COMP519 Web Programming
Lecture 20: PHP (Part 2)
Handouts

Ullrich Hustadt

Department of Computer Science
School of Electrical Engineering, Electronics, and Computer Science
University of Liverpool
Contents

1 Comparisons

2 Arrays
   Basics
   Foreach-loops
   Array Operators

3 Revision and Further Reading
### Comparison Operators

**Type juggling** also plays a role in the way PHP comparison operators work:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Truth Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>$expr1 == $expr2</code></td>
<td>Equal</td>
<td>$\text{TRUE}$ iff $\text{expr1}$ is equal to $\text{expr2}$ after type juggling</td>
</tr>
<tr>
<td><code>$expr1 !== $expr2</code></td>
<td>Not equal</td>
<td>$\text{FALSE}$ iff $\text{expr1}$ is not equal to $\text{expr2}$ after type juggling</td>
</tr>
<tr>
<td><code>$expr1 !== $expr2</code></td>
<td>Not equal</td>
<td>$\text{FALSE}$ iff $\text{expr1}$ is not equal to $\text{expr2}$ after type juggling</td>
</tr>
<tr>
<td><code>$expr1 === $expr2</code></td>
<td>Identical</td>
<td>$\text{FALSE}$ iff $\text{expr1}$ is not equal to $\text{expr2}$, or they are not of the same type</td>
</tr>
<tr>
<td><code>$expr1 !== $expr2</code></td>
<td>Not identical</td>
<td>$\text{FALSE}$ iff $\text{expr1}$ is not equal to $\text{expr2}$, or they are not of the same type</td>
</tr>
</tbody>
</table>

**Note:** For `==`, `!==`, and `<>`, *numerical strings* are converted to numbers and compared numerically:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;123&quot; == 123</td>
<td>$\text{TRUE}$</td>
</tr>
<tr>
<td>&quot;123&quot; !== 123</td>
<td>$\text{FALSE}$</td>
</tr>
<tr>
<td>&quot;1.23e2&quot; == 123</td>
<td>$\text{TRUE}$</td>
</tr>
<tr>
<td>&quot;1.23e2&quot; == &quot;12.3e1&quot;</td>
<td>$\text{TRUE}$</td>
</tr>
<tr>
<td>&quot;10hello5&quot; == 10</td>
<td>$\text{TRUE}$</td>
</tr>
<tr>
<td>5 == TRUE</td>
<td>$\text{TRUE}$</td>
</tr>
<tr>
<td>&quot;123&quot; === 123</td>
<td>$\text{FALSE}$</td>
</tr>
<tr>
<td>&quot;123&quot; !== 123</td>
<td>$\text{TRUE}$</td>
</tr>
<tr>
<td>1.23e2 === 123</td>
<td>$\text{FALSE}$</td>
</tr>
<tr>
<td>&quot;1.23e2&quot; !== &quot;12.3e1&quot;</td>
<td>$\text{FALSE}$</td>
</tr>
<tr>
<td>&quot;10hello5&quot; !== 10&quot;</td>
<td>$\text{FALSE}$</td>
</tr>
<tr>
<td>5 !== TRUE</td>
<td>$\text{FALSE}$</td>
</tr>
</tbody>
</table>
# Comparison Operators

Type juggling also plays a role in the way PHP comparison operators work:

<table>
<thead>
<tr>
<th>Comparison</th>
<th>Description</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>expr1 &lt; expr2</code></td>
<td>Less than</td>
<td>TRUE iff <code>expr1</code> is strictly less than <code>expr2</code> after type juggling</td>
</tr>
<tr>
<td><code>expr1 &gt; expr2</code></td>
<td>Greater than</td>
<td>TRUE iff <code>expr1</code> is strictly greater than <code>expr2</code> after type juggling</td>
</tr>
<tr>
<td><code>expr1 &lt;= expr2</code></td>
<td>Less than or equal to</td>
<td>TRUE iff <code>expr1</code> is less than or equal to <code>expr2</code> after type juggling</td>
</tr>
<tr>
<td><code>expr1 &gt;= expr2</code></td>
<td>Greater than or equal to</td>
<td>TRUE iff <code>expr1</code> is greater than or equal to <code>expr2</code> after type juggling</td>
</tr>
</tbody>
</table>

Note: For `>`, `>=`, `<`, and `<=` numerical strings are converted to numbers and compared numerically

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>'35.5' &gt; 35</td>
<td>TRUE</td>
</tr>
<tr>
<td>'ABD' &gt; 'ABC'</td>
<td>TRUE</td>
</tr>
<tr>
<td>'1.23e2' &gt; '12.3e1'</td>
<td>FALSE</td>
</tr>
<tr>
<td>&quot;F1&quot; &lt; &quot;G0&quot;</td>
<td>TRUE</td>
</tr>
<tr>
<td>TRUE &gt; FALSE</td>
<td>TRUE</td>
</tr>
<tr>
<td>5 &gt; TRUE</td>
<td>FALSE</td>
</tr>
<tr>
<td>'35.5' &gt;= 35</td>
<td>TRUE</td>
</tr>
<tr>
<td>'ABD' &gt;= 'ABC'</td>
<td>TRUE</td>
</tr>
<tr>
<td>'1.23e2' &gt;= '12.3e1'</td>
<td>TRUE</td>
</tr>
<tr>
<td>&quot;F1&quot; &lt;= &quot;G0&quot;</td>
<td>TRUE</td>
</tr>
<tr>
<td>TRUE &gt;= FALSE</td>
<td>TRUE</td>
</tr>
<tr>
<td>5 &gt;= TRUE</td>
<td>TRUE</td>
</tr>
</tbody>
</table>
Comparisons

Comparison operators

- To compare strings ‘as strings’ the `strcmp` function can be used
- PHP 7 introduced the so-called ‘spaceship operator’ for three-way comparisons (that converts numeric strings to numbers)

<table>
<thead>
<tr>
<th>Function</th>
<th>String comparison</th>
<th>Returns</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>strcmp(expr1, expr2)</code></td>
<td>For string comparison. Returns <code>&lt; 0</code> if <code>expr1</code> is less than <code>expr2</code>, <code>&gt; 0</code> if <code>expr1</code> is greater than <code>expr2</code>, <code>0</code> if <code>expr1</code> is equal to <code>expr2</code>.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>expr1 &lt;=&gt; expr2</code></td>
<td>Three-way comparison. Returns <code>-1</code> if <code>expr1</code> &lt; <code>expr2</code>, <code>+1</code> if <code>expr1</code> &gt; <code>expr2</code>, <code>0</code> if <code>expr1</code> == <code>expr2</code>.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Examples**

```plaintext
strcmp('ABD','ABC') \(\sim\) 1
strcmp('aaa','aaa') \(\sim\) 0
strcmp('1.23e2','12.3e1') \(\sim\) -1

'ABD' <=> 'ABC' \(\sim\) 1
'aaa' <=> 'aaa' \(\sim\) 0
'1.23e2' <=> '12.3e1' \(\sim\) 0

35.5' <=> 35 \(\sim\) 1
TRUE <=> FALSE \(\sim\) 1
5 <=> TRUE \(\sim\) 0

'F1' <=> 'G0' \(\sim\) -65536
'AAA' <=> 'aaa' \(\sim\) 2105376
'10hello5' <=> 10 \(\sim\) 0
0.0 <=> FALSE \(\sim\) 0
'FALSE' <=> TRUE \(\sim\) 0
```
## Integers and Floating-point numbers: NaN and Inf

NaN and Inf can be compared with each other and other numbers using equality and comparison operators:

<table>
<thead>
<tr>
<th>NaN == NaN</th>
<th>NaN === NaN</th>
<th>NaN == 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>FALSE</td>
<td>FALSE</td>
<td>FALSE</td>
</tr>
<tr>
<td>INF == INF</td>
<td>INF === INF</td>
<td>INF == 1</td>
</tr>
<tr>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
<tr>
<td>NaN &lt; NaN</td>
<td>INF &lt; INF</td>
<td>1 &lt; INF</td>
</tr>
<tr>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
<tr>
<td>NaN &lt; INF</td>
<td>INF &lt; NaN</td>
<td>INF &lt; 1</td>
</tr>
<tr>
<td>TRUE</td>
<td>TRUE</td>
<td>FALSE</td>
</tr>
<tr>
<td>NaN &lt; 1</td>
<td>1 &lt; NaN</td>
<td>TRUE</td>
</tr>
</tbody>
</table>

In PHP 5.3 and earlier versions, INF == INF returns FALSE
INF === INF returns TRUE

In PHP 5.4 and later versions, INF == INF returns TRUE
INF === INF returns TRUE
Integers and Floating-point numbers: NAN and INF

- PHP provides three functions to test whether a value is or is not NAN, INF or -INF:
  - `bool is_nan(value)`
    returns TRUE iff `value` is NAN
  - `bool is_infinite(value)`
    returns TRUE iff `value` is INF or -INF
  - `bool is_finite(value)`
    returns TRUE iff `value` is neither NAN nor INF/-INF

- In conversion to a `boolean` value, both NAN and INF are converted to `TRUE`
- In conversion to a `string`, NAN converts to 'NAN' and INF converts to 'INF'
Arrays

- PHP only supports **associative arrays (hashes)**, simply called **arrays**
- PHP **arrays** are created using the **array** construct or, since PHP 5.4, `[ ... ]:

  ```php
  array(
    key => value, ...
  )
  [key => value, ...]
  ```

  where **key** is an integer or string and **value** can be of any type, including **arrays**

  ```php
  $arr1 = [1 => "Peter", 3 => 2009, "a" => 101];
  $arr2 = array(200846369 => array("name" => "Jan Olsen",
                                      "COMP518" => 69,
                                      "COMP519" => 52));
  ```

- The size of an array can be determined using the **count** function:

  ```php
  int count(array[, mode])
  ```

  ```php
  print count($arr1); // prints 3
  print count($arr2); // prints 1
  print count($arr2,1); // prints 4
  ```
Arrays

- It is possible to omit the keys when using the `array` construct:

  ```php
  $arr3 = array("Peter", "Paul", "Mary");
  ```

  The values given in `array` will then be associated with the natural numbers 0, 1, ...

- All the keys of an array can be retrieved using

  ```php
  array_keys($array1)
  ```

  → returns a natural number-indexed array containing the keys of `$array1`

- All the values of an array can be retrieved using

  ```php
  array_values($array1)
  ```

  → returns a natural number-indexed array containing the values stored in `$array1`
Arrays

- An individual array element can be accessed via its key
- Accessing an undefined key produces a PHP notice and returns NULL

```php
$arr1 = array(1 => "Peter", 3 => 2009, "a" => 101);
print "'a' => \"".$arr1["a"]."\n";
'a' => 101

print "'b' => \"".$arr1["b"]."\n";
PHP Notice: Undefined index: b in <file> on line <lineno>
'b' => // $arr1["b"] returns NULL

$arr1["b"] = 102;
print "'b' => \"".$arr1["b"]."\n";
'b' => 102

- The function `array_key_exists(key, array1)` can be used to check whether there is a value for key in array1

`array_key_exists("a",$arr1)` # returns TRUE
`array_key_exists("c",$arr1)` # returns FALSE
Arrays

- **PHP allows the construct**

  ```php
  $array[] = value;
  ```

  PHP will determine the maximum value $M$ among the integer indices in `$array` and use the key $K = M + 1$; if there are no integer indices in `$array`, then $K = 0$ will be used.

  ~ auto-increment for array keys

  ```php
  $arr4[] = 51;  // 0 => 51
  $arr4[] = 42;  // 1 => 42
  $arr4[] = 33;  // 2 => 33
  ```

- A key-value pair can be removed from an array using the `unset` function:

  ```php
  $arr1 = array(1 => "Peter", 3 => 2009, "a" => 101);
  unset($arr1[3]);  // Removes the pair 3 => 2009
  unset($arr1);     // Removes the whole array
  ```
Arrays: foreach-loop

- PHP provides a **foreach-loop** construct to ‘loop’ through the elements of an array

```php
foreach (array as $value) {
    statement
}
```

```php
foreach (array as $key => $value) {
    statement
}
```

- `array` is an array expression
- `$key` and `$value` are two variables, storing a different key-value pair in `array` at each iteration of the **foreach-loop**
- We call `$value` the **foreach-variable**
- **foreach** iterates through an array in the order in which elements were defined
**Arrays: foreach-loop**

`foreach` iterates through an array in the order in which elements were defined.

**Example 1:**
```php
foreach (array("Peter", "Paul", "Mary") as $key => $value) {
    print "The array maps $key to $value\n";
}
```

The array maps 0 to Peter
The array maps 1 to Paul
The array maps 2 to Mary

**Example 2:**
```php
$arr5[2] = "Mary";
$arr5[0] = "Peter";
$arr5[1] = "Paul";
// 0 => 'Peter', 1 => 'Paul', 2 => 'Mary'
foreach ($arr5 as $key => $value) {
    print "The array maps $key to $value\n";
}
```

The array maps 2 to Mary
The array maps 0 to Peter
The array maps 1 to Paul
Arrays: foreach-loop

Does changing the value of the **foreach-variable** change the element of the list that it currently stores?

Example 3:

```php
$arr6 = array("name" => "Peter", "year" => 2009);

foreach ($arr6 as $key => $value) {
    print "The array maps $key to $value
";
    $value .= " - modified"; // Changing $value
}
print "\n";
```

The **array** maps name to Peter
The **array** maps year to 2009

```php
foreach ($arr6 as $key => $value)
    print "The array now maps $key to $value\n";
```

The **array** now maps name to Peter
The **array** now maps year to 2009
Arrays: foreach-loop

- In order to modify array elements within a **foreach-loop** we need use a reference

```php
foreach (array as &\$value)
    statement
unset(\$value);
```

```php
foreach (array as \$key => &\$value)
    statement
unset(\$value);
```

- In the code schemata above, **\$value** is a variable whose value is stored at the same location as an array element
- PHP does not allow the **key** to be a reference
- The **unset** statement is important to return **\$value** to being a ‘normal’ variable
Arrays: foreach-loop

- In order to modify array elements within a **foreach**-loop we need use a reference.
- In each iteration, `$value` contains a reference to an array element; changing `$value` changes the array element.

```php
$arr6 = array("name" => "Peter", "year" => 2009);
foreach ($arr6 as $key => & $value) {
    // Note: reference!
    print "The array maps $key to $value
"
    $value .= " - modified"
}
unset($value);  // Remove the reference from $value
print "\n";

The array maps name to Peter
The array maps year to 2009

// See what the content of $arr6 is now
foreach ($arr6 as $key => $value)
    print "The array now maps $key to $value
";

The array now maps name to Peter - modified
The array now maps year to 2009 - modified
Array Assignments

- In JavaScript arrays were objects and as a consequence array assignments were done by reference.
- In PHP, this is not the case.

```
$mem1 = memory_get_usage();
$array1 = range(1, 1000);
$mem2 = memory_get_usage();
echo "(1)", sprintf("%6d", $mem2-$mem1), "more bytes\n";
$array2 = $array1;
$mem3 = memory_get_usage();
echo "(2)", sprintf("%6d", $mem3-$mem2), "more bytes\n";
$array2[1] += 10000;
$mem4 = memory_get_usage();
echo "(3)", sprintf("%6d", $mem4-$mem3), "more bytes\n";
```

(1) 36920 more bytes
(2) 0 more bytes
(3) 36920 more bytes

The PHP implementation uses copy-on-write for array assignments.
Arrays

Array Assignments

- The PHP implementation uses **copy-on-write** for array assignments.
- If we want two array variables to point to the same array literal, then we need to explicitly use a **reference**.

```php
$array1 = range(1, 1000);
$mem2 = memory_get_usage();
$array2 = &$array1;
$mem3 = memory_get_usage();

echo "(2)\n", sprintf("%6d", $mem3-$mem2), " more bytes\n";
$array2[1] += 10000;

echo "$array1[1]=\n", $array1[1], " | \\
";
echo "$array2[1]=\n", $array2[1], "\n";
$mem4 = memory_get_usage();

echo "(3)\n", sprintf("%6d", $mem4-$mem3), " more bytes\n";
```

(2) 24 more bytes


(3) 0 more bytes
Array Operators

PHP has no **stack** or **queue** data structures, but has **stack** and **queue** operators for **arrays**:

- **array_push**($array, value1, value2, ...)**
  appends one or more elements at the end of an array variable; returns the number of elements in the resulting array

- **array_pop**(array)
  extracts the last element from an array and returns it

- **array_shift**(array)
  shift extracts the first element of an array and returns it

- **array_unshift**($array, value1, value2, ...)**
  inserts one or more elements at the start of an array variable; returns the number of elements in the resulting array

Note: **&$array** needs to be a **variable**
Revision and Further Reading

- Read
  - Chapter 4: Expressions and Control Flow in PHP: Operators
  - Chapter 6: PHP Arrays


- Read
  - Language Reference: Types: Arrays
  - Language Reference: Control Structures: foreach