

- A way to think of an array is like a string of beads, with the beads representing variables that can be numbers, strings, or even other arrays. They are like bead strings because each element has its own location and (with the exception of the first and last ones) each has other elements on either side.
- Some arrays are referenced by numeric indexes; others allow alphanumeric identifiers.
- Built-in functions let you sort them, add or remove sections, and walk through them to handle each item through a special kind of loop. And by placing one or more arrays inside another, you can create arrays of two, three, or any number of dimensions.

### **Numerically Indexed Arrays**

- Example: Let's assume that you've been tasked with creating a simple website for a local office supply company and you're currently working on the section devoted to paper. One way to manage the various items of stock in this category would be to place them in a numeric array.
- *Example 6-1. Adding items to an array*

```
<?php
$paper[] = "Copier";
$paper[] = "Inkjet";
$paper[] = "Laser";
$paper[] = "Photo";
print_r($paper);
?>
```
- In this example, each time you assign a value to the array `$paper`, the first empty location within that array is used to store

the value, and a pointer internal to PHP is incremented to point to the next free location, ready for future insertions.

- The `print_r` function (which prints out the contents of a variable, array, or object) is used to verify that the array has been correctly populated. It prints out the following:

- **Array**  
(  
  **[0] => Copier**  
  **[1] => Inkjet**  
  **[2] => Laser**  
  **[3] => Photo**  
)

- The previous code could also have been written as shown in Example 6-2, where the exact location of each item within the array is specified. But, that approach requires extra typing and makes your code harder to maintain if you want to insert supplies into or remove them from the array. So, unless you wish to specify a different order, it's usually better to simply let PHP handle the actual location numbers.

- *Example 6-2. Adding items to an array using explicit locations*

```
<?php
$paper[0] = "Copier";
$paper[1] = "Inkjet";
$paper[2] = "Laser";
$paper[3] = "Photo";
print_r($paper);
?>
```

- The output from these examples is identical, but you are not likely to use `print_r` in a developed website, so Example 6-3 shows how you might print out the various types of paper the website offers using a `for` loop.

- *Example 6-3. Adding items to an array and retrieving them*

```
<?php
$paper[] = "Copier";
```

```
$paper[] = "Inkjet";  
$paper[] = "Laser";  
$paper[] = "Photo";  
for ($j = 0 ; $j < 4 ; ++$j)  
echo "$j: $paper[$j]<br>";  
?>
```

- This example prints out the following:

```
0: Copier  
1: Inkjet  
2: Laser  
3: Photo
```

### Associative Arrays

- Keeping track of array elements by index works just fine, but can require extra work in terms of remembering which number refers to which product. It can also make code hard for other programmers to follow. This is where **associative arrays** come into their own.
- Using them, you can **reference the items in an array by name** rather than by number. Example 6-4 expands on the previous code by giving each element in the array an identifying name and a longer, more explanatory string value.
- *Example 6-4. Adding items to an associative array and retrieving them*

```
<?php  
$paper['copier'] = "Copier & Multipurpose";  
$paper['inkjet'] = "Inkjet Printer";  
$paper['laser'] = "Laser Printer";  
$paper['photo'] = "Photographic Paper";  
echo $paper['laser'];  
?>
```

- Each item now has a unique name that you can use to reference it elsewhere, as with the **echo** statement—which simply prints out Laser Printer. The names (**copier**, **inkjet**, and so on) are called

*indexes* or *keys*, and the items assigned to them (such as **Laser Printer**) are called *values*.

- This very powerful feature of PHP is often used when you are extracting information from XML and HTML. For example, an HTML parser such as those used by a search engine could place all the elements of a web page into an associative array whose names reflect the page's structure:

```
$html['title'] = "My web page";  
$html['body'] = "... body of web page ...";
```

- The program would also probably break down all the links found within a page into another array, and all the headings and subheadings into another. When you use associative rather than numeric arrays, the code to refer to all of these items is easy to write and debug.

### **Assignment Using the array Keyword**

- A compact and faster assignment method uses the array keyword. Example 6-5 shows both a numeric and an associative array assigned using this method.

- *Example 6-5. Adding items to an array using the array keyword*

```
<?php  
$p1 = array("Copier", "Inkjet", "Laser", "Photo");  
echo "p1 element: " . $p1[2] . "<br>";  
$p2 = array('copier' => "Copier & Multipurpose",  
'inkjet' => "Inkjet Printer",  
'laser' => "Laser Printer",  
'photo' => "Photographic Paper");  
echo "p2 element: " . $p2['inkjet'] . "<br>";  
?>
```

- The first half of this snippet assigns the old, shortened product descriptions to the array **\$p1**. There are four items, so they will occupy slots 0 through 3. Therefore, the **echo** statement prints out the following:

**p1 element: Laser**

- The second half assigns associative identifiers and accompanying longer product descriptions to the array `$p2` using the format *key => value*.
- The use of `=>` is similar to the regular `=` assignment operator, except that you are assigning a value to an *index* and not to a *variable*.
- The index is then inextricably linked with that value, unless it is assigned a new value.
- The `echo` command therefore prints out this:  
p2 element: Inkjet Printer
- You can verify that `$p1` and `$p2` are different types of array, because both of the following commands, when appended to the code, will cause an **Undefined index** or **Undefined offset** error, as the array identifier for each is incorrect:  
echo `$p1['inkjet'];` // Undefined index  
echo `$p2[3];` // Undefined offset

### The `foreach...as` Loop

- Using it, you can step through all the items in an array, one at a time, and do something with them.
- The process starts with the first item and ends with the last one, so you don't even have to know how many items there are in an array.
- *Example 6-6. Walking through a numeric array using `foreach...as`*

```
<?php
$paper = array("Copier", "Inkjet", "Laser", "Photo");
$j = 0;
foreach($paper as $item)
{
echo "$j: $item<br>";
++$j;
}
```

?>

- When PHP encounters a **foreach** statement, it takes the first item of the array and places it in the variable following the **as** keyword; and each time control flow returns to the **foreach**, the next array element is placed in the **as** keyword.
- In this case, the variable `$item` is set to each of the four values in turn in the array `$paper`.
- Once all values have been used, execution of the loop ends. The output from this code is exactly the same as in [Example 6-3](#).

- *Example 6-7. Walking through an associative array using `foreach...as`*

```
<?php
$paper = array('copier' => "Copier & Multipurpose",
'inkjet' => "Inkjet Printer",
'laser' => "Laser Printer",
'photo' => "Photographic Paper");
foreach($paper as $item => $description)
echo "$item: $description<br>";
?>
```

- Each item of the array `$paper` is fed into the key/value pair of variables `$item` and `$description`, from which they are printed out. The displayed result of this code is as follows:

```
copier: Copier & Multipurpose
inkjet: Inkjet Printer
laser: Laser Printer
photo: Photographic Paper
```

- As list function an alternative syntax to `foreach...as`, you can use the `list` function in conjunction with the `each` function, as in [Example 6-8](#).

- *Example 6-8. Walking through an associative array using `each` and `list`*

```
<?php
$paper = array('copier' => "Copier & Multipurpose",
'inkjet' => "Inkjet Printer",
```

```
'laser' => "Laser Printer",
'photo' => "Photographic Paper");
while (list($item, $description) = each($paper))
echo "$item: $description<br>";
?>
```

- In this example, a **while** loop is set up and will continue looping until **each** returns a value of **FALSE**. The **each** function acts like **foreach**: it returns an array containing a key/value pair from the array **\$paper** and then moves its built-in pointer to the next pair in that array.
- When there are no more pairs to return, **each** returns **FALSE**.
- The **list** function takes an array as its argument (in this case, the key/value pair returned by the function **each**) and then assigns the values of the array to the variables listed within parentheses.
- Example of how **list** works where an array is created out of the two strings **Alice** and **Bob** and then passed to the **list** function, which assigns those strings as values to the variables **\$a** and **\$b**.

*Example 6-9. Using the list function*

```
<?php
list($a, $b) = array('Alice', 'Bob');
echo "a=$a b=$b";
?>
```

- The output from this code is as follows:  
a=Alice b=Bob

## Using Array Functions

### is\_array

- Arrays and variables share the same namespace. This means that you cannot have a string variable called **\$fred** and an array also called **\$fred**. If you're in doubt and your code needs to check whether a variable is an array, you can use the **is\_array** function, like this:

```
echo (is_array($fred)) ? "Is an array" : "Is not an array";
```

- Note that if `$fred` has not yet been assigned a value, an **Undefined variable** message will be generated.

### **count**

- To know exactly how many elements there are in your array, particularly if you will be referencing them directly.
- To count all the elements in the top level of an array, use a command such as this:  
`echo count($fred);`
- To know how many elements there are altogether in a multidimensional array, you can use a statement such as the following:  
`echo count($fred, 1);`
- The second parameter is optional and sets the mode to use. It should be either `0` to limit counting to only the top level, or `1` to force recursive counting of all subarray elements too.

### **sort**

- Sorting is so common that PHP provides a built-in function for it.  
`sort($fred);`
- Unlike some other functions, **sort** will act directly on the supplied array rather than returning a new array of sorted elements. It returns **TRUE** on success and **FALSE** on error and also supports a few flags—the main two that you might wish to use force items to be sorted either numerically or as strings, like this:  
`sort($fred, SORT_NUMERIC);`  
`sort($fred, SORT_STRING);`
- To sort an array in reverse order using the **rsort** function:  
`rsort($fred, SORT_NUMERIC);`  
`rsort($fred, SORT_STRING);`

### **shuffle**

- There may be times when you need the elements of an array to be put in random order, such as when you're creating a game of playing cards:  
`shuffle($cards);`
- Like `sort`, `shuffle` acts directly on the supplied array and returns `TRUE` on success or `FALSE` on error.

### **explode**

- Useful function with which you can take a string containing several items separated by a single character (or string of characters) and then place each of these items into an array.
- One handy example is to split up a sentence into an array containing all its words, as in Example 6-12.
- *Example 6-12. Exploding a string into an array using spaces*

```
<?php
$temp = explode(' ', "This is a sentence with seven words");
print_r($temp);
?>
```

- This example prints out the following (on a single line when viewed in a browser):

```
Array
(
[0] => This
[1] => is

[2] => a
[3] => sentence
[4] => with
[5] => seven
[6] => words
)
```

- The first parameter, the delimiter, need not be a space or even a single character.
- Example 6-13 shows a slight variation.

*Example 6-13. Exploding a string delimited with \*\*\* into an array*

```
<?php
$temp = explode('***', "A***sentence***with***asterisks");
print_r($temp);
?>
```

- The code in Example 6-13 prints out the following:

```
Array
(0 => A
1 => sentence
2 => with
3 => asterisks
)
```

#### **extract**

- Sometimes it can be convenient to turn the key/value pairs from an array into PHP variables. One such time might be when you are processing the `$_GET` or `$_POST` variables sent to a PHP script by a form.
- When a form is submitted over the web, the web server unpacks the variables into a global array for the PHP script.
- If the variables were sent using the GET method, they will be placed in an associative array called `$_GET`; if they were sent using POST, they will be placed in an associative array called `$_POST`.
- Sometimes you just want to store the values sent into variables for later use. In this case, you can have PHP do the job automatically:  
`extract($_GET);`
- So, if the query string parameter `q` is sent to a PHP script along with the associated value `Hi there`, a new variable called `$q` will be created and assigned that value.

- Be careful with this approach, though, because if any extracted variables conflict with ones that you have already defined, your existing values will be overwritten.
- To avoid this possibility, you can use one of the many additional parameters available to this function, like this:

```
extract($_GET, EXTR_PREFIX_ALL, 'fromget');
```

In this case, all the new variables will begin with the given prefix string followed by an underscore, so \$q will become \$fromget\_q.

## Compact

- At times you may want to use **compact**, the inverse of **extract**, to create an array from variables and their values.
- *Example 6-14. Using the compact function*

```
<?php
$name = "Doctor";
$sname = "Who";
$planet = "Gallifrey";
$system = "Gridlock";
$constellation = "Kasterborous";
$contact = compact('fname', 'sname', 'planet', 'system', 'constellation');
print_r($contact);
?>
```

- The result of running Example 6-14 is as follows:

```
Array
(
  [fname] => Doctor
[sname] => Who
[planet] => Gallifrey
[system] => Gridlock
[constellation] => Kasterborous
)
```

- Note how **compact** requires the variable names to be supplied in quotes, not preceded by a \$ symbol. This is because **compact** is looking for a list of variable names, not their values.

- Another use of this function is for debugging, when you wish to quickly view several variables and their values, as in [Example 6-15](#).

- *Example 6-15. Using compact to help with debugging*

```
<?php
$j = 23;
$temp = "Hello";
$address = "1 Old Street";
$age = 61;
print_r(compact(explode(' ', 'j temp address age')));
?>
```

- This works by using the `explode` function to extract all the words from the string into an array, which is then passed to the `compact` function, which in turn returns an array to `print_r`, which finally shows its contents.
- If you copy and paste the `print_r` line of code, you only need to alter the variables named there for a quick printout of a group of variables' values.
- In this example, the output is shown here:

```
Array
(
  [j] => 23
  [temp] => Hello
  [address] => 1 Old Street
  [age] => 61
)
```

## **reset**

- When the `foreach...as` construct or the `each` function walks through an array, it keeps an internal PHP pointer that makes a note of which element of the array it should return next.
- If your code ever needs to return to the start of an array, you can issue `reset`, which also returns the value of that element.
- Examples of how to use this function are as follows:

```
reset($fred); // Throw away return value
$item = reset($fred); // Keep first element of the array in $item
end
```

- As with `reset`, you can move PHP's internal array pointer to the final element in an array using the `end` function, which also returns the value of the element, and can be used as in these examples:

```
end($fred);
$item = end($fred);
```

For any doubt contact 9873961590