Il Linguaggio PHP

Questa documentazione libera é stata sviluppata all'interno delle attività formative erogate da Truelite S.r.l.

Società italiana specializzata nella fornitura di servizi, consulenza e formazione esclusivamente su GNU/Linux e software libero.
Per informazioni:

Truelite Srl

Via Monferrato, 6
50142 Firenze
Tel: 055-7879597 Fax: 055-7333336
e-mail: info@truelite.it web: http://ww.truelite.it

Indice

1 - Introduzione

Il linguaggio PHP è uno dei linguaggi di scripting più comuni, probabilmente il più usato, adottati sul web per la creazione di pagine dinamiche.

E' un linguaggio molto semplice, creato per soddisfare dele esigenze molto pratiche. Questa caratteristica è anche, tuttavia, anche una sua debolezza. I puristi dei linguaggi di programmazione *guardano storto* PHP dal momento che ha la fama di essere un linguaggio un po' raffazzonato, che ha ereditato sintassi e comportamenti da più linguaggi, evitando però di portarsi dietro da questi complicazioni o strutture troppo complesse. E', insomma, un linguaggio un po' furbetto magari un po' *sporco* e, sicuramente, non il più elegante in assoluto.

Però sta di fatto che con PHP non c'è niente che, in quanto a tecnologie web, non si possa fare. Dove veniva trovato un limite tecnologico a PHP, subito ci si preoccupava di turare quella falla. Alla fine, PHP è diventato un vero e proprio arsenale con il quale poter affrontare con sicurezza qualsiasi problema relativo ad implementazione di soluzioni web based.

L'ultima versione, la 5, tenta di dare al linguaggio quella certa rigorosità che gli è sempre mancata (ancora una volta, però, senza inventare niente di nuovo ma prendendo buone idee di altri linguaggi) introducendo strutture di controllo e gestione degli oggetti avanzati (e *veri)*.

2 - Un po' di storia

“PHP” è (o almeno lo era inizialmente) l'acronimo di “Personal Home Page” e fu creato da una persona per esigenze sue personali, per la gestione del proprio sito web.

Nel 1995 Rasmus Lerdorf pubblicizzò su un newsgroup il rilascio (sotto licenza GPL) di un set di tools per l'automazione di alcune operazioni che solitamente egli si trovava a compiere con noiose analisi dei log del web server. Si trattava semplicemente di una raccolta di alcuni CGI dal nome, appunto, di PHP Tools.

Poco tempo dopo Rasmus annuncia la nascita di un vero e proprio *parser* di pagine HTML contenenti delle stringhe particolari riconosciute ed interpretate dal

linguaggio.

> Le istruzioni del linguaggio PHP sono “affogate” all'interno di normali pagine HTML. Non appena il parser del linguaggio trova un speciale segnale di “inizio codice PHP” esso entra in azione interpretando il codice che segue fino ad arrivare al segnale di “fine codice PHP”. Tutto il resto è normale HTML, che il parser non interpreta e si limita a rendere così com'è. Per questo motivo il PHP, e tutti i linguaggi che hanno questa caratteristica si possono chamare *embedded*.

Il parser si chiamava FI (Form Interpreter). Una delle caratteristiche principali era la possibilità di interagire con un database (mSQL). Il “linguaggio”, che inizia a chiamarsi PHP/FI, come sintassi è ancora distante da quello che diventerà in futuro.

Nel 1997 esce la versione 2.0. La sintassi è ormai quella che conosceremo presto, e il supporto ai database è allargato ad altri DBMS (postgresql e MySQL).

La prima “rivoluzione” avviene nel giugno 1998, quando esce la versione 3 del linguaggio. Il parser viene riscritto completamente, usando il un *motore* scritto da due israeliani. Inoltre, con questa versione, il php passa da essere un *one man project* ad un progetto supportato da un vero team di sviluppatori (tra i quali, per inciso, lo stesso Rasmus acquisterà via via un'importanza sempre minore).

Con la versione 3.0, il PHP diventa “PHP: Hypertext Processor”, e ne viene rilasciata anche una versione per server Windows.

Nel maggio 2000, esce la versione 4.0. Con esso viene utilizzato il *motore* Zend, ancora più efficente del motore usato nel PHP 3. Questo fatto comporta da un lato un miglioramento delle prestazioni e dall'altro la possibilità di maggiori e più facili estensioni del linguaggio. Al PHP 4 vengono aggiunte tutte quelle funzionalità che il PHP 3 adottava tramite l'uso di moduli esterni al linguaggio, in primis l'uso *nativo* delle sessioni. Viene migliorata la gestione della memoria e il modello delle *API* (applicazion program interface). Con la versione 4, il PHP cambia anche licenza: dalla GPL si passa alla più restrittiva PHP Licence.

L'ultima versione del php, la 5, è stata rilasciata il 13 luglio 2004, ben 4 anni dopo la versione 4. Con questa versione il linuaggio raggiunge una prima maturità. Il nuovo Zend Engine 2 introduce un nuovo modello di programmazione ad oggetti, è stato introdotto un nuovo modello di interazione con i file XML e il nuovo supporto ai Web Services.

3 - Installazione e configurazione

Come abbiamo già visto nelle dispense dove introducevamo i concetti alla base della pagine dinamiche, il funzionamento del PHP è basato sull'istruire il nostro web server in modo che, a fronte della ricezione di un particolare tipo di pagina (quelle con estensione php, nel nostro caso), esso giri la richiesta, e dunque l'elaborazione, al rispettivo interprete, attendendo da esso lo *stream* di dati HTML da girare al UA che aveva fatto la richiesta.

Questa operazione avviene modificando la configurazione del server Apache (httpd.conf o similari) in questo modo:

- all'avvio di Apache, caricare il *modulo* che gestisce le pagine PHP, il vero e proprio interprete del linguaggio. Questo farà sì che l'inteprete risieda *dentro* il processo principale di Apache, condividendone le risorse. In questo modo, quando di debba elaborare una pagina PHP, non verrà richiesta l'esecuzione (pesante) di un processo interno, ottimizzando l'uso della memoria e la velocità di esecuzione.

```
LoadModule php4_module    extramodules/mod_php4.so
```

- Istruire Apache che qualsiasi richiesta di pagina PHP sia girata all'interprete.

```
AddType application/x-httpd-php .php
```

Con questa cofigurazione, qualsiasi pagina con estensione ".php", sarà gestita internamente dall'*handler* del mime type " application/x-httpd-php", ovvero dal mod_php4.so.

Questo è quanto per la configurazione di Apache. Il PHP stesso, però, possiede un proprio file di configurazione, il php.ini.

All'interno di questo file sono definite moltissime variabile che possono influire il

comportamento del php. Inoltre, come apache, anche PHP ha una sorta di “moduli”, oggetti cioè che possono essere caricati solo su richiesta in maniera da rendere il processo principale più “snello”, togliendo tutto ciò di cui si sa che non faremo uso. In PHP questi moduli si chiamano “extension”.

Nel file di configurazione è inoltre possibile definire dei singoli dettagli per ogni singola *extension* caricata (è per esempio possibile definire una username ed una password da usare per la connessione ad un server mysql).

> Essendo il PHP *parte* del server web, quando si effettuino modifiche al file php.ini, occorre riavviare Apache.

Per esigenze particolari è inoltre possibile utilizzare la stesso file di configurazione di Apache per modificare dei parametri PHP (quelli che solitamente dovrebbero trovarsi all'interno del file php.ini). E` sufficiente dichiarare queste variabili all'interno del httpd.conf prefissandole con la stringa “php_”.

Molte variabili di configurazione del PHP sono modificabili anche a *run time*, cioè direttamente dall'interno degli script php. Alcune di queste, però, non possono esserlo e devono essere esplicitamente modificate all'interno del file di configurazione. Non sempre, però, si ha accesso al file di configurazione del php (quando si sia ospitati, per esempio, su un server del quale non possediamo la password di amministrazione). In tal caso, allora, queste variabili possono essere modificate tramite l'utilizzo del file “.htaccess”.

4 - Reperire informazioni online

Il sito di riferimento di PHP è http://www.php.net. La quantità di documentazione reperibile è completamente sufficiente per qualsiasi esigenza. Inoltre ogni singola funzione e elemento del linguaggio è spesso commentato dagli utenti. Si trovano così dei preziosi consigli o addirittura porzioni di codice (*snippets*) da poter utilizzare.

5 - Concetti di base

5.1 - Sintassi

Come abbiamo già accennato, il codice php è presente affogato all'interno (*embedded)* di normali pagine HTML (le quali dovranno però avere l'estensione ".php" o un altra comuqnue configurata).
Il parser, una volta ricevuta la richiesta dal web server di interpretare una pagina, andrà a cercarla sull'hard disk, ed inizierà a leggerla. Fintanto che troverà codice HTML, questo lo rispedirà direttamente al server, ma appena troverà il *marker* di inizio codice PHP, inizierà ad interpretare il codice fino a che non troverà il marker di fine codice (o la fine del documento).
Il codice PHP iniza dunque dopo i caratteri

```
<?php
```

e terminerà con i caratteri

```
?>
```

tutto quello che si trovi prima o dopo tali marker, non sarà interpretato dal PHP.

Ecco un primo esempio di una pagina HTML/PHP completa (esempio.php):

```
<html>
<head>
<title>Un esempio di PHP</title>
</head>
<body>
<h1>Esempio</h1>
<?php
   $a = 10;
   print "<strong>Benvenuto, visitatore $a !</strong>";
?>
</body>
</html>
```

L'istruzione di PHP "print", che permette di stampare una stringa è affogata (embedded) all'interno di una pagina HTML. Il parser inizierà a leggere la pagina

ma non entrerà in azione fintanto che non troverà i caratteri “<?php”.

Una volta che la pagina sia stata interpretata, al server arriverà uno stream HTML così fatto:

```
<html>
<head>
<title>Un esempio di PHP</title>
</head>
<body>
<h1>Esempio</h1>
<strong>Benvenuto, visitatore 10 !</strong>
</body>
</html>
```

Quello che appunto deve essere capito è che non è il parser a spedire indietro la risposta all'UA che ha fatto la richiesta, ma sempre e comunque il server web.

La pagina che arriverà al UA, dunque, non conterrà mai nessuna instruzione php. L'utente vedrà sempre e comunque il *risultato* dell'interpretazione dello script (solitamente, appunto, del puro codice HTML).

Ogni istruzione di php deve terminare con il carattere punto e virgola “;”.

5.2 - I commenti

All'interno di uno script PHP è possibile definire tramite dei caratteri di controlli l'inizio e la fine di un commento, ovvero del testo libero (solitamente usato appunto per commentare il codice che si sta scrivendo o altro tipo di annotazioni) che il parse non interpreterà.
I commenti possono essere definiti come segue:

```
/* Questo è un commento
   su più righe         */

// Questo è un commento su una sola riga
```

```
# Un altro modo per commenti di una sola riga
```

5.3 - Le variabili

Il concetto di *variabile*, presente nella maggior parte dei linguaggi di programmazione, è semplicemente un sistema tramite il quale è possibile riferirsi a un valore presente all'interno della memoria del computer (organizzata in un modo che in questo documento non ci interessa approfondire).

Una variabile, identificata da un nome, è dunque un riferimento al contenuto di una particolare area di memoria che il programma (in questo caso l'inteprete PHP) ha avuto a disposizione da parte del sistema operativo.

Quando effettuiamo l'assegnazione di una variabile non facciamo altro che inserire il valore dell'espressione con il quale vogliamo *valorizzare* la variabile all'interno di questa area di memoria. A tale contenuto sarà poi possibile fare accesso usando, appunto, il nome della variabile.

In PHP il nome delle variabili sono stringhe di caratteri precedute sempre dal carattere “$”.

Esempi di variabili semplici:

```
$contatore = 0;
$nome = “Mario Rossi”;
```

Il nome delle variabili è case sensitive per cui, nell'esempio “$nome”, “$Nome” e “$NOME” sarebbero tre diverse variabili.

Il nome delle variabili deve iniziare con un _ (underscore) o con una lettera. Non si può iniziare il nome di una variabile con un numero.

5.3.1 - Il nome delle variabili

Occorre aprire un capitolo a parte per discutere brevemente su una questione che, pur non facendo parte strettamente della sintassi del linguaggio, è tuttavia molto importante.
Nel corso del tempo ci troveremo, nei nostri script, a dover dare un nome alle

variabili. Ma quale nome scegliere ?
La regola fondamentale è: scegliere sempre un nome *significativo*, anche per variabili “usa e getta”, per variabili temporanee usate per pochissimo tempo all'interno dello script.
Nomi di variabile come “a”, “pp”, “x” o altre simili, rendono il codice illeggibile e poco manutebile. Non abbiate paura ad usare nomi di variabili lunghi.
Se devo dichiarare una variabile che debba contenere, ad esempio, la data di nascita dell'utente, allora “$UserBornDate”, sarà sicuramente preferibile a “$date” o addiruttura a “$d”. Quel poco di tempo che perderemo a scrivere il nome più lungo, sarà sicuramente compensato da quello che NON perderemo quando dovremmo capire, una volta che successivamente metteremo mano al codice, cosa mai indicasse quel “$d”.
Da altri linguaggi si possono per esempio ereditare delle convenzioni, a questo riguardo.
Per esempio, potrei far iniziare con un carattere particolare tutte le variabili che definisco allo scopo di ricordare il *tipo* di scalare che utilizzo:
In questo modo $**b**Test sarà un boolean, $**i**Counter sarà un contatore intero, $**s**CustomerName sarà una stringa contenente il nome del cliente, etc.
Facciamo ammenda del fatto che, nel testo, ci troveremo invece per praticità ad usare spesso nomi di variabile senza senso (parzialmente scusati dal fatto che, comunque, il nome di una variabile si intende anche essere contestualizzata, cosa che in un esempio di tre righe è ben difficile fare).
Sullo stile del nome, invece, sono generalmene due le varianti usate.
Usando la variante “*camel words*”, scriveremo il nome delle nostre variabili o delle nostre funzioni componendolo con parole attaccate l'una con l'altra con l'iniziale di ogni parola in maiuscolo, eccetto la prima.
Esempio:

```
$userBornDate = “8/4/1980”;
$userName = “mario”;
$counter = 0;
function getUserName() { ... }
```

Ereditando una convenzione da Java, le classi (che vedremo più avanti), iniziano invece sempre con la lettere maiuscola.
Un'altra variante molto usate è quella "*GNU style*", tramite la quale tutte le parole che compongono il nome di una variabile o di una funzione, sono scritte in minuscolo e separate dal carattere "_":

```
$user_born_date = "8/4/1980";
$user_name = "mario";
$counter = 0;
function get_user_name() { ... }
```

De gustibus.

5.3.2 - Nomi variabili di variabili

Tramite una particolare sintassi è possibile utilizzare una variabile PHP per registrare... il nome di un'altra variabile (e non il suo contenuto). Per poter effettuare questa operazione è sufficiente far precedere al nome della variabile che contiene il nome di un'altra variabile i due caratteri $$.

Un esempio:

```
<?php
$varName = "nomeUtente";
$$varName = "Franco";
// Stampa il contenuto di $nomeUtente
print $nomeUtente;
?>
```

5.4 - Tipi di dati semplici (scalari)

Il PHP fornisce quattro tipi di dato fondamentali:

- Numeri interi (positivi e negativi)

- Numeri in virgola mobile (positivi e negativi)
- Stringhe
- Boolean (true o false)

Esempi:

```
<?php
   // Numeri interi
   $intero1 = 18;
   $intero2 = -2994;
   $intero3 = +42;
   // Floating point
   $float1 = 0.144;
   $float2 = .57;
   $float3 = -5.44;
   // Stringhe
   $stringa1 = "Ecce homo";
   $stringa2 = "\$float3 vale $float3";
   $stringa3 = "Altro \"esempio\"";
   $stringa4 = 'Anche con "virgolette" semplici';
   // Boolean
   $bool1 = true;
   $bool2 = false;
?>
```

Nelle ultime due stringhe abbiamo anche introdotto il concetto di *escaping*, ovvero quel metodo che ci consente di usare, all'interno di una stringa, dei caratteri che normalmente sarebbero interpretati come elementi sintattici PHP. Nel caso di "stringa2", il carattere "$" sarebbe interpretato come inizio di nome di una variabile. Prefissando tale carattere dal carattere *backslash* "\", facciamo in modo che l'interprete PHP consideri il "$" letteralmente, senza intrpretarlo. Lo stesso dicasi per l'esempio di $stringa3, dove abbiamo usato, al suo interno, il carattere doppie virgolette " " ".

Nella $stringa4, invece, abbiamo fatto uso delle virgolette semplici, per definire il contenuto della variabile. In questo modo è possibile usare le doppie virgolette

senza escaping.

All'interno delle stringhe, è possibile inoltre utilizzare dei caratteri speciali (sempre tramite escaping), che permettono di inserire caratteri difficilmente inseribili in maniera letterale. Essi sono:

```
\n = Nuova riga
\r = Ritorno carrello
\t = Carattere di tabulazione
\\ = backslash
```

In PHP il tipo di variabile è implicito, ovvero non occorre definire a priori il tipo di variabile, prima di effettuare un assegnamento. Nello stesso modo vengono effettuate delle conversioni implicite di tipo quando, per esempio, si effettuassero delle assegnazioni o delle operazioni su tipi di dato diversi.

Dunque:

```
$a = 10;
$b = "103";
print ($a + $b); // Il risultato sarà 113
```

Ovviamente è sempre bene non fidarsi troppo dell'intelligenza dell'interprete quando si tratta di conversioni implicite. E` buona norma fare comunque attenzione nel caso si dovessero moltiplicare "pere con mele".

5.5 - Valore e reference

Quando si assegna un valore ad una variabile, l'intero valore viene copiato direttamente nell'area di memoria alla quale sarà possibile accedere direttamente usando, appunto, il nome della variabile. Ogni variabile ha dunque il suo *spazio* di memoria all'interno del quale si troverà il proprio valore.
Dal PHP 4 è stato però inserito un nuovo tipo di assegnazione (*by reference*), tramite il quale è possibile creare una sorta di *alias* di una variabile.
In pratica, una volta dichiarata una variabile e una volta che le sia stato assegnato un valore, è possibile, tramite un altro nome, riferirsi alla stessa zona di memoria.

In questo modo, ogni volta che si modifica l'una o l'altra variabile (quella vera e la sua reference, il suo alias) verrano modificate entrambe in quanto queste condividono la stessa zona di memoria.
Una reference viene creata prependendo il carattere “&” al nome della nuova variabile.
Segue un esempio:

```
<?php
$foo = 'Roberto';     // Assegna il valore “Roberto” a $foo
$bar = &$foo;   // Creo una reference a $foo tramite $bar.
$bar = "Il mio nome è $bar";  // Modifica $bar...
print $bar;
print $foo; // Anche $foo è stato modificato.
?>
```

E' possibile usare le reference soltanto su variabile con un nome, dunque non su espressioni numeriche o costanti (vedi prossimo paragrafo).

Il seguente esempio è dunque errato:

```
<?php
$foo = &(24 * 7);  // Errore
?>
```

5.6 - Costanti

E' possibile definire delle costanti, ovvero dei nomi mnemonici da usare per comodità in caso di particolari necessità.
Le costanti, lo dice il nome stesso, non sono variabili ovvero non è possibile modificare il loro valore una volta che questo è stato definito. Le costanti, inoltre, non devono iniziare con il carattere $.
Si definiscono tramite l'apposita funzione “define”.

```
define (“ERR_NOT_FOUND”, “File non trovato”);
print (ERR_NOT_FOUND);
```

Il nome delle costanti non è case sensitive, ma per convenzione sono solitamente scritte sempre in maiuscolo.

L'utilizzo pratico di questi costrutti sarà spiegato dettagliatamente più avanti.

5.7 - Operatori

Gli operatori rappresentano simbolicamente delle operazioni che vengono compiute su una o più espressioni, chiamati operandi dell'espressione. Talvolta si usa anche la denominazione di *left operand* e *right operand* a seconda della posizione dell'operando rispetto al simbolo dell'espressione.

5.7.1 - Le espressioni

Le espressioni sono qualsiasi cosa che possa essere *valutata* a formare un valore di un qualche tipo. E' un concetto generico della programmazione e non è direttamente collegato al PHP.
La forma più semplice di espressione è un numero:

```
10
```

Questa espressione può essere valutata al valore intero 10.

Un'espressione può essere una variabile:

```
$a
```

L'espressione verrà valutata al valore della variabile a.

Un'assegnazione stessa, può essere un'espressione:

```
$a = 10;
```

In questo caso il valore a cui viene valutata l'espressione può dipendere dal linguaggio. In PHP, per esempio, tale espressione assume il valore finale dell'assegnamento, ovvero 10. In C, invece, l'espressione di assegnamento torna sempre “1” (da intendersi come “vero”. Un'assegnazione è sempre... vera).

Negli esempi che seguono, useremo come segnaposto per gli operandi le variabili “$a” e “$b”. In realtà intenderemo qualcosa di più del semplice valore di $a e $b, in quanto, come accennato l'espressione può essere qualcosa di più del semplice valore di una variabile.

5.7.2 - Operatori aritmetici

Operatori per compiere semplice operazioni aritmetiche sulle espressioni:

Nome	Esempio	Risultato
Addizione	$a + $b	La somma di $a e $b
Sottrazione	$a - $b	La differenza di $a e $b
Divisione	$a / $b	Quoziente tra $a e $b
Moltiplicazione	$a * $b	Prodotto di $a e $b
Modulo	$a % $b	Il resto di $a diviso $b

L'operatore di divisione restituisce un float anche se i due operandi fossero interi.

5.7.3 - Operatore di assegnazione

L'operatore di assegnazione è l'uguale: “=”.

Lo abbiamo ormai già incontrato diverse volte. Serve per assegnare un valore ad una variabile (non ad una espressione).

Il valore da assegnare può essere ovviamente un'espressione:

```
<?php
$a = 10;
$b = 34;
$c = ($b +1) - $c;
print ($a); // Stamperà 25
?>
```

5.7.4 - Operatori di confronto

Gli operatori di confronto, come suggerisce il loro nome, permettono di confrontare due valori.

Tabella riassuntiva:

Nome	Esempio	Risultato
Uguale	$a == $b	Vero se $a e $b hanno valore uguale
Identico	$a === $b	Vero se $a e $b sono uguali e sono dello stesso tipo
Diverso	$a != $b	Vero se $a e $b hanno valori diversi

Diverso	$a <> $b	Vero se $a e $b hanno valori diversi
Non identici	$a !== $b	Vero se $a e $b hanno valori diversi o se sono di tipo diverso
Minore	$a < $b	Vero se il valore di $a è minore di quello di $b
Maggiore	$a > $b	Vero se il valore di $a è maggiore di quello di $b
Minore o uguale	$a <= $b	Vero se il valore di $a è minore o uguale a quello di $b
Maggiore o uguale	$a >= $b	Vero se il valore di $a è maggiore o uguale ad $b

5.7.5 - Operatori di incremento e decremento

PHP supporta lo stile C degli operatori di pre- e post-incremento e decremento sulle variabili.

Nome	Esempio	Risultato
Pre-incremento	++$a	Incrementa $a di una unità, inoltre restituisce $a.
Post-incremento	$a++	Restituisce $a, inoltre incrementa $a di una unità.
Pre-decremento	--$a	Decrementa $a di una unità, inoltre restituisce $a.
Post-decremento	$a--	Restituisce $a, inoltre decrementa $a di una unità.

Esempi:

```
<?php
   $a=10;
   print $a++;    // Stamperà 10
   print $a;      // Stamperà 11
   print --$a;    // Stamperà 10
   print $a--;    // Stamperà 10
   print $a;      // Stamperà 9
```

```
?>
```

5.7.6 - Operatori logici

Gli operatori logici si applicano principalmente ad espressioni composte (come vedremo dagli esempi). Di seguito lo specchietto riassuntivo:

Nome	Esempio	Risultato
AND (&&)	$a and $b	Vero se $a e $b sono veri
OR (||)	$a or $b	Vero se uno tra $a o $b è vero
XOR	$a xor $b	Vero se uno tra $a o $b è vero ma non entrambi
NOT	! $a	Vero se $a è falso

Esempi:

```
<?php
   $a = 100;
   //  $a > 10 è VERO
   // ($a > 10) AND ($a < 1000) è VERO
   // ($a > 10) AND ($a < 1000) è VERO
   // ($a < 100) OR ($a > 10) è VERO
   // ($a < 100) OR ($a > 100) è FALSO
   // ($a == 100) XOR ($a > 10) è FALSO
   // (! ($a > 10)) è FALSO
?>
```

5.7.6.1 - Operatore ternario

Un altro operatore condizionale è l'operatore "?:" (o ternario), che opera come in C e molti altri linguaggi.

```
espressione1 ? espressione2 : espressione3;
```

Questa espressione vale *espressione2* se *espressione1* è **TRUE**, e *espressione3* se *espressione1* è **FALSE**.

5.7.7 - Operatori di stringa

L'unico operatore proprio solo delle stringhe è quello di concatenazione: il punto ".".

Tramite questo operatore è possibile concatenare delle stringhe a formarne una nuova come somma delle altre.

Esempio:

```
<?php
   $a = "Questa";
   $a = $a . " è una stringa";
   $a = $a . " creata con la concatenazione";
   // $a adesso vale "Questa è una stringa creata con
concatenazione"
?>
```

5.7.8 - Operatori bitwise

Questi operatori permettono di lavorare "a basso livello" con il contenuto delle variabili, ovvero con la loro "componente minima": i bit. Gli operatori bitwise vi permettono di alterare bit specifici in posizione on oppure off. Se entrambi gli operandi di sinistra e destra sono stringhe, l'operatore bitwise opererà sui caratteri di questa stringa (carattere per carattere).

Nome	Esempio	Risultato
And	$a & $b	Sono impostati ad 1 i bit che sono 1 sia in $a che in $b.
Or	$a | $b	Sono impostati ad 1 i bit che sono 1 in $a oppure in $b.
Xor	$a ^ $b	Sono impostati ad 1 i bit che sono 1 in $a oppure in $b na non quelli che sono entrambi 1
Not	~ $a	Sono impostati ad 1 i bit che sono 0 in $a, e viceversa.

Left shift	$a << $b	Sposta i bit di $a a sinistra di $b passi (ogni passo significa "moltiplica per due")
Right shift	$a >> $b	Sposta i bit di $a a destra di $b passi (ogni passo significa "dividi per due")

5.7.9 - Concatenazione degli operatori

PHP permette un'operazione, chiamata concatenazione delle operazioni, tramite la quale è possibile scrivere in forma più contratta delle espressioni di assegnamento.

Supponiamo di dover aggiungere 5 al valore della nostra variabile $a. Potremmo scrivere:

```
<?php
   $a = $a + 10;
?>
```

La concatenazione degli operatori ci permette però dipoter scrivere la stessa espressione in maniera più compatta:

```
<?php
   $a += 10;
?>
```

La stessa operazione può essere fatta, oltre che con gli operatori aritmetici anche con l'operatore di stringa. In questo modo, l'esempio riportato per l'operatore di concatenazione delle stringhe potrebbe essere così riscritto:

```
<?php
   $a = "Questa";
   $a .= " è una stringa";
   $a .= " creata con la concatenazione";
   // $a adesso vale "Questa è una stringa creata con
concatenazione"
?>
```

5.8 - Array

Un array è un tipo di dato complesso, ovvero un tipo di dato costruito sulla base

dei tipi di dato fondamentali o addiruttura di altri tipi di dati complessi. Nella fattispecie si tratta concettualmente di un insieme di valori di tipo fondamentale o complesso al quale viene dato un unico nome. Ogni elemento all'interno dell'array occuperà un *posto*, identificato da un indice univoco, all'interno dell'insieme. Accedendo all'elemento a quel particolare indice della variabile array, accederemo direttamente al valore dell'elemento.

In questo senso potremmo pensare agli array come ad una *mappa*, ovvero ad una lista di elementi composti ognuno da una *chiave* (l'indice di cui parlavamo) ed un valore (si dice che i valori vengono *mappati* sulle chiavi).

La forma più semplice di array è l'array di valori scalari (interi, floating point, stringhe o boolean).

Il numero di elementi all'interno dell'array si chiama *cardinalità* dell'array.

Gli array si creano tramite la funzione preposta allo scopo: array().

Per accedere ad un elemento di un array, si fa seguire il nome dell'array dalle parentesi quadre "[]" all'interno delle quali specificheremo l'indice cercato.

All'interno di un array possono essere tranquillamente mischiati valori di tipo di dato diversi. Posso dunque creare array che contengono un po' di numeri e un po' di stringhe.

L'indice degli array inizia sempre da 0 (zero), ovvero il primo elemento dell'array si trova all'indice 0.

Facciamo subito degli esempi chiarificatori:

```
<?php
// Creo un array di 4 elementi interi
$myArray = array(10, 20, 30, 40);
// Adesso posso accedere a qualsiasi elemento DEFINITO (o riceverei
un errore)
print ($myArray[1]); // Stamperà 20
print ($myArray[3]); // Stamperà 40
print ($myArray[5]); // Errore ! Il sesto elemento dell'array non è
definito
// Creo di nuovo l'array, mischiando numeri e stringhe
$persona = array("Mario", "Rossi", 45);
print ("Mi chiamo" . $persona[0] . " " . $persona[1] . " ed ho " .
$persona[2] . " anni");
?>
```

5.8.1 - Array multidimensionali

Come abbiamo già accennato il tipo complesso Array può anche essere costituito, a sua volta, di altri tipi complessi. Dunque è possibile creare array di... array (*array multidimensionali*).

```
<?php
// Un array multidimensionali è un array che contiene array
$myMultiArray = array(array(1,2,3), array("a", "b"));
// Creo di nuovo l'array con i dati di due persone
$persona1 = array("Mario", "Rossi", 45);
$persona2 = array("Paolo", "Bianchi", 38);
$persone = array($persona1, $persona2);
/* $persone[0] conterrà l'array con i dati della prima persona.
   Se volessi accedere al primo elemento (il nome) della seconda
   persona, dovrò usare un ulteriore indice */
print ("Il nome della seconda persona è: " . $persone[1][0]);
?>
```

5.8.2 - Array associativi

Dover avere a che fare con indici numerici, per lavorare sugli array, è tuttavia un'operazione scomoda e portatrice di errori. PHP fornisce un meccanismo per accedere agli elementi di un array in maniera più "naturale". Questo meccanismo sfrutta appunto il concetto degli array associativi.

In pratica invece di accedere ad un elemento di un array tramite il suo indice numerico, si può accedervi tramite un nome più esplicativo. Se infatti un certo elemento di un array contiene il *nome* della nostra persona, perchè non accederrvi direttamente tramite una chiave che si chiama proprio "nome" ?

Un array associativo si definisce in maniera diversa rispetto ad un normale array, usando però ancora la stessa funzione di inizializzazione array().

```
<?php
   $persona = array ( "nome" => "Mario",
                      "cognome" => "Rossi",
                      "età" => 45);
```

```
?>
```

Notare l'uso del simbolo “=>”, con il quale si associa ad una *chiave* esplicativa il proprio *valore* (la particolare indentazione dell'esempio è solo per chiarezza).

A questo punto, senza preoccuparsi di quale sia l'indice del nome potremmo scrivere:

```
<?php
   $persona = array ( “nome” => “Mario”,
                      “cognome” => “Rossi”,
                      “età” => 45);
   print (“Il mio nome è “ . $persona[“nome”]);
   print (“Il mio cognome è “ . $persona[“cognome”]);
   print (“La mia età è “ . $persona[“età”]);
?>
```

L'operazione è ancora quella di mappatura di un valore su una chiave. Se avessi infatti scritto qualcosa come:

```
<?php
   $persona = array ( 0 => “Mario”,
                      1 => “Rossi”,
                      2 => 45);
?>
```

avrei creato, in maniera più complessa, lo stesso array che abbiamo incontrato precedentemente. In questo modo con $persona[0] avrei acceduto al primo elemento dell'array, come se questo fosse un “normale” array.

5.9 - Print

Il comando print è il comando principe per la visualizzazione di dati.

Il concetto di visualizzazione di dati deve essere concepito come “scrivere nel flusso di dati HTML che ritorna al server, e poi viene direttamente rispedito al UA che ha fatto la richiesta della pagina”.

In pratica, all'atto pratico, quello che stampiamo con il comando print (è un

comando, non una funzione) andrà a comporre direttamente la pagina HTML (o altro) che l'utente visualizzerà sul prioprio browser.

E` molto comune anche scrivere direttamente dell'HTML, tramite il comando print. Quando l'utente dovesse eventualmente visualizzare il contenuto della propria pagina HTML, quello che vedrà sara ovviamente solo e soltanto HTML.

Spesso il comando “echo” viene usato al posto di print. I due comandi sono perfettamente intercambiabili.

5.10 - Test condizionali

Ogni linguaggio che si rispetti adotta dei meccanismi sintattici per “prendere delle decisioni” durante l'esecuzione del codice dello script. PHP non fa eccezione, adottando per questo scopo la struttura IF – ELSE e il costrutto SWITCH.

5.10.1 - Il comando IF

La sintassi del comando IF è così schematizzabile:

```
if (<espressione>)
{
   [set di istruzioni se <espressione> è VERA]
}
else
{
   [set di istruzioni se <espressione> è FALSA]
}
```

Innanzitutto la sintassi:

- l'espressione che l'istruzione IF deve valutare deve essere racchiusa sempre tra parentesi tonde
- L'istruzione ELSE con accetta nessun parametro
- L'istruzione ELSE è opzionale. Quando non necessario lo si può omettere.
- I blocchi di istruzioni devono essere racchiusi tra parentesi graffe (eccezione:

quando il set di istruzioni è composto da una sola riga, le graffe sono opzionali)

- L'instruzione IF è nidificabile, ovvero all'interno del blocco di istruzioni di un IF o di un ELSE può essere presente a sua volta un altro IF.

Ogni volta che l'interprete PHP raggiunge una IF, per prima cosa viene *valutata* l'espressione all'interno delle parentesi tonde. Di questa espressione verrà considerato il fatto che sia VERA o FALSA. Se è vera allora verrà eseguito il codice sottostante la if, compreso tra parentesi graffe, altrimenti se presente il codice sottostante l'ELSE.

Vediamo degli esempi:

```
<?php
$a = 10;
if ($a < 10)
{
   print "\$a è minore di 10";
}
else
{
   print "\$a è maggiore di 10";
}
?>
```

L'esempio precedente è **sbagliato**. Verrà stampato che $a è maggiore di dieci, mentre in realtà $a è UGUALE a 10.

Correggiamolo:

```
<?php
$a = 10;
if ($a < 10)
{
   print "\$a è minore di 10";
}
else
```

```
{
   if ($a > 10)
   {
        print "\$a è maggiore di 10";
   }
   else
   {
        print "\$a è uguale a 10";
   }
}
?>
```

Un altro esempio:

```
<?php
   $a = 34;
   $b = 10;
   if ($a > 10 and $b < 4)
   {
        print "espressione vera";
   }
   else
   {
        print "espressione falsa";
   }
?>
```

In questo caso l'espressione che viene valutata è ($a > 10 and $b < 4). Nell'ipotesi fatta, ovvero che $a valga 34 e $b valga 10, l'espressione è FALSA.

Un ulteriore sequenza di esempi:

```
<?php
   $a = 35;
   if ($a) # esempio 1
   {
```

```
        print "\$a è diverso da 0";
    }
    if (true) # esempio 2
    {
        print "Stamperò comunque questa frase";
    }
    if (0) # esempio 3
    {
        print "Non stamperò mai questa frase";
    }
    if ($a = 1) #esempio 4
    {
        print "Stamperò questa frase";
    }
    if ($a = 0) #esempio 5
    {
        print "Non stamperò mai questa frase";
    }
    if ($a == 0) #esempio 6
    {
        print "Stamperò questa frase";
    }
?>
```

Prima di analizzare l'esempio una premessa importante: qualsiasi valore diverso da zero è valutato com VERO. Il valore 0 è invece sempre valutato come FALSO.

Detto questo vediamo di capire l'esempio:

1) L'espressione "$a" vale quanto il valore di $a, ovvero 35. Il valore è diverso da zero, dunque la stringa viene stampata
2) L'espressione "true" è, ovviamente, valutata come VERA (true, appunto). La stringa viene sempre stampata
3) L'espressione "0" vale quanto il numero ovvero, 0. 0 è sempre falso per cui la

stringa non viene stampata.

4) L'espressione di assegnamento “$a = 1” viene valutata al valore dell'assegnamento, dunque a 1. 1 è vero per cui la stringa viene stampata.

5) Per lo stesso motivo di 4) la stringa non viene stampata

6) Dopo l'assegnamento avvenuto in 5), $a vale 0. L'espressione che rappresenta un test di confronto per l'uguaglianza “==” è dunque vero e la stringa verrà stampata.

> Uno degli errori più comuni in assoluto è scambiare l'operatore di assegnamento “=” con l'operatore di test di uguaglianza in un'istruzione IF. Il loro comportamento, come abbiamo visto, è assolutamente diverso, quando ne viene valutato il valore. Fate sempre molta attenzione a quello che viene scritto !

5.10.2 - Il comando Switch

Il comando switch permette di scrivere in maniera più elegante i casi nei quali ci si trovi costretti a fare un test (tramite IF), su tutta la serie di valori possibili che una variabile può assumere.

Il caso seguente, dove vengono controllati in cascata tutti i valori possibili di una ipotetica variabile $var, per esempio:

```
<?php
if ($var == 10)
{
    // azioni relative al valore 10 di $var...
}
if ($var == 10)
{
    // azioni relative al valore 10 di $var...
}
if ($var == 20)
{
    // azioni relative al valore 20 di $var...
}
if ($var == 30)
```

```
{
    // azioni relative al valore 30 di $var...
}
if ($var != 10 and $var != 20 and $var != 30)
{
    // azioni relative ad un valore inaspettato di $var
}
?>
```

può essere riscritto in maniera più compatta tramite il comando SWITCH:

```
<?php
switch ($var)
{
   case "10":
        // azioni relative al valore 10 di $var
        break;
   case "20":
        // azioni relative al valore 20 di $var
        break;
   case "30":
        // azioni relative al valore 30 di $var
        break;
   default:
        // azioni relative ad un valore inaspettato di
$var
        break;
}
?>
```

Ogni singolo *case* deve essere concluso con un'istruzione "break". Il break interrompe di fatto lo switch e ne esce. Se il break non fosse presente, il flusso continuerebbe valutando il case successivo.

Questo può essere talvolta un effetto voluto. Se, nelle esempio di prima, nel caso

del valore “20” e “30” della variabile $var fossere da eseguire le stesse operazioni (una sorta di AND), allora si sarebbe potuto scrivere:

```
<?php
switch ($var)
{
   case “10”:
        // azioni relative al valore 10 di $var
        break;
   case “20”:
   case “30”:
        // azioni relative al valore 20 e 30 di $var
        break;
   default:
        // azioni relative ad un valore inaspettato di
$var
        break;
}
?>
```

Il valore controllabile tramite ogni *case* deve essere un valore espressione costante.

Il seguent

5.11 - Cicli

Le istruzioni che riguardano la creazione di cicli, servono semplicemente per effettuare più volte lo stesso set di operazioni. Pensate per esempio al dover incrementare di uno una variabile fino a che questa non raggiunga un valore determinato. In questo caso inserirò l'incremento di uno all'interno del ciclo e farò ripetere l'operazione, tramite l'opportuna istruzione, tante volte finché la variabile non raggiunga il valore prefissato.
Esistono due costrutti principali per effettuare dei cicli ripetuti: il WHILE e il FOR.

5.11.1 - Ciclo WHILE

Vediamone schematicamente la sintassi:

```
WHILE (<espressione>)
{
      [set di istruzioni finché <espressione> è VERA]
}
```

Anche in questo caso, similarmente al IF, ci troviamo a valutare l'espressione tra le parentesi dell'istruzione while. Finché tale espressione rimarrà vera, allora verranno eseguite in continuazione le istruzioni all'interno del blocco compreso tra le parentesi graffe. E` ovvio che il valore dell'espressione dovrà in qualche modo essere modificata dalle istruzioni all'interno del blocco. Se così non fosse, il ciclo non terminerebbe mai, e il programma si bloccherebbe all'infinito (o meglio, finché il server non decidesse di terminarlo).

Esempi:

```
<?php
   $a = 0;
   while ($a < 100)
   {
        $a++;
   }
   print $a;
?>
```

In questo esempio (abbastanza ovvio e inutile), il ciclo sarà eseguito 100 volte. Appena la variabile $a assumerà il valore 100, il ciclo terminerà e ne verrà stampato il valore.

Una variante del ciclo WHILE è il DO – WHILE.

Nel while, l'espressione viene prima valutata e, nel caso risulti vera, il ciclo verrà iniziato. Se l'espressione iniziale fosse subito falsa, o non fosse valutabile, il ciclo non verrebbe eseguito neanche una volta. Possono esserci casi nei quali sia richiesto che il ciclo venga eseguito ALMENO una volta. In questo caso si usa il DO WHILE.

5.11.2 - Ciclo FOR

Il ciclo FOR ha la seguente sintassi:

```
FOR (<inizializzazione>; <test>; <istruzioni>)
{
      [set di istruzioni finché <test> è VERA]
      ([esecuzione di <istruzioni>])
}
```

In un ciclo FOR viene prima di tutto eseguita (quando presente) l'istruzione di inizializzazione presente come primo suo parametro e poi, per ogni ciclo, viene valutata “espressione1”. Se questa risulta VERA allora viene eseguito il codice all'interno del blocco di istruzioni. Alla fine del blocco (automaticamente), viene eseguita l'istruzione <istruzioni>. Il ciclo ricomincia dal test.

Esempio:

```
<?php
for ($i=0; $i < 100; $i++)
{
   print “Valore di \$i è $i<br>”;
}
?>
```

Nell'esempio precedente viene subito inizializzata la variabile $i al valore “0”. Il ciclo dovrà terminare appena $i raggiungerà il valore 100. Alla fine di ogni ciclo, il valore della variabile $i verrà incrementato di uno.

Le istruzioni di inizializzazione, ovviamente, vengono eseguite soltano all'inizio del ciclo.

5.11.3 - Foreach

Tramite il ciclo foreach è possibile ciclare su tutti i valori di un array *deferenziando*, ad ogni ciclo, il successivo elemento, fintanto che un elemento esiste all'interno dell'array.

Le sintassi possibili sono due.

La prima

```
foreach (<array> as $var)
{
   // Istruzioni...
}
```

itererà su tutti gli elementi dell'array valorizzando di volta in volta la variabile $var con il contenuto del corrente elemento dell'array.

Esempio:

```
<?php
$arr = array(10, 20, 30);
$i = 0;
foreach ($arr as $var)
{
   print "La posizione $i dell'array contiene il valore
$var <br>";
   $i++;
}
?>
```

La seconda sintassi invece permette di estrarre per ogni elemento dell'array la chiave e il valore (utile negli array associativi):

```
foreach (<array> as $key => $val)
{
   // Istruzioni...
}
```

Segue esempio di utilizzo

```
<?php
$arr = array("nome" => "Roberto",
             "cognome" => "Rossi",
             "eta" => "34");
```

```
foreach ($arr as $key => $val)
{
   print "Il valore della chiave $key dell'array è
$val<br>";
}
?>
```

E' importante notare che la variabile di appoggio del ciclo foreach (che negli esempi precedenti era $val, $key e $var) conterrà sempre *una copia* dell'elemento dell'array, e non il suo attuale valore. Modificando una di queste variabile di appoggio non si modificherebbe dunque il valore dell'elemento dell'array.

5.12 - Include e require

Tramite le istruzioni REQUIRE ed INCLUDE è possibile inserire (includere, appunto) uno script PHP all'interno di un altro.

Il comportamento, sintassi e metodo di ricerca del file da includere è identico per tutti e due i costrutti. L'unica differenza sta nel fatto che se INCLUDE non dovesse trovare il file richiesto, questa emetterà uno *warning* e lo script all'interno del quale si trova il comando continuerà comunque la sua esecuzione. REQUIRE, invece, non trovando il file emetterà un *fatal error* e lo script si interromperà immediatamente.

Generalmente si usa, dunque, sempre il REQUIRE.

Per includere un file si usa la seguente sintassi:

```
require ("file_da_includere.php");
```

Il file verrà cercato nella *include_path* (modificabile nel php.ini o anche a runtime tramite la funzione set_include_path).

Se non modificato, l'include_path conterrà solo la directory corrente ".".

Per file inclusi da file che sono al loro volta inclusi, allora verrà effettuata una ricerca anche nella directory corrente dello script incluso.

Se per esempio avessimo il file a.php nella directory /www contenente:

```
<?php
require ("include/b.php");
?>
```

allora b.php verrebbe cercato in "/www/include".
Se all'interno di b.php fosse però scritto:

```
<?php
require ("c.php");
?>
```

allora il file c.php verrebbe prima cercato nella directory corrente (che sarà sempre /www) e poi nella directory corrente dello script b.php, ovvero "/www/include".
Il file incluso, da un punto di vista logico, farà parte in tutto e per tutto del file che lo include. Tutte le variabili, le funzioni e le classi definite nel file incluso saranno visibili dal file che lo include proprio come se fossero stati definiti nel file stesso.
Dal momento che il parser PHP deve aprire uno script da includere, allora passerà dalla modalità PHP alla modalità HTML. Per questo motivo ogni file che dovrà essere incluso dovrà contenere all'inizio e alla fine i tag di inizio e fine php (<?php e ?>).
Quando i file da includere dovessero aumentare e le loro relazioni (chi include chi e quando) si dovessere complicare, può essere comodo utilizzare dei costrutti simili: REQUIRE_ONCE (e relativo INCLUDE_ONCE).
Tali costrutti assicurano che il file che si richiede sia incluso una sola volta.
Il seguente esempio che fa uso di un concetto, le funzioni, non ancora introdotto tuttavia rende intuitivo un caso di possibile utilizzo della require_once.
Si supponga l'esistenza di un file inc.php contenente la seguenre dichiarazione di funzione:

```
<?php
function moltiplica($a, $b)
{
   return ($a * $b);
}
?>
```

Supponiamo poi un file main.php che contenga:

```
<?php
require (“inc.php”);
// Segue una serie lunghissima di istruzioni
require (“inc.php”);
?>
```

L'esempio mostra come, per errore, sia stato incluso due volte lo stesso file.
Il parser PHP renderebbe un errore di *funzione definita due volte* in quanto, con la doppia inclusione, la funzione *moltiplica* verrebbe definita, appunto, due volte.
Per ovviare a questo inconveniente, quando la complessità del nostro programma, rendesse più sicuro preventivarci in qualche modo, potremmo scrivere:

```
<?php
require_once (“inc.php”);
// Segue una serie lunghissima di istruzioni
require_once (“inc.php”);
?>
```

In questo modo soltanto il PRIMO require effettuerebbe l'inclusione del file. Il secondo non sortirebbe nessun effetto.

6 - Concetti avanzati

6.1 - Le funzioni

Le funzioni sono un modo per raggruppare efficacemente del codice PHP che altrimenti si dovrebbe ripetere, uguale, più e più volte all'interno dei nostri script. Tutte le volte che ci trovassimo a scrivere più di una volta lo stesso codice in uno o più script del nostro progetto, ci sono buone possibilità che quello che stiamo scrivendo (ripetendo), possa essere il corpo di una funzione.

Come le variabili, anche le funzioni hanno un nome (e le regole di nomenclatura introdotte per i nomi di variabili, possono essere applicate anche a quelli delle funzioni).

Una nuova funzione viene definite tramite la parola chiave FUNCTION, seguita dal nome che abbiamo deciso di darle e da una coppia di parentesi tonde, all'interno delle quali possono trovarsi gli *eventuali parametri della funzione* separati l'uno dall'altro da una virgola. Il corpo della funzione, ovvero il codice vero e proprio che compone la funzione, deve essere racchiuso tra parentesi graffe.

I parametri di una funzione sono delle variabili, passate dal *chiamante*, con le quali la funzione deve in qualche modo interagire.

Una funzione può *ritornare* un valore al chiamante tramite la parola chiave RETURN.

Le funzioni vengono *chiamate* invocando il loro nome seguito dalle parentesi tonde, contenenti eventualmente i parametri necessari.

Non appena il PHP incontra la chiamata ad una funzione, questo *salta* al corpo della funzione, lo esegue e al termine ritorna al punto dove la funzione era stata chiamata.

Il primo esempio, banale, illustra come una funzione si possa definire e chiamare:

```
<?php
print "Prima funzione.<br>";
sayHello();
print "Chiamata con successo.<br>";
function sayHello()
{
```

```
   print "Hello !<br>";
}
?>
```

L'output di questo script sarà il seguente:

```
Prima funzione.
Hello !
Chiamata con successo.
```

Una volta stampata la stringa "Prima funzione", il PHP salterà al corpo della funzione sayHello, lo eseguirà (stampando il saluto), e poi riprenderà il flusso dalla chiamata della funzione (stampando "Chiamata con successo").

Quello che è importante notare è che la funzione, essendo una "scatola nera", non verrà eseguita anche dopo che sia stata stampata l'ultima scritta. Lo script terminerà alla fine del print di "Chiamata con successo".
Per introdurre il passaggio di parametri, vediamo adesso una semplice funziona che stampi la moltiplicazione tra due numeri:

```
<?php
print "Il prodotto di 4 e 20 è";
moltiplica(4, 20);
function moltiplica ($a, $b)
{
   print "$a * $b";
}
?>
```

I numeri 4 e 20 saranno incamerati come valori dei parametri $a e $b rispettivamente.
L'output dello script sarà ovviamente:

```
80
```

Le funzioni, come detto, possono anche ritornare dei valori. Modifichiamo la nostra funzione in modo che, invece di stampre il valore della moltiplicazione ce ne ritorni il risultato:

```
<?php
$risultato = moltiplica(4, 20);
print "Il prodotto di 4 e 20 è $risultato";

function moltiplica ($a, $b)
{
   return "$a * $b";
}
?>
```

Notare come il valore di ritorno di una funzione possa essere direttamente assegnato ad una variabile.

6.1.1 - Le funzioni di libreria

Il PHP mette a disposizione centinaia di funzioni predefinite (dette anche *di libreria*) utili agli scopi più vari. Esistono funzioni che permettono di manipolare le date, altre che operano particolari funzioni sulle stringhe altre ancora espressamente pensate per la gestione degli array, etc. Inoltre, essendo un linguaggio estensibile, ogni modulo che andremo ad inserire all'interno del PHP che gira sul nostro server aggiungerà a sua volta un altro set di funzioni addizionali (per esempio, il modulo GD permette di effettuare operazioni grafiche 2D, mentre per l'accesso al database MySQL è possibile utilizzare tutte le funzioni che il relativo modulo mette a disposizione). L'elenco è dunque lunghissimo e sicuramente queste dispense, scritte come complemento alla docenza in aula, non vogliono essere una manuale esaustivo di tutte le funzioni di libreria, e tantomeno di quelle fornite dalle decine di moduli disponibili per il PHP.
Si faccia sempre riferimento alla guida online (anche scaricabile, in modo da avere sempre una guida anche offline) per i dettagli e la sintassi di ogni singola funzione. Il sito php.net è inoltre provvisto di un'efficente motore di ricerca che vi

permetterà di trovare tutto quello di cui si ha bisogno (sempre ammesso che... esista!). Se all'inizio orientarsi nel mare delle funzioni disponibili potrà sembrare un'impresa disperata, vi assicuriamo che con l'aumentare dell'esperienza sarà sempre più veloce attingere al manuale di riferimento, andano a colpo sicuro a trovare quello che ci interessa.
Un elenco, non esaustivo, delle possibilità offerte dalle funzioni standard (o quasi) di PHP è il seguente:

- ✔ Funzioni sulle variabili
- ✔ Funzioni sulle stringe
- ✔ Espressioni regolari
- ✔ Interazione con il filesystem (file, directory)
- ✔ Gestione Log ed Errori
- ✔ Esecuzione di programmi
- ✔ Opzioni e informazioni
- ✔ Gestione sessioni e cookie
- ✔ Funzioni HTTP
- ✔ Funzioni per gli URL
- ✔ Funzioni di rete
- ✔ Database (Mysql, Postgresql)
- ✔ Immagini con GD
- ✔ Encoding dei dati
- ✔ Funzioni per data e ora
- ✔ Funzioni per XML
- ✔ Pear (autenticazione, form, SQL, template con SMARTY)

6.1.2 - Consultare la guida online

Di seguito si vuole illustrare la struttura del manuale online e il modo in cui esso va consultato per cercare, scegliere ed infine utilizzare le innumerevoli funzioni predefinite del linguaggio. Come sempre il modo migliore è l'utilizzo di un semplice esempio. Si supponga di voler realizzare uno programma che visualizzi una porzione di una stringa. Bisogna come prima cosa individuare le funzioni che permettono di manipolare le stringhe.

Nella sezione "IV. Guida Funzioni" sono catalogate le funzioni messe a disposizione del programmatore, esse sono raggruppate per argomento. La sottosezione "LXXXIII. String functions" contiene le funzioni dedicate alla gestione delle stringhe. Come prima cosa si trova una descrizione della sezione e poi un lungo elenco, in ordine alfabetico, di funzioni con accanto una breve ma molto intuitiva descrizione. La funzione substr() potrebbe risolvere il problema sollevato nell'esempio.

Nel primo blocco della pagina descrittiva si trova il nome della funzione subito seguito dalle versioni dell'interprete PHP che la gestiscono e da una breve descrizione. Nel caso particolare:

```
substr
(PHP 3, PHP 4 >= 4.0.0)
substr -- Return part of a string
```

A seguire la descrizione approfondita che inizia con una delle parti più importanti della documentazione, infatti, in una sola riga viene spiegato l'intero funzionamento del comando. Nell'esempio proposto:

```
string substr (string string, int start [, int
length])
```

Come detto questa è forse la parte più importante, la prima parola descrive il tipo di dato restituito dalla funzione, nel caso particolare string, dunque la funzione restituisce una stringa. Dopo il nome della funzione tra parentesi i valori che la funzione può accettare ed il relativo tipo di dato, va sottolineato che tra parentesi quadre vengono elencati i parametri non obbligatori e il relativo tipo di dato.

Nel particolare la funzione accetta un primo dato di tipo stringa, separato da virgola il numero del carattere di partenza di tipo intero (si parte sempre da 0 e non da 1) ed infine come dato opzionale la lunghezza, di tipo intero, della porzione di stringa da estrarre.

A questo schema seguono una descrizione approfondita ed una serie di esempi di funzionamento. Gli esempi se presenti sono molto utili ed autoesplicativi, inoltre trattano varie eccezioni ed eventuali casi particolari.

In fine ma non meno importante la sezione contenente le funzioni correlate o in qualche modo relazionate con quella in oggetto. Il manuale spesso suggerisce di consultare anche altre funzioni, in questo caso:

```
   See also strrchr() and ereg().
```

Spesso non esiste un unico modo per raggiungere la soluzione del problema e spesso questa ultima sezione fornisce degli spunti molto interessanti che possono portare ad una soluzione più semplice e brillante, il consiglio è di non sottovalutarla.

Nella maggior parte dei casi, inoltre, è possibile usufruire dei consigli e anche di pezzi di codice che gli utenti del sito hanno voluto lasciare sulla pagine del manuale che stiamo leggendo. Spesso si trovano consigli talmente preziosi da farvi risparmiare, letteralmente, ore di lavoro !

6.1.3 - Valore di default dei parametri

Quando per una funzione si definiscono dei parametri è necessario che, quando la si chiama, tutti i parametri richiesti siano forniti (o altrimenti si verificherebbe uno warning).

E' tuttavia possibile creare delle funzioni che abbiano dei parametri opzionali, i quali prenderanno, quando non ricevuti dal chiamante, un valore di default.

Un esempio chiarirà il concetto:

```
<?php
stampaParametro("Parametro passato");
stampaParametro();
function stampaParametro($par = "Parametro non passato")
{
   print $par . ".<br>";
```

```
}
?>
```

Come si intuisce facilmente, l'output del precedente esempio sarà il seguente:

```
Parametro passato.
Parametro non passato.
```

6.1.4 - Passaggio di paramentri per reference

Nei casi visti finora, quando passavamo un parametro ad una funzione, questo veniva passato per valore. Questo vuol dire che, anche modificando il valore del parametro all'interno della funzione, la variabile passata dal chiamente non sarebbe modificata.

Esempio:

```
<?php
$val = 10;
incrementa($val);
print $val;
function incrementa($par)
{
   $par++;
}
?>
```

Dal momento che la funzione “incrementa” riceve il parametro *per valore*, essa lavorerà su una *copia* della variabile $val, non su quella stessa. L'ouput dello script sarà dunque:

```
10
```

E' possibile tuttavia, usando il passaggio di parametri *per reference*, fare in modo che la funzioni lavori su un alias della variabile e non su una sua copia. In questo modo, modificando il valore dell'alias verrà modificato anche il valore della variabile iniziale.
Per effettuare il passaggio per reference, si deve prependere nella lista di parametri il carattere & al nome del parametro:

```
<?php
$val = 10;
incrementa($val);
print $val;
function incrementa(&$par)
{
   $par++;
}
?>
```

Adesso l'output del nostro script sarà proprio

```
11
```

in quanto la funzione lavorerà sulla stessa area di memoria identificata da $val. $par è dunque una reference (un alias...) della variabile $val.

6.2 - Visibilità delle variabili

Quando si definisce una variabile, questa assume una (cosiddettà) visibilità.
Essa (e dunque anche il suo valore), sarà “vista” solo in certi ambiti mentre in altri tale variabile risulterà sconosciuta (e se si tenta di usarla si otterrà una segnalazione).
Quando si definisce una variabile all'interno di uno script, se questa non è stata definita all'interno di una funzione o di una classe (che vedremo più avanti) allora assumerà una visibilità (detto anche *scope*) *globale*. Chiunque, da qualunque altra

parte *dello stesso script*, avrà a disposizione il valore di questa variabile.
Una variabile definita, invece, all'interno di una classe o di una funzione assumerà una visibilità solo all'interno di quel blocco di istruzioni, non oltre. Si dice in quel caso che la variabile è *locale* alla funzione (o alla classe).
Ecco come appare, semplicemente, una variabile globale:

```
<?php
// Variabile globale, visibile per tutto lo script
$variabile = "Una riga di testo";
// Nello resto dello script sarà possibile riferirsi
// alla variabile $variabile.
?>
```

Tramite un esempio (sbagliato) esprimiamo adesso il concetto di variabile locale:

```
<?php
$var1 = "Test";
sommaDueNumeri(10, 30);
print $risultato;
function sommaDueNumeri($a, $b)
{
   $risultato = $a + $b;
}
?>
```

L'esempio precedente non produrrà affatto il risultato sperato. Invece della somma dei due numeri ci ritroveremo con un bel messaggio di avvertimento del PHP, che dirà che stiamo cercando di usare una variabile non *inizializzata*.
La variabile $risultato, infatti, è stata valorizzata (e dunque inizializzata) in uno scope locale (quello della funzione sommaDueNumeri), ma poi è stata utilizzata in uno scope globale, dove tale variabile non esiste!
La "vita" della variabile $risultato, dunque, inizia e termina con la chiamata alla

funzione: una volta che la funzione termina, tutte le variabili locali al suo scope vengono distrutte.
Tutte le variabili utilizzate all'interno di una funzione o di una classe, sono dunque gestite come se fossero locali ad essa. Talvolta è però necessario fare esplicito riferimento, dall'interno di una funzione o di una classe, ad una variabile che si trova nello *scope* globale. In questo caso occorre "predichiarare" l'utilizzo di questa variabile facendone precedere il nome dalla parola chiave GLOBAL
Inoltre, se definissimo una variabile globale con il nome $dataNascita e una variabile locale con lo stesso nome, avremo di fatto DUE variabili con lo stesso nome ma in due scope diversi e, duqnue, con due valori diversi.

```
<?php
$dataNascita = "10 aprile 1944";
stampaDataNascita();

function stampaDataNascita()
{
   if (!isset($dataNascita))
        $dataNascita = "1 gennaio 1900";
   print $dataNascita;
}
?>
```

L'esempio precedente non sortirebbe gli effetti desiderati.
Impostiamo la variabile globale $dataNascita ad un valore predefinito e poi chiamiamo la funzione che ci stamperà questa data, facendo prima un controllo.
Il controllo verificherà che quella variabile sia stata definita da qualche altra parte e, nel caso non lo fosse, provvederà a imporgli un valore di default.
Ma la $dataNascita interna alla funzione è locale alla stessa e dunque sarà una nuova variabile e (ovviamente) non sarà mai stata settata. Dunque, la variabile locale $dataNascita varrà SEMPRE "1 gennaio 1900".

Vediamo come correggere questo esempio, utilizzando il preannunciato costrutto GLOBAL:

```
<?php
$dataNascita = "10 aprile 1944";
stampaDataNascita();

function stampaDataNascita()
{
   global $dataNascita;
   if (!isset($dataNascita))
        $dataNascita = "1 gennaio 1900";
   print $dataNascita;
}
?>
```

Con l'aggiunta della riga in neretto, il nostro esempio sarà corretto. Il comando "global", infatti, provvederà in un certo senso ad "importare" all'interno dello scope della funzione la variabile $dataNascita che si trova nello scope globale rendendola, di fatto, utilizzabile dalla funzione stessa.

6.2.1 - Variabili superglobali

In PHP esistono anche tutta una serie di variabili dette superglobali, ovvero visibili a qulasiasi script, in qualsiasi momento. Le variabili superglobali sono visibili da subito anche all'interno delle funzioni e delle classi senza la necessità di importarle prima con il comando global.

Non è possibile, da utente, definire una variabile superglobale. Di questo tipo di variabili fanno parte le variabili di ambiente del sistema (per esempio $_PATH), le variabili che contengono i dati di una pagina ricevuta in POST o in GET (gli array associativi $_GET, $_POST, $_REQUEST) , le variabili di SESSIONE (l'array associativo $_SESSION), le variabili che fanno parte della trnsazione HTTP (l'array associativo $_SERVER), eccetera. Spesso ci si riferisce ad esse come le variabili

predefinite.

6.3 - Classi e oggetti

Una classe non è altro che un contenitore (un pò come le funzioni, che abbiamo visto, ma molto più sofisticato e potente) all'interno del quale è possibile definire tutta una serie di funzioni e variabili che in qualche modo descrivono il comportamento e le proprietà di un *oggetto*.

Quando vorremo usare un oggetto di quel tipo, potremo dunque *istanziare* la classe che lo descrive e manipolarlo tramite le funzioni (detti *metodi della classe*) che vi abbiamo definito.

Per farsi un'idea del concetto, pensiamo "all'oggetto" veicolo.

Un oggetto di tipo veicolo (sia esso una bicicletta, una moto o un auto) avrà comunque delle caratteristiche comuni che potranno essere descritte tramite un elenco di *proprietà* dello stesso. Per modificare (o definire) le proprietà di un veicolo potrà essere necessario ricorrere a dei *metodi* che manipolano le sue proprietà.

Una classe può anche essere *estesa*, in modo da far *derivare* da un oggetto generico (come appunto un veicolo), una classe più particolare atta, ad esempio, alla descrizione di un'automobile.

Vediamo adesso la sintassi di base di una classe:

```
class NomeDellaClasse
{
   var $proprieta_1;
   var $proprieta_2;
   [...]
   var $proprieta_N;

   function NomeDellaClasse()
   {
        // Inizializzazione delle proprietà
        $this->proprieta_1 = 0;
        // Codice...
   }
```

```
    function metodo_1()
    {
         // Codice...
    }
    function metodo_1()
    {
         // Codice...
    }
    [...]
    function metodo_N()
    {
         // Codice...
    }
}
```

Dalla sintassi esposta, si enunciano le seguenti regole:

- ogni classe viene definita tramite la parola chiave CLASS seguita dal nome della classe che andiamo a definire.
- Il corpo della classe è racchiuso tra parentesi graffe.
- Le proprietà vengono dichiarate subito dopo la graffa di apertura della definizione di classe, ognuna preceduta dalla parola chiave VAR. Non si può inizializzare una proprietà al momento della sua dichiarazione.
- All'interno della classe deve essere presente una funzione (metodo...) con lo stesso nome della classe. Tale metodo "speciale" di chiama *costruttore* e viene invocato sempre non appena la classe viene instanziata, al momento del suo utilizzo. All'interno del costruttore devono essere inizializzate tutte le proprietà
- I metodi sono dichiarati come normali funzioni e sono le regole delle funzioni.
- All'interno della classe, le singole proprietà della stessa sono accessibili tramite la speciale sintassi $THIS seguito dai due caratteri "->" (ad indicare una sorta di freccia) e dal nome della proprietà. La variabile speciale $THIS rappresenta la

classe stessa (e i caratteri "->" sono la sintassi per accedere ad un membro (sia esso metodo o proprietà) di una classe).

Vediamo dunque come sarebbe possibile definire una ipotetica classe Veicolo.

```
<?php
class Veicolo
{
   var $numeroDiRuote;
   var $colore;
   var $modello;
   var $carburante;

   function Veicolo()
   {
        $this->numeroDiRuote = 0;
        $this->colore = "";
        $this->modello = "";
        $this->carburante = "";
   }

   function setNumeroDiRuote($n)
   {
        $this->numeroDiRuote = $n;
   }

   function setColore($c)
   {
        $this->colore = $c;
   }

   function setModello($m)
   {
```

```
        $this->modello = $m;
    }

    function setCarburante($c)
    {
        $this->numeroDiRuote = $c;
    }

    function getNumeroDiRuote()
    {
        return($this->numeroDiRuote);
    }

    function getColore()
    {
        return($this->colore);
    }

    function getModello()
    {
        return($this->modello);
    }

    function getCarburante()
    {
        return($this->carburante);
    }
}
?>
```

Convenzionalmente, i metodi che valorizzano le proprietà e quelli che ne ritornano il valore sono detti, rispettivamente, *setters e getters*.

Chi avesse già esperienza di programmazione ad oggetti (C++, per esempio) potrà notare che in PHP non è possibile definire un *metodo distruttore e* che non è possibile definire delle proprietà private o pubbliche (in PHP le proprietà sono tutte pubbliche). Inoltre in PHP non esiste il concetto di ereditarietà multipla.

Abbiamo dunque definito la classe Veicolo. Vediamo adesso come poterla usare:

```
<?php
$bicicletta = new Veicolo();
$bicicletta->setColore("Rosso");
$bicicletta->setNumeroDiRuote(2);
$bicicletta->setModello("Mountain Bike");
print "Ho appena definito una bicicletta di tipo " .
      $bicicletta->getModello() .
      "di colore " .
      $bicicletta->getColore();
?>
```

Dall'esempio di deducono le seguenti regole sintattiche:

- un oggetto (istanza di una classe) si crea tramite la parola chiave NEW seguita dal costruttore della classe.
- Una volta creato, è possibile riferirsi ai metodi di un oggetto tramite i due caratteri "->"

Non è necessario impostare tutte le proprietà di un oggetto (come, nell'esempio precedente non abbiamo impostato la proprietà carburante).

Abbiamo accennato al fatto che una classe può essere estesa in maniera da creare delle sottoclassi più specializzate. Nel prossimo esempio vediamo come sia possibile creare una classe Automobile partendo dalla definizione della *superclasse* Veicolo (la classe padre).

```
<?
```

```
class Automobile extends Veicolo
{
   var $marca;
   var $cilindrata;
   var $numeroCilindri;

   function Automobile($m)
   {
        parent::Veicolo();
        $this->marca = $marca;
        $this->cilindrata = 0;
        $this->numeroPistoni = 0;
        $this->numeroDiRuote = 4;
   }
   [seguono setters e getters delle 3 proprietà]
}
?>
```

Ancora una volta, a partire dall'esempio traiamo delle regole sintattiche:

- una classe diventa un'estensione di un'altra tramite l'utilizzo della parola chiave EXTENDS nella parte di dichiarazione della stessa.
- All'interno del costruttore abbiamo usato una particolare sintassi per invocare il costruttore della classe padre (tramite il riferimento alla variabile speciale "parent" (notare la mancanza del $)). La sintassi è "nome della classe" :: metodo(); Questa sintassi serve per invocare *staticamente* un metodo di una classe, ovvero invocarlo senza prima dover istanziare un oggetto.
- Notare che abbiamo utilizzato un parametro nel costruttore.

Una volta creata una classe che estende una classe padre, questa *erediterà* automaticamente tutti i metodi e le proprietà della classe padre. Per questo motivo invochiamo il costruttore della classe padre in modo da inizializzare le proprietà di questa.

Vediamo un esempio di utilizzo:

```
<?php
$auto = new Automobile(“BMW”);
$auto->setColore(“Giallo”);
$auto->setCarburante(“Benzina verde”);
$auto->setModello(“730Ci”);
$auto->setCilindrata(3000);
$auto->setNumeroCilindri(8);
print “La mia “ . $auto->getNumeroDiRuote() .
      “ ruote dei sogni è una “ .
      $auto->getMarca() . “ “ . $auto->getModello() .
      “ di colore “ . $auto->getColore();
?>
```

Come vediamo dall'esempio abbiamo potuto usare i metodi definiti nella classe Veicolo con l'oggetto della classe Automobile, in quanto ereditate da questa. Inoltre, la proprietà NumeroDiRuote è stata valorizzata nel costruttore della classe figlia. La marca dell'auto è passata alla classe attraverso il parametro del costruttore.

Il metodo costruttore non è obbligatorio. In una classe derivata, comunque, se il costruttore non esiste verrà automaticamente chiamato il costruttore della classe padre (se esiste).

In una classe estesa è possibile creare un metodo con lo stesso nome di un metodo della classe base (o classe padre, o superclasse). In questo caso, quando il metodo sarà invocato, verrà eseguito il codice all'interno del metodo della classe derivata.

6.4 - Interazione con le FORM HTML

Come sappiamo, l'elemento FORM in HTML permette di inserire dei dati da parte dell'utente e spedirli tramite metodo HTTP GET o POST al server, dove uno script dedicato (specificato tramite l'attributo ACTION dell'elemento FORM) provvederà a processare i dati inseriti.

Una volta che lo script PHP riceverà i dati della form, il loro valore si troverà nell'array associativo superglobale $_GET o $_POST a seconda che la form sia stata sottomessa tramite, rispettivamente, metodo GET o POST. E' comunque possibile, se non è dato sapere a priori con quale metodo la form è stata sottomessa, utilizzare l'array associativo superglobale $_REQUEST.
Le chiavi di questi array superglobali sarà il nome del campo della form.
Vediamo un esempio, composto da una form HTML e dal relativo script di gestione.

```
[...]
<form action="myform.php" method="POST">
   <p><label>Inserisci il tuo nome</label>
   <input type="text" name="nome">
   </p>
   <p><label>Inserisci il tuo cognome</label>
   <input type="text" name="cognome">
   </p>
   <p>
   <input type="submit" value="invia">
   </p>
</form>
[...]
```

Lo script che gestisce la form, myform.php, sarà il seguente:

```
<?php
   $nome = $_POST["nome"];
   $cognome = $_POST["cognome"];
   print "<p>Salve, $nome $cognome !</p>";
?>
```

6.4.1 - Nota sui valori dei campi in input

Quando si debbano stampare dei valori che dovranno essere parte dell'attributo

VALUE di campi di input di una FORM, occorre premunirsi che questi non contengano dei caratteri che potrebbero produrre errori sintattici all'interno della form stessa.

Si prenda ad esempio il frammento di codice seguente, nel quale viene predisposto il valore per un campo di tipo hidden di una form:

```
print "<input type=\"hidden\" name=\"dati\"
value=\"$dati\">";
```

Se la variabile "$dati" contenesse la seguente stringa:

```
$dati = 'Qualcosa di "speciale"';
```

il codice HTML risultante sarebbe il seguente:

```
<input type="hidden" name="dati" value="Qualcosa si
"speciale"">
```

che, dal punto di vista di HTML sarebbe sicuramente scorretto.
Per rpeventivarsi da questo genere di problemi, occorre utilizzare la funzione htmlspecialchars(), che converte se necessario i caratteri all'interno di una stringa con le relative character entities.
Riscriviamo dunque l'esempio in maniera corretta:

```
print "<input type=\"hidden\" name=\"dati\"
      value=\"" . htmlspecialchars($dati) . "\">";
```

In questo modo, fermo restando il valore dato precedentemente alla variabile dati, l'HTML prodotto sarà:

```
<input type="hidden" name="dati" value="Qualcosa si
&amp;quot;speciale&amp;quot;">
```

che è corretto.

6.4.2 - URL e FORM encoding

Quando dei dati devono essere trasmessi verso un server, sia tramite GET che tramite POST, questi devono essere codificati in una maniera (esposta nel RFC1738) che permette il transito corretto di tutte le informazioni.

Secondo questa codifica, tutti i caratteri che non siano strettamente alfanumerici (eccezion fatta per i due caratteri "_" e "-") sono trasformati in un numero esadecimale preceduto dal carattere "%". Il carattere "spazio" viene trasformato in "+".

Per quanto riguarda le FORM, questa trasformazione è effettuata automaticamente dal browser che, prima di effettuare la trasmissione dei dati, li trascodifica secondo questa regola.

Gli URL invece, quando contengano dati da codificare, devono essere gestiti dal programmatore.

Il PHP per effettuare questa operazione mette a disposizione la funzione urlencode() (ed esiste anche una complementare funzione urldecode()).

Un esempio ci aiuterà a chiarirsi le idee:

```
<?php
$valoreFoo = "Stringa non codificata";
// Pericolo !, le informazioni non sono codificate
print '<a href="myscript.php?foo=$valoreFoo">';
// Giusto, usando la urlencode()
print '<a href="myscript.php?foo=" .
urlencode($valoreFoo") . '>';
// Ancora meglio, usando anche htmlspecialchars()
print '<a href="myscript.php?foo=" .
htmlspecialchars(urlencode($valoreFoo")) . '>';
?>
```

Al posto di htmlspecialchars() è possibile usare anche htmlentities() che è pressoché identica se non che *tutti* i caratteri trasformabili in entities lo saranno. htmlspecialchars, invece, trasforma soltanto in entities i caratteri che potrebbero recare confusione nella sintassi HTML (come “, <, > e &);

6.4.3 - Upload dei file

Come è risaputo, tramite FORM HTML è possibile anche effettuare l'upload di un file da un client al server.

In questo capitolo vediamo come si effettua questa operazione tramite l'ausilio del PHP.

Prima di tutto è necessario creare una FORM HTML scritta nella maniera corretta. Dallo studio dell'HTML abbiamo visto che quando una form deve trasmettere, oltre che a dati utente, anche un file deve allora avere impostato un particolare tipo di *encoding* (multipart/form-data).

Il primo campo che deve essere inserito nella form deve poi essere un campo input di tipo hidden dal nome “MAX_FILE_SIZE”. Successivamente tra i campi della form deve dunque essere inserito un input di tipo “FILE”.

```
<form enctype="multipart/form-data" action="upfile.php"
method="POST">
<input type="hidden" name="MAX_FILE_SIZE"
value="30000" />
Upload del file: <input name="userfile" type="file" />
<input type="submit" value="Send File" />
</form>
```

A questo punto il nostro utente è pronto per effettuare l'upload di un file, di una dimensione massimo di circa 30KB. Occorre notare subito che la dimensione massima che il PHP accetterà per un file è anche indicata all'interno del file di configurazione php.ini e può essere modificata a piacere (il default è 2MB).

Per gestire l'upload lato server, ovvero dallo script upfile.php, sarà necessario

lavorare con l'array associativo multidimensionale superglobale (waw!) $_FILES.

Una volta che il file sarà stato trasmesso dal browser al nostro server, questo si troverà in una directory temporanea con un nome creato da PHP (la directory temporanea è un dato di configurazione modificabile). Fintanto che non avremo deciso (nello script upfile.php) cosa fare del file che ci è stato trasmesso (salvarlo da qualche parte ? Inserirlo nel database ? Cancellarlo ?) questo rimarrà parcheggiato nella directory temporanea.

Per ogni file trasmesso sarà possibile accedere, tramite la variabile $_FILES, alle seguenti relative informazioni:

- il nome del file, così come si chiamava sul computer dell'utente (name)
- la sua dimensione (size)
- il suo MIME type (è un file di testo ? Un'immagine jpeg ? Un PDF ?) (type)
- il nome temporaneo creato dal PHP (tmpname)
- un eventuale codice di errore (error)

Con queste informazioni e con le apposite funzioni PHP predisposte a lavorare sui file trasmessi, vediamo come gestire questo upload:

```
<?php
if ($_FILES['userfile']['size'] > 30000)
{
   print "File troppo grande!\n";
   exit;
}

// Directory dove copiare il file caricato
$uploaddir = '/var/www/uploads/';

// Nome del file, completo di path
$uploadfile = $uploaddir .
```

```
                basename($_FILES['userfile']['name']);

// Tentiamo di spostare il file dalla directory
temporanea
// alla directory predisposta
if (move_uploaded_file($_FILES['userfile']['tmp_name'],
$uploadfile))
{
   print "Il file è stato caricato con successo!.\n";
}
else
{
   echo "File non caricato.\n";
}

?>
```

Non appena lo script terminerà senza aver effettuato la copia nella directory opportuna (è il caso nel quale il file risultasse troppo grande), il file nella directory temporanea verrebbe cancellato automaticamente dal PHP.

6.5 - Accesso al file system

Il file system è ben gestito dal PHP, le funzioni a disposizione degli sviluppatori sono innumerevoli, inoltre sono ben integrate con il sistema operativo in modo particolare GNU/Linux.

Ad esempio è possibile ottenere e cambiare i permessi, il proprietario e il gruppo di un file, creare collegamenti simbolici, copiare, spostare, eliminare directory e file. Dunque gli strumenti a disposizione non mancano e molti di essi verranno trattati nelle pagine seguenti.

Prima di entrare nello specifico è bene precisare ancora una volta che il file system su cui il PHP opera è quello dell'elaboratore server, quindi se si crea un file esso viene creato sul disco del server e non su quello del cliente.

6.5.1 - Apertura di un file

Se, metaforicamente, un file è considerato come un *contenitore* di informazioni, per poter accedere al suo contenuto è necessario **aprirlo**. Per farlo si utilizza la funzione fopen() a cui va fornito il file con il percorso, se diverso da quello dello script, e il tipo di apertura. Al momento dell'apertura del file occorre dunque dichiarare il tipo di operazione che si desidera eseguire su di esso, ad esempio se si vuole solo leggere, scrivere o semplicemente aggiungere dati alla fine.

Per ogni file aperto il PHP tiene in memoria un *puntatore* che serve a sapere, finché il file è aperto, a che punto dello stesso ci si trovi (all'inizio ? Alla fine ? Al byte 12314839 ?)

Ecco l'elenco dei possibili modi di accedere ad un file in PHP:

- 'r' Apre in sola lettura; posiziona il suo puntatore all'inizio del file;
- 'r+' Apre in lettura e scrittura; posiziona il suo puntatore all'inizio del file;
- 'w' Apre in sola scrittura; posiziona il suo puntatore all'inizio del file e ne elimina il contenuto. Se il file non esiste lo crea;
- 'w+' Apre in lettura e scrittura; posiziona il suo puntatore all'inizio del file e ne elimina il contenuto. Se il file non esiste lo crea;
- 'a' Apre in sola scrittura (append); posiziona il suo puntatore alla fine del file. Se il file non esiste lo crea;
- 'a+' Apre in lettura e scrittura; posiziona il suo puntatore alla fine del file. Se il file non esiste lo crea;

> Il PHP permette anche di aprire file remoti tramite i protocolli più diffusi come HTTP e FTP, ovviamente il file deve essere raggiungibile almeno in lettura.

Ecco un paio di esempi:

```
<?php
// Apriamo il file “dati.txt” per un accesso in
```

```
scrittura:
$f = fopen ("dati.txt", "w");
// Apriamo il file “dati.txt”, che si trova nel percorso
// specificato, in modalità “append”
$f = fopen ("/home/qualche_percorso/dati.txt", "a"
// Resto dello script...
?>
```

Il valore $f, detto *file handle* è un numero intero assegnato dal sistema operativo che ci permetterà di riferirci a questo particolare file nelle operazioni successive.

> Occorre fare molta attenzione ai permessi dei file che si intende manipolare e delle directory in cui sono contenuti. Ricordare che l'utente che accede al file è quello che lancia il server HTTP e di conseguenza l'interprete PHP in esso caricato come modulo. Tale utente viene specificato nel file di configurazione del server.

Ogni volta che abbiamo terminato di lavorare con il file aperto, questo va *chiuso*. La funzione per chiudere un file è close() che accetta come parametro (obbligatorio) il file handle del file aperto.
E' importante chiudere un file appena non abbiamo più bisogno dello stesso: ogni file aperto impegna risorse del sistema, delle quali è sempre buona norma fare un uso oculato. Secondo come il file viene aperto, inoltre, possono verificarsi problemi di concorrenza: tentare di aprire un file quando qualcun altro lo ha già aperto in maniera *esclusiva* ovvero per poterci scrivere, può risultare in un errore difficile da gestire. Appena possibile, dunque, chiudere il file.

6.5.2 - Lettura di un file di testo

Il primo esempio che ci troveremo a discutere sarà quello di leggere un semplice file di testo esistente e contenuto nella stessa directory dello script. Verrà utilizzato il metodo di sola lettura con l'indispensabile controllo di errore e ancora prima di esistenza del file da leggere. Se si salvano informazioni riservate in file di testo è bene farlo in una directory che non faccia parte dei documenti WEB

altrimenti potrebbe essere richiamato e letto direttamente dal browser. Inoltre è importante nascondere eventuali errori mediante una corretta gestione degli stessi al fine di evitare la visualizzazione di informazioni importanti come il percorso e il nome del file che si tenta di aprire. Prima di scrivere il codice è bene creare il file da leggere, nominarlo come dati_1.txt e riempirlo con del semplice testo.

```
prima riga del file di prova
seconda riga del file di prova
terza riga del file di prova
ultima riga del file di prova
```

Ecco il programma che legge il file appena creato:

```
<?
if (!file_exists("dati_1.txt"))
{
   print "Il file dati_1.txt non esiste.";
   exit;
}
// apriamo il file in sola lettura
$f = @fopen("dati_1.txt", "r");

// Se il file è stato aperto con successo, si
// procede alla lettura e alla stampa del suo contenuto
if ($f)
{
   // Finché non sia stata raggiunta la fine del file...
   while(!feof($f))
   {
        // Legge una riga (di max 4kb)
        $riga = fgets($f,4096);
        print "<br>". $riga;
   }
```

```
   // Alla fine della lettura, chiudiamo il file
   fclose($f);
}
else
{
   print "Errore durante l'apertura del file!";
}
?>
```

6.5.3 - Scrittura di un file di testo

Nell'esempio che segue, scriveremo all'interno di un file l'indirizzo IP dell'ultimo visitatore che ha caricato la pagina.

```
<?
// Apriamo il file. Se il file esiste, viene svuotato.
// Se non esiste, lo crea
$f = fopen("dati_3.txt", "w");
if ($f)
{
   print  "<p>File aperto correttamente</p>";
   print "<p>Sto salvando i tuoi dati nel file</p>";
   $frase = "Ultimo visitatore ad aver eseguito " .
            "lo script\n\n";
   $frase = $frase. "il suo IP:" .
             $_SERVER["REMOTE_ADDR"]. "\n";

   // Scrittura nel file
   fputs($f,$frase);
   fclose($f);
   print "Operazione terminata";
}
else
```

```
{
   print <p>Errore in apertura del file dati_3.txt\n";
}
?>
```

Si lascia come esercizio per lo studente, la scrittura di uno script che permetta di salvare sul file tutti gli accessi alla pagina, e non soltanto l'ultimo. Utilizzare a questo proposito il metodo di apertura file “a”, tramite il quale il file viene aperto in scrittura ma non cancellato e il suo puntatore viene spostato alla fine del file.

6.6 - I cookie

La gestione dei cookie (letteralmente *biscotti*) è il metodo per memorizzare, sul computer dei nostri utenti, delle informazioni che vogliamo persistano anche nelle successive visite al nostro sito.

I cookie sono utilizzati per memorizzare piccole quantità di dati come ad esempio il nome dell'utente o una serie di preferenze.

Tramite i cookie è possibile, dalla nostra applicazione Web, scoprire se l'utente è la prima volta che si connette al nostro sito oppure no, la data della sua ultima visita oppure mantenere dati più complessi come possono essere le informazioni riguardanti il suo carrello (virtuale) della spesa in un sito di commercio elettronico.

Essi forniscono una metodologia per superare un “limite” del protocollo HTTP, ovvero quello di essere *stateless*. Ogni volta che visitiamo un sito, ogni volta che richiediamo l'ennesima pagina di un sito, per quel sito non saremmo comunque che alla nostra prima visita. Il sito non ha un modo, nativo nel protocollo HTTP, per dire “questa utente è già stato qui” né tantomeno di salvare informazioni tra una visita e l'altra.

I cookie sono registrati dal browser in una specifica directory in formato di file di testo (un singolo file per cookie, o un unico file con tutti i cookie, a seconda del browser). Tutte le volte che deve richiedere una pagina ad un sito, questo controlla prima se quel sito ha qualche cookie registrato. In caso affermativo,

all'interno della richiesta HTTP esso provvederà ad aggiungere anche i dati dei cookie, mettendoli dunque a disposizione dello script sul server.

6.6.1 - Cookie e sicurezza

I cookie non sono adatti per registrare informazioni critiche come password o dati personali in quanto potrebbero crearsi dei problemi di sicurezza. Teniamo infatti presente che questi dati verranno salvati in chiaro (non criptati) in un file sul computer dell'utente. Se questo computer fosse in qualche modo aperto agli occhi di più persone (una stazione di lavoro condivisa, oppure un computer compromesso) sarebbe possibile da chiunque accedere a quelle informazioni.

Spesso i cookie sono accusati di essere uno strumento che, in mano a persone di pochi scrupoli, possa in qualche modo violare la privacy dell'utente. Un uso poco accorto degli stessi (ovvero la registrazione al loro interno di dati sensibili) unito a sistemi o a bachi del browser che permettano di leggere, da parte di un sito male intenzionato, i dati al loro interno fanno sì che spesso siano malvisti.

Occorre però tenere presente che, quando usati con giudizio, essi sono uno strumento ormai praticamente indispensabile per una gestione semplice di un sito dinamico e, soprattutto, di per sé innocui.

Fatti salvi gli eventuali bachi del browser, il modello di sicurezza dei cookie prevede comunque che le informazioni al loro interno inserite dal sito "www.example.com" possano essere letti e scritti solo da quel sito e non da altri.

Ad ogni cookie è infatti associato uno *dominio* e un *percorso* (path). Tali informazioni vengono utilizzate dal browser per stabilire quali siano i cookies validi per il sito in cui si sta navigando. Ovviamente la motivazione di questo meccanismo risiede nella necessità di impedire che gli script di un sito possano in qualsiasi modo manipolare i cookies di un sito diverso. I valori di default sono, rispettivamente, il nome di dominio ed il percorso corrente.

Di default il dominio viene impostato al sito che ha settato il cookie stesso, ma talvolta questo comportamento non è sufficiente. Se per esempio un cookie venisse impostato su "www.mionegozio.it", a un successivo accesso a

“shop.mionegozio.it” il browser non spedirebbe il cookie al server. In questo caso, impostando il dominio del cookie a “mionegozio.it” risolverebbe il problema.

Infine, ogni cookie ha un attributo *secure* che, se impostato, lo contrassegna come riservato, per cui esso potrà essere trasmesso solo attraverso connessioni sicure (in particolare connessioni HTTPS). Per default tale attributo non è impostato.

6.6.2 - Biscotti con la data di scadenza

Un'altra caratteristica del cookie è quella di avere un “data di scadenza” *(expiration date*), passata la quale il browser prevede a cancellarlo. Se la data di scadenza non viene impostata, un cookie attivato durante una sessione di navigazione sul nostro sito verrà cancellato appena si chiude il browser.

PHP permette ovviamente di gestire in maniera completa il salvataggio di informazioni all'interno di cookie.

La data di scadenza deve essere impostata in formato di *Unix timestamp*. In PHP la funzione time() provvede tale valore, ritornando il numero di secondi passati dal 1 gennaio 1970 (convenzionalmente intesa come inizio dell'*epoca Unix*) alla data odierna.

6.6.3 - Cookie e PHP

Salvare le informazioni all'interno di un cookie, significa semplicemente registrare una coppia NOME/VALORE in maniera del tutto simile ad una normale gestione di variabili.

L'uso dei cookies in PHP è estremamente semplice anche perché avviene mediante un'unica funzione, `setcookie()`. Occorre precisare subito che si tratta di una funzione con una particolarità: non può essere utilizzata in un punto qualsiasi di uno script ma solo nella sua parte iniziale; più precisamente, setcookie() può essere chiamata esclusivamente prima che una qualsiasi porzione di pagina sia inviata al client (tipicamente, prima del tag <HTML>).

Questo avviene perché la gestione dei cookies è realizzata a livello di intestazioni (headers) HTTP che, secondo quanto stabilito da tale protocollo, costituiscono (e devono costituire!) la parte iniziale del flusso di dati di una connessione. Se non si

rispetta tale regola le intestazioni HTTP si confonderanno con il contenuto della pagina e non produrranno il comportamento atteso (nel nostro caso specifico non verranno impostati i cookies desiderati).

Per il resto, una volta impostato, i dati all'interno di un cookie è accessibile da PHP come una variabile all'interno dell'array associativo $_COOKIES.

L'esempio successivo illustra come poter scorrere tutti i dati presenti all'interno dei cookie:

```
<?php
foreach ($_COOKIES as $key => $val)
{
   print "\"$key\" contiene \"$val\"<br>";
}
?>
```

Vediamo adesso nel dettaglio l'utilizzo della funzione setcookie() per inserire, aggiornare e rimuovere cookie.

6.6.4 - Un nuovo cookie

Per salvare un cookie la sintassi è la seguente:

```
setcookie([nome stringa],
        [valore stringa],
        [scadenza UNIX time stamp],
        [percorso stringa],
        [dominio stringa],
        [accesso sicuro intero(1/0)])
```

Esempio:

```
<?php
// Si imposta un cookie per salvare il valore di
```

```
“nomeUtente”
// Non impostando la data di scadenza, il cookie scadrà
con la chiusura del browser
setcookie(“nomeUtente”, “Mario”);
// Si imposta un altro cookie. Stavolta impostiamo la
scadenza ad un'ora (3600 secondi)
setcookie(“nomeUtente”, “Mario”, time() + 3600);
// Si imposta un altro cookie. Il cookie verrà spedito al
nostro sito e anche a “www.friendsite.com” ma solo su
connessioni sicure.
setcookie(“nomeUtente”, “Mario”, time() + 3600, “/”,
“friendsite.com”, 1);
?>
```

6.6.5 - Modificare un cookie

Per modificare un cookie, occorre reimpostarlo (sempre con setcookie()).

Esempio

```
<?php
// Leggiamo il valore di un cookie, un ipotetico
// contatore di accessi
$contatore = $_COOKIES[“contatore”];
$contatore++;
// Salviamo di nuovo il cookie
setcookie(“contatore”, $contatore);
?>
```

6.6.6 - Cancellare un cookie

Per cancellare un cookie è necessario reimpostarlo con una data di scadenza nel... passato.

Esempio:

```
<?php
// Cancelliamo un cookie, impostandone la data di
// scadenza nel passato (un'ora fa)
setcookie("contatore", "", time() - 3600);
?>
```

6.6.7 - Limiti nell'uso dei cookie

I cookie, sebbene siano uno strumento semplice, efficace e universalmente supportato soffrono di gravi limiti, che ci impongono di cercare altre soluzioni.

Tali limiti possono essere così riassunti:

- Ai browser è richiesto di mantenere solo 300 cookie in totale (il fatto che ne mantengano di più è comunque relativo alla loro particolare implementazione)
- Ai browser è richiesto di non mantenere più di 20 cookie per dominio
- I cookie non possono contenere più di 4KB di dati
- I cookie potrebbero essere stati disabilitati dall'utente.

6.7 - Le sessioni

Come abbiamo visto i cookie, nati per superare un limite del protocollo HTTP, sono uno strumento semplice ma anch'esso limitato da molti fattori.

Per ottenere un risultato simile (ovvero simulare una sorta di connessione *statefull* tra client e server) ma senza i limiti dei cookie, lo strumento di elezione sono le sessioni.

Il concetto alla base delle sessioni è il seguente:

tutte le volte che un utente, tramite browser, effettua una visita ad un sito questo assegna a tale utente un *id di sessione*. Durante tutta la navigazione, l'utente si

porterà dietro questo ID come identificativo univoco della sessione.

Utilizzando questo ID, il server (nella fattispecie il *motore* PHP), registrerà remotamente (e dunque non sul client, come per i cookie) tutte le informazioni necessarie (informazioni, queste, con lo stesso significato funzionale di quelle registrate con i cookie).

Possiamo pensare a questo meccanismo come una sorta di *database* dove il server registra le informazioni utilizzando come chiave di accesso a queste informazioni proprio quell'id univoco.

Lato server, i dati sono registrati in molti modi possibili (su file, su database, direttamente in memoria, etc...) ma qualunque sia il metodo scelto questo sarà del tutto trasparente sia allo sviluppatore del sito che soprattutto al nostro utente.

Ma se questo ID è univoco e rimane insieme all'utente per tutta la durata della navigazione, questo dove viene registrato ? Come viene gestito ?

Anche in questo caso i metodi sono diversi. Se l'utente ha i cookie abilitati, allora questo ID viene salvato in un cookie (con il nome di PHPSESSID, un default che può comunque essere modificato tramite la funzione session_name()), altrimenti PHP (automaticamente o manualmente a scelta del programmatore) può usare un altro sistema come quello di aggiungere la coppia PHPSESSID/valore ad ogni URL richiesta o addiruttua ad ogni FORM (come campo hidden).

6.7.1 - Usare le sessioni

La gestione delle sessioni è dunque praticamente trasparente, da un punto di vista della programmazione (anche se varie accortezze e personalizzazioni possono essere adottate per esigenze molto specifiche).

Tutto quello che serve è eseguire la funzione `session_start()` all'inizio di ogni script che faccia uso delle variabili di sessione. Anche in questo caso, come per la funzione setcookie(), la funzione session_start() deve essere eseguita prima di qualsiasi altro output generato dallo script.

La funzione session_start() provvederà a creare una nuova sessione o a ripristinare

quella precedente, leggendo i dati dal server e renderndoli disponibili allo script PHP.

A questo punto lo script ha a disposizione la variabile $_SESSION (un array associativo superglobale) in lettura e scrittura. I valori inseriti all'interno dell'array verrano inseriti in sessione, salvati sul server, e resi a disposizione del successivo script che invocasse la session_start().

Vediamo un esempio, composto da due script che si passano una variabile tramite l'utilizzo della sessione:

```
<?php
// a.php
// Valorizza una variabile di sessione
session_start();
$_SESSION[“nomeUtente”, “Franco”];
// Salta allo script b.php
header(“Location: b.php”);
?>

<?php
// b.php
// Visualizza il contenuto della variabile di sessione
nomeUtente
session_start();
print “Ill nome dell'utente è “ .
$_SESSION[“nomeUtente”];
?>
```

6.7.2 - Distruggere una sessione

Quando non avessimo più necessità di una sessione, allora occorre procedere alla sua distruzione. Questa operazione viene compiuta in due o tre parti. Prima si cancellano tutte le variabili di sessione, poi si distruggono tutti i dati ad essa

collegati ed infine si cancella (l'eventuale) cookie utilizzato per memorizzare il session id.

```
 <?php
// Sessione da distruggere
session_start();

// Si imposta l'array $_SESSION ad un array vuoto,
// di fatto cancellandone il contenuto
$_SESSION = array();

// Se esiste un cookie di sessione (contenente il
// nome della sessione), lo si fa cancellare
if (isset($_COOKIE[session_name()]))
{
   setcookie(session_name(), '', time()-42000, '/');
}

// Infine distruggiamo quanto è rimasto della sessione:
session_destroy();
?>
```

6.8 - La funzione header()

Particolare importanza ha la funzione di libreria standard `header()` che permette di aggiungere, oltre a quelli predefiniti, header addizionali alla risposta HTTP da rendere al browser insieme (o in sostituzione) del risultato dello script.

Due degli utilizzi più comuni sono:

- utilizzare l'header “Location:” per effettuare un redirect ad un'altra pagina. Una sorta di goto... HTTP.
- Utilizzare l'header “Content-Disposition:” per impostare il nome del file quando questo sia generato “al volo” da uno script PHP (ad esempio, un PDF).

6.9 - La funzione die()

La funzione die() è utile quando si voglia, con una sola istruzione terminare lo script corrente e visualizzare un messaggio di errore. Il suto uso è mostrato nel seguente esempio:

```
<?php
if (!($fh = fopen("r", "filedati.txt")))
   die ("Impossibile aprire il file !");
?>
```