

PHP - il linguaggio

Principali argomenti:

- Sintassi e semantica dei costrutti e dei dati di base
- Gestione dei form Web
- Gestione dei file
- Cookies e sessioni HTTP
- Altre funzionalità e librerie
- Oggetti
- Interazione con database

Per fonti e trattazione, v. prossime slide

Il codice di esempio in queste slide è su Teams in *php_examples.zip*

NB: slide e paragrafi con sfondo colorato così possono essere saltate ai fini della preparazione per l'esame

Alcune risorse online

- <https://www.w3schools.com/php> (seguito per le lezioni)
- https://www.w3schools.com/php/php_oop_what_is.asp
- <https://developer.hyvor.com/tutorials/php>
- <https://www.phptutorial.net> (completo e accurato)
- <https://www.php.net/manual/> anche in italiano: <https://www.php.net/manual/it/>
- <https://www.php.net/manual/language.oop5.php> (su oggetti PHP 5)
- <https://www.tutorialspoint.com/php>
- https://www.tutorialspoint.com/php/php_object_oriented.htm
- https://web.archive.org/web/20230408174804mp_/https://www3.ntu.edu.sg/home/ehchua/programming/index.html#php, tutorial conciso ma ricco: [setting up](#), [basics](#), [PHP/MySQL Webapps](#), [OOP in PHP](#), [PHP Miscellaneous](#), [PHP Unit Testing](#)
- <https://www.html.it/guide/guida-php-di-base/>
- <https://www.youtube.com/playlist?list=PL101314D973955661> (fcamusu su youtube)
- <https://laracasts.com/series/php-for-beginners-2023-edition> (video, molto chiaro)

PHP, HTML, HTTP

- **HTML** e **HTTP** sono la fondazione su cui è costruito tutto il Web
- HTTP viene presentato nel corso di Reti
- Le **basi** di HTML fanno certamente parte delle competenze di uno studente di Informatica
- **HTML avanzato** e **Javascript**: v. il corso di *Web programming*
- Vedremo solo alcuni approfondimenti utili ai fini del corso
- Principali **riferimenti** (non tutorial!) per approfondire in autonomia:
 - Mozilla Developer Network (eccellente, completo ma assai più conciso degli standard): <https://developer.mozilla.org>
 - <https://www.w3.org/TR/html4/> (HTML 4, dal consorzio W3C)
 - <https://whatwg.org> (HTML 5, il W3C ha delegato al WhatWG l'evoluzione della specifica di HTML 5 come “living standard”)

PHP: il tutorial w3schools

- Ottimo il tutorial <https://www.w3schools.com/php>
- Non più in italiano (ma era solo Google Translate)
- Lo utilizzeremo nel corso delle lezioni
- Si consiglia di studiare il linguaggio leggendo il tutorial
- Mette a disposizione box interattivi per eseguire codice PHP e visualizzarne il risultato
- Queste slide intendono **soltanto** illustrare alcuni punti integrativi rispetto al tutorial, o meritevoli di essere evidenziati, o approfonditi
- Si raccomanda di studiare, col relativo codice, gli esempi del tutorial per gli argomenti evidenziati nella mappa [qui](#) avanti
- Verranno proposti degli esempi aggiuntivi (v. Teams)

Il tutorial w3schools: materiale svolto

Dagli indici di <https://www.w3schools.com/php> riportiamo gli argomenti trattati, evidenziando quelli discussi per cenni o in seminario (extra syllabus)

PHP Tutorial

- PHP HOME
- PHP Intro
- PHP Install
- PHP Syntax
- PHP Comments
- PHP Variables
- PHP Echo / Print
- PHP Data Types
- PHP Strings
- PHP Numbers
- PHP Math
- PHP Constants
- PHP Operators
- PHP If...Else...Elseif
- PHP Switch
- PHP Loops
- PHP Functions
- PHP Arrays
- PHP Superglobals
- PHP RegEx

PHP Forms

- PHP Form Handling
- PHP Form Validation
- PHP Form Required
- PHP Form URL/E-mail
- PHP Form Complete

PHP Advanced

- PHP Date and Time
- PHP Include
- PHP File Handling
- PHP File Open/Read
- PHP File Create/Write
- PHP File Upload
- PHP Cookies
- PHP Sessions
- PHP Filters
- PHP Filters Advanced
- PHP Callback Functions
- PHP JSON
- PHP Exceptions

PHP OOP

- PHP What is OOP
- PHP Classes/Objects
- PHP Constructor
- PHP Destructor
- PHP Access Modifiers
- PHP Inheritance
- PHP Constants
- PHP Abstract Classes
- PHP Interfaces
- PHP Traits
- PHP Static Methods
- PHP Static Properties
- PHP Namespaces
- PHP Iterables

MySQL Database

- MySQL Database
- MySQL Connect
- MySQL Create DB
- MySQL Create Table
- MySQL Insert Data
- MySQL Get Last ID
- MySQL Insert Multiple
- MySQL Prepared
- MySQL Select Data
- MySQL Where
- MySQL Order By
- MySQL Delete Data
- MySQL Update Data
- MySQL Limit Data

Ci si concentri su questi argomenti, nonché sugli spunti nelle slide a **sfondo bianco** (non grigioazzurro)

La funzione *var_dump()*

Mostra in output (*dump*) struttura e contenuto di uno o più dati, in particolare *tipo* e *valore*; il prototipo (da <https://www.php.net/manual/>):

```
var_dump ( mixed $value , mixed ... $values ) : void
```

L'argomento *\$value* è (il valore di) una qualsiasi espressione PHP, specie (ma non solo) una *variabile* (ingannevole il nome *var_dump*)

Se *\$value* è di tipo strutturato (array o oggetto), verrà mostrato ricorsivamente, esplorandone la struttura (box a destra)

Esempi (con la REPL *php -a*):

```
php> var_dump(3+2);  
int(5)  
php> var_dump(true);  
bool(true)  
php> var_dump(3.14);  
float(3.14)  
php> $n = 10;  
php> var_dump($n);  
int(10)
```

```
php> var_dump([1, 3.14, 'ab']);  
array(3) {  
    [0]=>  
    int(1)  
    [1]=>  
    float(3.14)  
    [2]=>  
    string(2) "ab"  
}
```

Il tipo *int*

Il tipo *int* è rappresentato in complemento a 2, con i bit previsti dalla piattaforma su cui si esegue l'engine PHP

Malgrado ciò che dice il tutorial *w3schools*, tipicamente, oggi, *int* ha dimensione 64 bit (8 byte), per cui il range di *int* sarà $-2^{63} \dots 2^{63}-1$

Verifichiamolo, grazie alle *costanti predefinite* previste dallo standard PHP:

```
php> echo PHP_INT_SIZE;
8
php> echo PHP_INT_MIN . " .. " . PHP_INT_MAX;    # sotto si vede che PHP_INT_MAX è  $2^{63}-1$ , mentre
-9223372036854775808 .. 9223372036854775807      # PHP_INT_MIN vale  $-2^{63}$ 
php> printf("%x\n", PHP_INT_MAX);                # printf(), come in C, può mostrare il formato hex ("%x")
7fffffffffffffff                                # vediamo 16 cifre hex, quindi 64 bit
```

NB: $\text{PHP_INT_MAX} = 0x7\text{ffffffffffffffff}$ è $0x8000000000000000 - 1$,
e $0x8000000000000000$, cioè $0x8$ seguito da 15 zeri, è appunto 2^{63}

Precisione numerica: *int*

La disponibilità di un numero finito di bit per il tipo *int* inficia la precisione dei calcoli, se questi richiedono più di 64b per il risultato finale o intermedio

```
php> echo 2**62 - 1 + 2**62; # l'espressione vale 263-1, cioè PHP_INT_MAX, che sta in 64 bit, e vale...  
9223372036854775807      # il risultato è corretto, sta in 64 bit, come il risultato intermedio 262-1
```

Ora, in matematica $2^{62}-1+2^{62} = 2^{62}+2^{62}-1 = 2^{63}-1$, ma nel calcolatore...

```
php> echo 2**63-1;  
9.223372036854776E+18  
php> var_dump(2**63-1);  
float(9.223372036854776E+18)
```

$2^{63}-1$ è calcolato valutando prima 2^{63} , il cui valore, non rappresentabile come *int* a 64 bit, viene "promosso" a *float*, il che rende *float* anche il valore finale di $2^{63}-1$!

```
php> var_dump(2**62+2**62-1);  
float(9.223372036854776E+18)
```

Lo stesso accade nel calcolo di $2^{62}+2^{62}-1$

Inoltre, i valori *float* delle espressioni 2^{63} e $2^{63}-1$ coincidono (v. qui a destra), per via della (im)precisione dei calcoli *float* a 64 bit, cf. <https://www.php.net/manual/en/language.types.float.php>

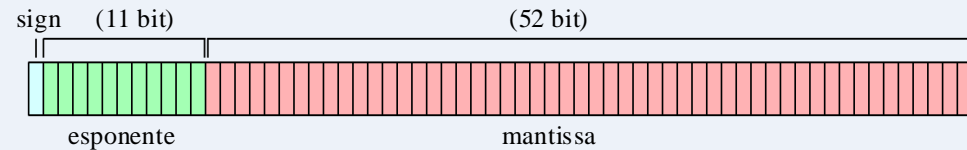
```
php> var_dump(2**63);  
float(9.223372036854776E+18)  
php> var_dump(2**63-1);  
float(9.223372036854776E+18)
```

In sostanza, cioè, la rappresentazione *float* di 2^{63} ha una *precisione* (ridotta, di 15 cifre decimali), non sensibile al -1 (servirebbero 19 cifre)

Precisione numerica: *float*

Cf. <https://www.php.net/manual/en/language.types.float.php> e lo standard IEEE 754

L'imprecisione nei calcoli, rispetto alla matematica "ideale" e in base 10, ha due cause: (a) rappresentazione interna binaria e calcoli in base 2 (non 10) e (b) tale rappresentazione, qui in fig., ha numero di bit finito (64)



Ne segue che la precisione nei calcoli, espressa dalla costante PHP_FLOAT_DIG, è "solo" di 15 cifre decimali significative.

La precisione numerica è comunque argomento assai complesso

Qui (a destra) del codice esemplificativo dei problemi accennati

```
php> echo PHP_FLOAT_DIG;
15

php> $x = 12345678901234567E-17; # mantissa di 17 cifre, ma precisione 15
# per questo, sommare ripetutamente 1E-17, cioè (in teoria) 1 alla mantissa, non
# ha l'effetto "teorico", che ci si aspetterebbe, come illustrato dal codice qui sotto

php> for ($n=0;$n<94;$n++) {echo "$n: "; var_dump($x); $x+=1E-17;}
0: float(0.12345678901234566) # atteso: 0.12345678901234567
1: float(0.12345678901234568) # atteso: 0.12345678901234568
2: float(0.12345678901234569) # atteso: 0.12345678901234569
3: float(0.1234567890123457) # atteso: 0.1234567890123457
4: float(0.12345678901234572) # atteso: 0.12345678901234571
5: float(0.12345678901234573) # atteso: 0.12345678901234572
6: float(0.12345678901234575) # atteso: 0.12345678901234573
...
93: float(0.12345678901234695) # atteso: 0.12345678901234660
```

Precisione numerica: *float* / 2

L'esempio precedente illustra la perdita di precisione che si verifica quando le cifre significative desiderate sono più di 15.

Sulla stessa linea, si vede qui a destra una variante che somma ripetutamente 1 alla mantissa, (un *int*), che poi viene moltiplicata per il *float* 1E-17

Anche in questo caso, come si vede, si hanno **perdite** di precisione

```
php> $m = 12345678901234567; # mantissa $m int a 17 cifre, precisione float solo 15
# per questo, sommare ripetutamente 1 alla mantissa, non ha l'effetto "teorico» che
# ci si aspetterebbe, come illustrato dal codice qui sotto
```

```
php> for ($n=0;$n<94;$n++) {echo "$n: "; var_dump($m*1E-17); $m++;}
0: float(0.12345678901234569) # atteso: 0.12345678901234567
1: float(0.12345678901234569) # atteso: 0.12345678901234568
2: float(0.12345678901234569) # atteso: 0.12345678901234569
3: float(0.1234567890123457) # atteso: 0.12345678901234570
4: float(0.12345678901234573) # atteso: 0.12345678901234571
5: float(0.12345678901234573) # atteso: 0.12345678901234572
6: float(0.12345678901234573) # atteso: 0.12345678901234573
7: float(0.12345678901234575) # atteso: 0.12345678901234574
8: float(0.12345678901234577) # atteso: 0.12345678901234575
9: float(0.12345678901234577) # atteso: 0.12345678901234576
10: float(0.12345678901234577) # atteso: 0.12345678901234577
11: float(0.12345678901234579) # atteso: 0.12345678901234578
12: float(0.1234567890123458) # atteso: 0.12345678901234579
13: float(0.1234567890123458) # atteso: 0.12345678901234580
14: float(0.1234567890123458) # atteso: 0.12345678901234581
15: float(0.12345678901234583) # atteso: 0.12345678901234582
16: float(0.12345678901234584) # atteso: 0.12345678901234583
17: float(0.12345678901234584) # atteso: 0.12345678901234584
18: float(0.12345678901234584) # atteso: 0.12345678901234585
19: float(0.12345678901234587) # atteso: 0.12345678901234586
...
93: float(0.12345678901234661) # atteso: 0.12345678901234660
```

NaN e INF

Lo standard IEEE 754 consente di rappresentare i valori *float*:

- **NaN** (Not a Number): risultato di calcoli erranei
- **Inf** e **-Inf**: infinito e infinito negativo

PHP ha delle costanti *float* **NAN** e **INF** corrispondenti a questi valori

```
php> $x = sqrt(-9.0);  
php> var_dump($x);  
float(NAN)
```

```
php> echo PHP_FLOAT_MAX;  
1.7976931348623E+308  
php> echo PHP_FLOAT_MAX + PHP_FLOAT_MAX;  
INF
```

NB: il prossimo **INF** si ottiene con engine diversi dallo standard, Zend Engine > 4.2.x, che causa invece l'eccezione a runtime *DivisionByZeroError*):

```
php> $y = 1.0/0.0;  
php> var_dump($y);  
float(INF)
```

Tipo e valore *null*

È il "valore" di una variabile non inizializzata, se riferita:

```
php> error_reporting(E_ERROR | E_PARSE);    # serve a sopprimere i warning
php> var_dump($z);                          # la variabile $z non è mai stata assegnata
NULL
```

Per ri-inizializzare una variabile, vi sono più modi:

```
php> $z = 1;
php> var_dump($z);
int(1)
php> $z = NULL;          # ri-inizializzazione
php> var_dump($z);
NULL
php> $z = 1;
php> var_dump($z);
int(1)
php> unset($z);          # ri-inizializzazione
php> var_dump($z);
NULL
```

NB: la rappresentazione del valore *NULL*, è case-insensitive

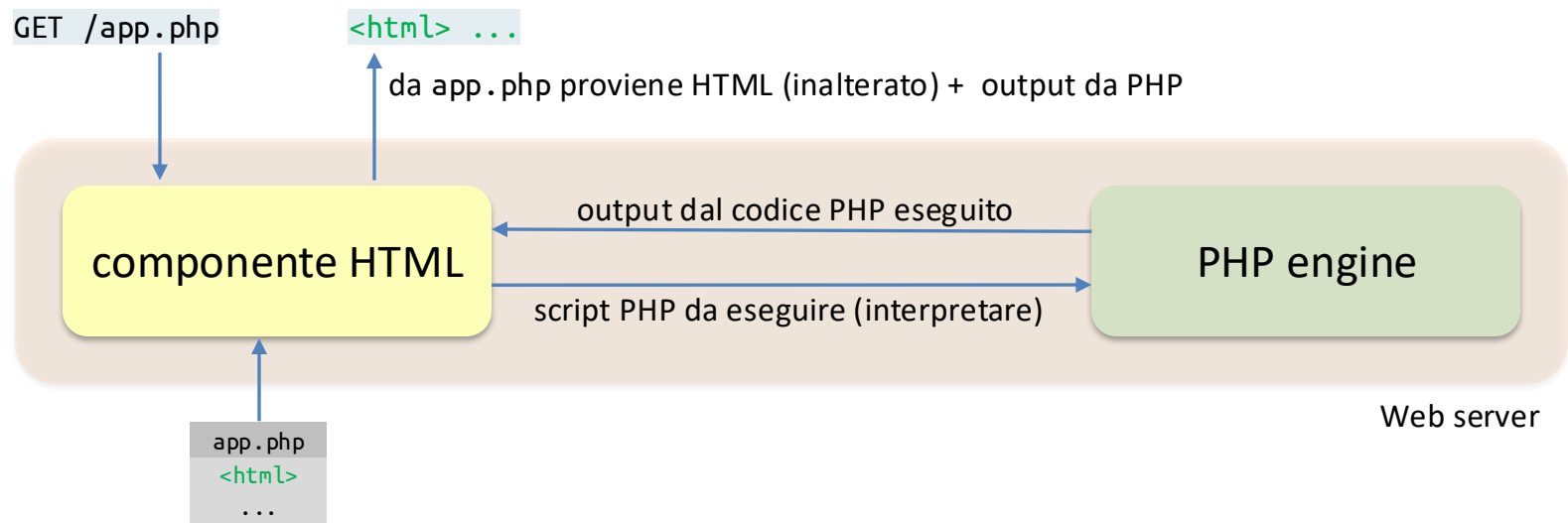
Un Web server per PHP (e HTML)

Come detto, nel servire al cliente un file *app.php*, un Web server, come *php -S* o Apache/PHP, svolge di fatto 2 attività (pressoché) indipendenti:

1. **emette** in output (verso il cliente) l'HTML in *app.php*
2. **interpreta** (analizza ed esegue) gli script PHP in *app.php* generando output (presumibilmente HTML) anche esso inviato al cliente

Il passaggio (1) → (2) avviene al tag `<?php`, quello (2) → (1) al tag `?>`

Nel seguito, è utile (e verosimile) pensare che, nel web server, l'attività (1) sia svolta da un **componente** (che emette) **HTML**, la (2) da un **PHP engine**:



"Escape" da HTML a PHP

Dunque, nel servire un file PHP con HTML, un Web server:

1. con il *componente HTML*, emette in output (verso il cliente) l'HTML
2. con il *PHP engine*, interpreta gli script PHP nel file, il che genera output HTML anche esso *inviato* al cliente

Si può pensare che il server, nel leggere il file *.php* da servire:

- parte con l'attività (1) (invio HTML), ma, appena trova il tag `<?php`,
- effettua un "**escape**" dall'HTML, va cioè all'attività (2) (interpreta PHP)
- il successivo tag `?>` di chiusura dello script PHP riporterà dalla (2) alla (1)

Le attività (1) e (2) sono indipendenti e non si influenzano a vicenda; ognuna di esse riprende da dove si era interrotta per il tag `<?php` o `?>`

Un esempio evidenzia (1), poi (2), (1), (2), (1), sul server e gli effetti sul client:

```
<!-- escape.php -->
<html><body>
First PHP tag below:<BR><BR>
<?php $x = 1; ?>
Second PHP tag below:<BR>
<?php echo 2."\\n"; ?>
</body></html>
```

→ ↻ ⓘ view-source:localhost:8000/escape.php

```
<!-- escape.php -->
<html><body>
First PHP tag below:<BR><BR>
Second PHP tag below:<BR>
2
</body></html>
```

← → ↻ ⓘ localhost:8000/escape.php

First PHP tag below:

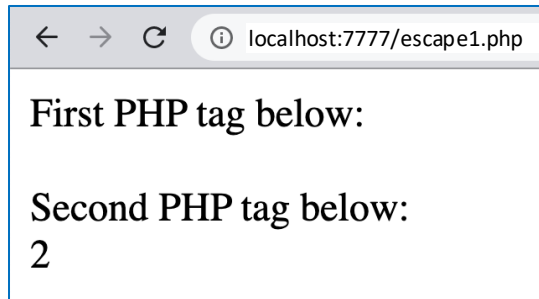
Second PHP tag below:

2

Lo stato dell'esecuzione PHP

Vediamo qui a destra un'alternanza HTML-PHP con la variabile `$x` del primo script che ricompare nel secondo:

```
<!-- escape1.php -->
<html><body>
First PHP tag below:<BR>
<?php $x = 1; ?>
Second PHP tag below:<BR>
<?php echo $x+1 ?>
</body></html>
```



Si noti come, nel secondo script, la variabile `$x` risulta definita e mantiene il valore che le era stato assegnato nel primo script

Quindi, nell'interpretazione di un file `.php`, lo **stato** (valori delle variabili) **persiste** tra uno script e i successivi e lo scope delle variabili si estende all'intero file

È, cioè, come se l'engine PHP vedesse **un unico flusso di codice PHP**, a prescindere da (1) chiusure e (2) riaperture di script (`?> ... <?php`):

1. a ogni chiusura (`?>`) il controllo va al modulo che emette l'HTML ...
2. a ogni riapertura (`<?php`), l'engine PHP riprende a interpretare PHP dal tag `?>` dove aveva smesso, senza che l'HTML ... inframmezzato abbia alcuna influenza sullo stato dell'esecuzione del codice PHP

Si veda: <https://www.php.net/manual/en/language.basic-syntax.phpmode.php>

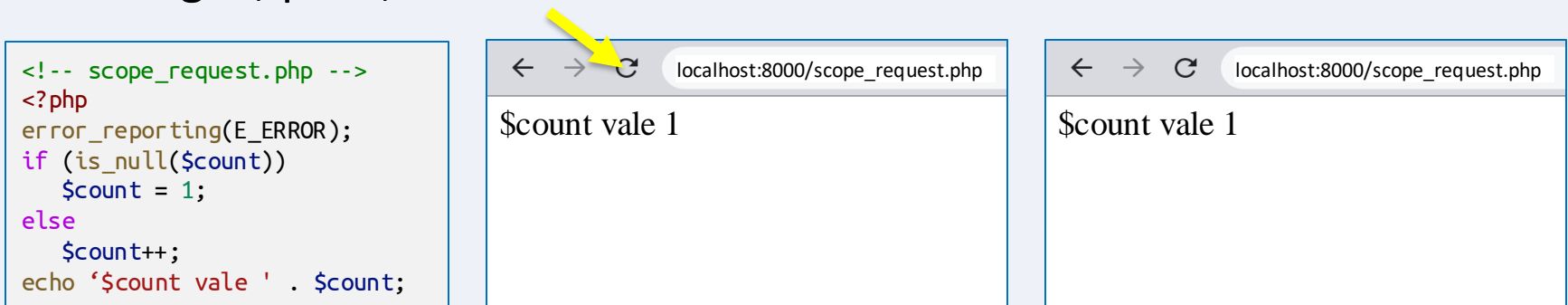
Scope delle variabili: la richiesta

Lo scope delle variabili PHP di un file *.php* è quindi il file, ma **limitatamente** alla singola esecuzione del file, attivata da ciascuna richiesta HTTP verso il file stesso.

All'inizio di ogni esecuzione, ogni variabile è indefinita (NULL) e non mantiene il valore assunto durante l'esecuzione precedente.

Così, se la business logic (codice PHP) ha necessità di preservare lo stato dell'interazione con i clienti, non può fidare sulle variabili

Si immagini, p.es., di voler contare il numero di richieste ricevute:



The diagram illustrates the concept of variable scope in PHP. It consists of three panels:

- Left Panel:** A code editor showing the PHP code for `scope_request.php`. The code initializes a variable `$count` to 1 if it is null, and increments it for subsequent requests.
- Middle Panel:** A browser screenshot showing the output `$count vale 1`. A yellow arrow points to the refresh button in the browser's address bar, indicating the first request.
- Right Panel:** A second browser screenshot showing the output `$count vale 1` after a second request, demonstrating that the variable's value is not preserved between requests.

Contare le richieste nella variabile fallisce (sarebbe ok con Java servlet)

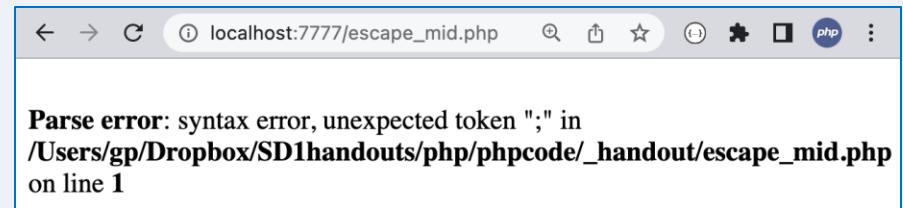
In PHP, la soluzione è memorizzare lo stato in un file o DB esterno

Script con istruzione PHP incompleta

Finora si è tacitamente supposto che lo script `<?php ... ?>` contiene in `...` una o più istruzioni PHP (sintatticamente) complete e, in tal caso, è ovvio che, a seguito dell'escape da HTML, l'engine eseguirà `...` completamente. Chiediamoci però: il tag `?>` può anche interrompere il codice PHP nel mezzo di un'istruzione (che poi riprenderà con un successivo `<?php`)?

In generale,
no!

```
<!-- escape_mid.php -->  
<?php $temperatura = ?>  
Come va il tempo?<BR>  
<?php 30; ?>
```



Vi sono però delle eccezioni, vediamo tre importanti: con il tag `?>`

1. si può interrompere un blocco PHP `{istr1;istr2;... }` dopo `{` o un `;`
2. si può interrompere un costrutto PHP `if (cond):` oppure `elseif (cond):` o `else:` dopo `:` (vedi oltre per la sintassi di tali costrutti)
3. dopo `:` si può anche interrompere un costrutto `for (s1,cond,s2):`

Istruzione condizionale: formato alternativo

L'istruzione condizionale *if... elseif... else* ha la sintassi C-like mostrata nel box qui a destra

È consigliabile (v. [oltre](#)) usare le graffe {...} anche se, come in C, potrebbero in effetti omettersi in presenza di un solo statement

In una forma alternativa (v. a fianco), si usa : al posto di {...} per delimitare il codice attivato da ciascuna (condition) e da else

```
if (condition) {  
    statements; ...  
} elseif (condition) {  
    statements; ...  
} else {  
    statements; ...  
}
```

```
if (condition):  
    statements; ...  
elseif (condition):  
    statements; ...  
else:  
    statements; ...  
endif;
```

```
if (condition) {  
    statements; ...  
} elseif (condition) {  
    statements; ...  
} else {  
    statements; ...  
}
```

Istruzione condizionale “mista” (con HTML)

Vi è anche una forma “mista” di `if ... endif` in cui le `(condition):` o l' `else:` presenti nel codice PHP, se **true**, attivano dei blocchi di **codice HTML**

Notare gli *escape* multipli PHP→HTML→PHP (è il caso (2) di una [slide precedente](#))

```
...  
if (condition): ?>  
    HTML code ...  
<?php elseif (condition): ?>  
    HTML code ...  
<?php else: ?>  
    HTML code ...  
<?php endif; ?>  
...
```

Tipicamente, ciò consente di inviare al cliente blocchi alternativi di codice HTML, secondo lo stato di certe variabili PHP (che riflettono lo stato del DB o dati inviati dal cliente p.es. con form web)

Nell'esempio a destra, la condizione `$x == 0` falsa fa sì che sia emesso il codice HTML `condizione falsa`

Se la prima istruzione fosse `$x = 10` verrebbe emesso invece il codice HTML `condizione vera`

```
HTML qui, ora PHP<BR>  
<?php  
$x = 10;  
if ($x == 0): ?>  
    <B>condizione vera</B>  
<?php else: ?>  
    <B>condizione falsa</B>  
<?php endif; ?>  
<BR>fine PHP, ora HTML
```

localhost:8000/escape_if_1.php

HTML qui, ora PHP
condizione falsa
fine PHP

(Meglio ancora, il valore di `$x` potrebbe provenire da input effettuati sul cliente)

Escape PHP \longleftrightarrow HTML nell'**if** ... **endif** "misto"

Come interagiscono nell'**if** ... **endif** misto, p.es. qui a destra, **PHP engine** e **modulo HTML** del server?

- al tag di chiusura in `... == 0) ?>`, il **PHP engine** ha in corso una istruzione **if** con condizione **false**, quindi sa di doverne eseguire il ramo **else**, quindi...
- **non dà** il controllo al **modulo HTML** e cerca il primo script PHP `<?php else:?>`
- solo dopo aver "ingerito" anche `<?php else:?>`, il **PHP engine** passa il controllo al **modulo HTML** che emette il codice `condizione falsa`

Se, invece, la condizione fosse **true**, il controllo andrebbe subito al **modulo HTML** che emetterebbe `condizione vera`

```
HTML qui, ora PHP<BR>
<?php
$x = 10;
if ($x == 0): ?>
    <B>condizione vera</B>
<?php else: ?>
    <B>condizione falsa</B>
<?php endif; ?>
<BR>fine PHP, ora HTML
```

localhost:8000/escape_if_1.php

HTML qui, ora PHP
condizione falsa
fine PHP

L'engine PHP mantiene lo stato dell'esecuzione

Quindi: dentro un file `.php`, l'engine PHP mantiene, tra uno script `<?php ... ?>` e il successivo non solo lo stato delle variabili, ma anche lo **stato dell'esecuzione**. Cioè: se al tag `?>` di chiusura di uno script, l'esecuzione richiede un salto, l'engine mantiene il controllo e va a cercare il successivo script `<?php ... ?>` a cui saltare!

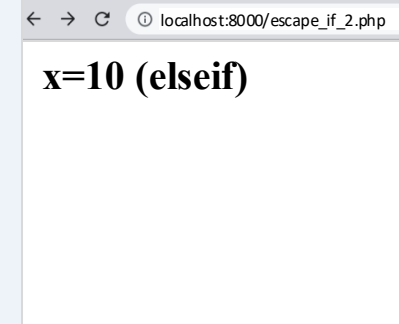
```
<?php if (cond-1): ?>
    codice-per-cond-1
<?php elseif (cond-2): ?>
    codice-per-cond-2
<?php else: ?>
    codice-per-else
<?php endif; ?>
```

Nello schema mostrato nel box, se `cond-1` è falsa, l'engine bypassa il relativo `codice-per-cond-1` e salta al successivo script `<?php elseif ...`, e così via.

Tipicamente, ciò consente di inviare al cliente blocchi alternativi di codice HTML (v. esempio [precedente](#)), secondo lo stato di certe variabili PHP (che riflettono lo stato del DB o dati inviati dal cliente p.es. con form web)

NB: `codice-per-cond-k` in generale può contenere **sia HTML che PHP**, che sarà eseguito solo se `cond-k` è falsa, bypassato altrimenti, v. esempio qui a destra

```
<?php $x = 10; ?>
<?php if ($x == 0): ?>
    <B>x=<?php echo $x;?> (then)</B>
<?php elseif ($x <= 100): ?>
    <B>x=<?php echo $x;?> (elseif)</B>
<?php else: ?>
    <B>x=<?php echo $x;?> (else)</B>
<?php endif; ?>
```



Istruzione for “mista” (con HTML)

Anche per `for (...) ...` c'è una sintassi alternativa (v. box)

```
for (init; cond; reinit):  
    statements;  
endfor;
```

```
for (init; cond; reinit) {  
    statements;  
}
```

E anche per `for (...):` si può interrompere lo script PHP dopo `(...):`, proseguire con codice HTML/PHP per il loop e chiudere il loop con `endfor` :

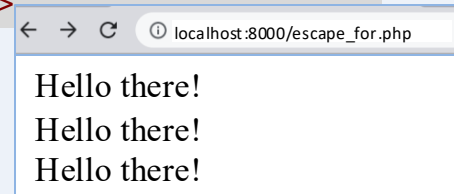
```
<?php for (init; cond; reinit): ?>  
    codice-HTML  
    [codice-PHP-opzionale]  
    ...  
<?php endfor; ?>
```

Al tag di chiusura `?>` del `for`, PHP engine valuta `cond` e:

- se `cond` è true, il controllo va al modulo HTML che elabora il `codice-HTML` del loop
- se `cond` è false, salta oltre il `<?php endfor; ?>` che chiude il loop

Di nuovo, quindi, se dopo il tag `?>` di chiusura dello script `<?php for (...): ?>`, il flusso dell'esecuzione richiede un salto, l'engine PHP va a cercare il punto del file `.php` al quale saltare! E cioè `<?php endfor; ?>` (v. esempio qui a destra)

```
<?php for ($i = 0; $i < 3; ++$i): ?>  
Hello, there!<BR>  
<?php endfor; ?>
```



Costrutti HTML/PHP misti e blocchi { ... }

I costrutti PHP “misti”, interrotti con tag `?>` e HTML “nel mezzo”, sono ammessi anche con la sintassi in cui `{` è al posto di `:` e `}` al posto dell'`endfor`

```
HTML here<BR>
<?php
$x = 10;
if ($x == 0) { ?>
    <B>condizione vera</B>
<?php } else { ?>
    <B>condizione falsa</B>
<?php } ?>
<BR>HTML again
```



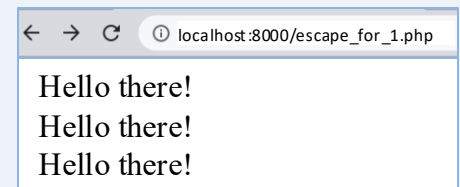
NB: le graffe sono necessarie (anche se non lo sono nell' `if` “non interrotto”, se i rami `if` e `else` sono istruzioni singole); qui omettere `{` dopo `if` causa errore sintattico, mentre dopo `else` ha un effetto inatteso:

```
HTML here<BR>
<?php $x = 0;
if ($x == 0) ?>
    <B>condizione vera</B>
<?php else ?>
    <B>condizione falsa</B>
<?php ?><BR>HTML again
```



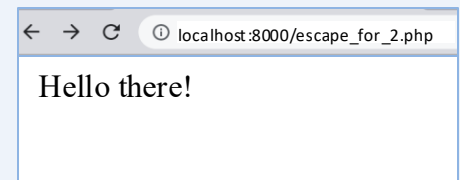
Il caso del `for` è del tutto analogo:

```
<!-- escape_for_1.php -->
<?php for ($i = 0; $i < 3; ++$i) { ?>
    Hello, there!<BR>
<?php } ?>
```



Pure col `for`, omettere `{e }` produce risultati forse inattesi e inutili:

```
<!-- escape_for_2.php -->
<?php for ($i = 0; $i < 3; ++$i) ?>
    Hello, there!<BR>
<?php ?>
```



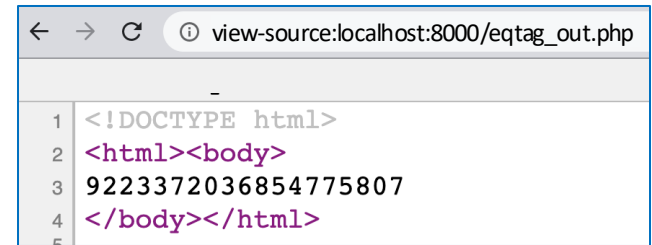
Il tag `<?=`

All'interno di un file `.html`, il tag `<?=` è trattato come una abbreviazione di `<?php echo`

Pertanto, se, all'interno di un file `.html` del server compare uno script `<?= ... ?>`, tale script viene passato all'engine PHP e, se `...` è un'espressione, l'engine la valuta e invia in output (come con `echo`) il valore di `...`, che viene quindi inserito nell'HTML prodotto

Qui a destra l'espressione valutata è la costante (predefinita) `PHP_INT_MAX`

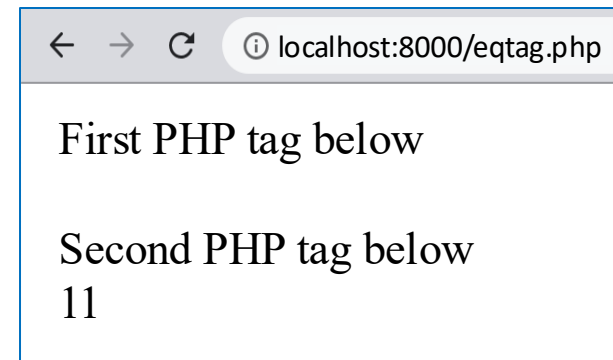
```
<!-- eqtag_out.php -->
<!DOCTYPE html>
<html><body>
<?= PHP_INT_MAX ?>
</body></html>
```



```
1 <!DOCTYPE html>
2 <html><body>
3 9223372036854775807
4 </body></html>
5
```

Qui a destra, in `<?= ... ?>` compare l'espressione aritmetica `$x+1`, di cui viene emesso il valore (11)

```
<!-- eqtag1.php -->
<!DOCTYPE html>
<html><body>
First PHP tag below:<BR>
<?php $x = 10; ?>
Second PHP tag below:<BR>
<?= $x+1 ?>
</body></html>
```



```
<-- > <?php $x = 10; ?>
First PHP tag below
Second PHP tag below
11
```


Tag `<?=` vs. `<?php`

Come detto, in un file `.html`, il tag `<?=` equivale a `<?php echo`

Esso serve a inserire **in maniera concisa**, all'interno dell'`.html` inviato al client, il **valore** dell'espressione che compare tra `<?=` e `?>`

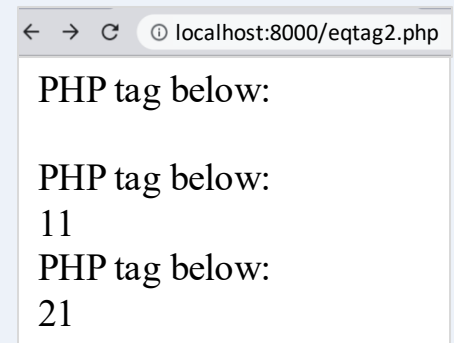
L'utilità di ciò deriva da un aspetto che conviene sottolineare: quando l'engine PHP è attivata da un tag `<?php`, essa inserisce nell'html generato, **soltanto** il testo che il codice PHP eseguito invia **esplicitamente** alla standard output e **non i valori** che esso produce.

Qui a destra, `$x = 1`, oltre a assegnare a `$x` il valore `1`, "produce" il valore `1` (come in C, l'assegnazione ha un valore), che però non viene emesso in output, quindi non lascia traccia

```
<!-- eqtag2.php -->
PHP tag below:<BR>
<?php $x = 1; ?>

PHP tag below:<BR>
<?= $x += 10 ?>

<BR>PHP tag below:<BR>
<?php echo($x += 10) ?>
```



localhost:8000/eqtag2.php

PHP tag below:

11

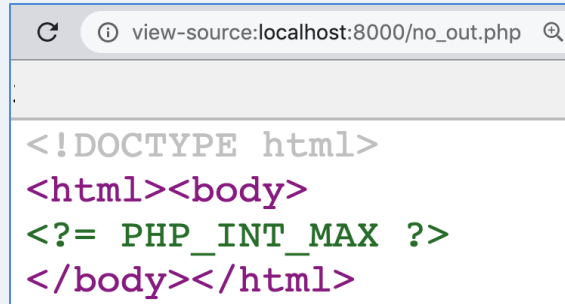
PHP tag below:

21

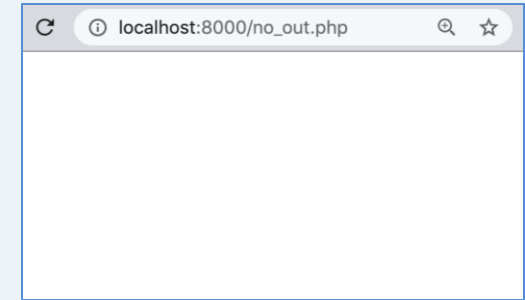
Al contrario, nel secondo script `<?= ... ?>` qui sopra e nel terzo (equivalente) `<?php echo ... ?>`, il valore prodotto dall'assegnazione `$x += 10` viene emesso in output

Un chiarimento chiesto a lezione

```
<!-- no_out.php -->
<!DOCTYPE html>
<html><body>
<?php
echo '<?= PHP_INT_MAX ?>'; ?>
</body></html>
```



```
view-source:localhost:8000/no_out.php
<!-- no_out.php -->
<!DOCTYPE html>
<html><body>
<?php
echo '<?= PHP_INT_MAX ?>'; ?>
</body></html>
```



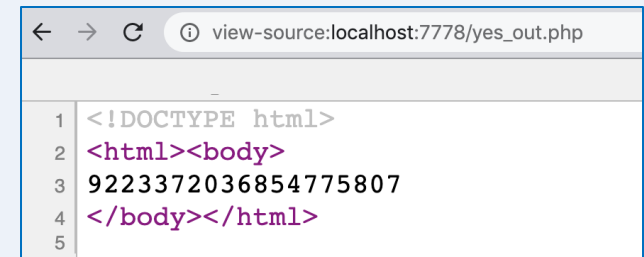
Perché l'output di `no_out.php` sul client è nullo? L'errore è nel **codice rosa**

L'istruzione `echo '<?= ... ?>';` contenuta nello script fa sì che l'HTML ricevuto dal browser contenga la coppia di tag `<?= ... ?>`, la quale:

- per il browser non ha senso (e non dà output), v. riquadri sopra
- ha senso invece se è "embedded" in un file `.php` lato server, infatti:
 - il tag `<?=` è un'abbreviazione di `<?php echo` quindi, se `...` in `<?= ... ?>` è un'espressione, `...` viene valutata e il valore viene

inserito nell'HTML
prodotto, come nell'
esempio qui a destra

```
<!-- yes_out.php -->
<!DOCTYPE html>
<html><body>
<?= PHP_INT_MAX ?>
</body></html>
```



```
view-source:localhost:7778/yes_out.php
1 <!-- yes_out.php -->
2 <!DOCTYPE html>
3 <html><body>
4 <?= PHP_INT_MAX ?>
5 </body></html>
```

Riferimenti o alias (&)

Data la variabile `$x` (anche NULL), `&$x` rappresenta un riferimento alla variabile `$x`, o alias di `$x`

Il riferimento si può assegnare ad altra variabile `$z`

Dopo, `$z` potrà figurare al posto di `$x` *in qualsiasi contesto*, anche come target di un assegnazione, senza alterare il comportamento del codice

Sono ammessi anche riferimenti a una variabile array `$a` (v. esempio qui a destra)

Sono poi possibili riferimenti a un elemento di una variabile array (v. esempio a destra)

```
php> $z = &$x;  
php> $x = 10;  
php> echo $z;  
10  
php> $z = $x / 2;  
php> echo $x;  
5  
php> echo $z;  
5
```

```
php> $a = [22,27,25];  
php> $aa = &$a;  
php> echo $aa[1];  
27  
php> $aa[1] = 30;  
php> echo $a[1];  
30
```

```
php> $a1 = &$a[1];  
php> echo $a1;  
30  
php> $a1 = 18;  
php> echo $a[1];  
18
```

Funzioni e parametri alias (&)

Una funzione PHP `f()` può avere un *parametro* `$x`

In un'invocazione `f(expr)` della funzione, al posto di `$x` figura un'espressione `expr` detta *argomento*

All'atto dell'esecuzione dell'invocazione `f(expr)`, `expr` viene valutata e il parametro `$x` è come una variabile con il valore di `expr`

NB: anche se `expr` è una variabile `$w`, l'invocazione `f($w)` non può modificare l'argomento `$w` attraverso il parametro `$x`, che è solo una **copia** della variabile `$w`

Perché una funzione `fa()` riesca a modificare un argomento variabile come la `$w`, occorre che nella sua definizione figurino un parametro *alias* `&$x`

In questo caso, il parametro `$x` si comporterà come un **alias** dell'argomento `$w`, e le modifiche su `$x` si rifletteranno su `$w`

```
<!-- fun.php -->
<?php
function f($x) {
    $x++; echo $x;
}
```

```
php> include "fun.php";
php> f(3*2);
7
```

```
php> $w = 10;
php> f($w);
11
php> echo $w;
10
```

```
<?php
function fa(&$x) {
    $x++; echo $x;
}
```

```
php> fa($w);
11
php> echo $w;
11
```

foreach e alias (&)

Come noto, in un ciclo `foreach ($a as $x)` la variabile indice `$x` assume, a ogni iterazione, il valore di ciascun elemento dell'array `$a`

Più precisamente, a ogni ciclo, in `$x` viene memorizzata una **copia** dell'elemento corrente di `$a`

Ne segue che ogni modifica di `$x` non produce effetti sull'array `$a` (cf. box qui sotto a sinistra)

```
<!-- foreach0.php -->
<?php
$voti = [22, 27, 21];
foreach ($voti as $voto)
    $voto = $voto+2;
var_dump($voti);
// output: [22, 27, 21]
```

```
<!-- foreach1.php -->
<?php
$voti = [22, 27, 21];
foreach ($voti as &$voto)
    $voto = $voto+2;
var_dump($voti);
// output: [24, 29, 23]
```

Invece, in un `foreach ($a as &$x)` (box sopra, a destra) l'indice `$x` sarà, a ogni iterazione, un **riferimento** all'elemento corrente dell'array `$a` o, come si dice, un **alias** dell'elemento corrente

Ora, modificando `$x` si modifica l'elemento corrente dell'array `$a`!