno.