OBJECT-ORIENTED PHP

Concepts, Techniques, and Code

by Peter Lavin

NO STARCH PRESS

San Francisco

8

USING THE PAGENAVIGATOR CLASS

In this chapter we'll use the `PageNavigator` class to step through a directory of images reduced on the fly using the `ThumbnailImage` class. We'll use all three of the classes you have developed so far:

- The `DirectoryItems` class stores a list of filenames of images.
- The `ThumbnailImage` class reduces the dimensions of each image.
- The `PageNavigator` class steps through these images in an orderly fashion.

We'll also look at how to use CSS classes to adjust the appearance of the page navigator; this will greatly improve the reusability of the class. (This isn't directly related to object-oriented programming [OOP], but if a class's appearance cannot blend with various different designs, then its usefulness—and reusability—is greatly compromised. A web development language should integrate well with other web technologies.)

DirectoryItems Change

Fortunately, because the list of images in the `DirectoryItems` class is an array, you can use the ready-made PHP function to return a portion of an array—`array_slice`. All you need to do is wrap this function inside a method. Here is the additional method you require:

```
public function getFileArraySlice($start, $numberitems){
    return array_slice($this->filearray, $start, $numberitems);
}
```

The `$start` variable passed to this method performs the same function as the `start` variable in the Google query string discussed in Chapter 7. The `$numberitems` variable sets the number of items you wish to display per page. In a way, the entire `PageNavigator` class is an answer to the question, "How can you pass values to the `getArraySlice` method so that you can step through the list of images in an orderly fashion?"

CSS and Reusability

No matter how reusable an object is, it won't be reused if it can't be adapted to fit to a variety of page designs. Unlike the `DirectoryItems` class, which does its work on the server, the navigator *is* client-side HTML—it is a series of enabled or disabled hyperlinks. It's important to control the page navigator's appearance, because it's a component of a web page. Figure 8-1 shows that page navigator again.

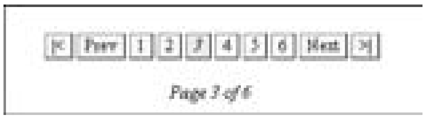

Figure 8-1: The page navigator

Recall that in order to control the navigator's appearance, you wrapped it in a `div` tag and set the class attribute of the `div` tag to `navigator`. One way to display this component is to shrink the font by setting the `font-size` property to `smaller` and to use the `text-align` property to center the text. Here's how the CSS code to produce that effect might look:

```
div.navigator{
    font-size:smaller;
    padding:5px;
    text-align:center;
}
```

This CSS code will ensure that the navigator is centered and that the font size of its buttons is smaller than the surrounding text.

The div tag of the class, totalpagesdisplay, manipulates the appearance of the total page count in the following way:

```
div.totalpagesdisplay{
    ❶font-style:italic;
    ❷font-size:8pt;
    text-align:center;
    ❸padding-top:15px;
}
```

A different ❶ font style and ❷ size are appropriate for displaying the current page and the total page count (*page 3 of 6*, as shown in Figure 8-1). Increased ❸ padding at the top separates the page number display from the navigator proper, which improves readability.

You'll make the anchor tags within your navigator distinctive by assigning style characteristics to them. Because the inactive spans will share some of those characteristics, you can define them here as well. Those shared properties might look something like the following:

```
.navigator a, span.inactive{
    margin-left:0px;
    border-top:1px solid ❶#999999;
    border-left:1px solid ❶#999999;
    border-right:1px solid ❷#000000;
    border-bottom:1px solid ❷#000000;
    padding: 0px 5px 2px 5px;
}
```

Using ❶ a lighter color for the top and left borders and then ❷ a darker color for the bottom and right borders outlines the links and creates the illusion of depth.

Assign properties to the anchor pseudo-classes in order to override the default behavior—they should be different from other anchors on this page:

```
.navigator ❶a:link, .navigator ❶a:visited,
         .navigator ❶a:hover,.navigator ❶a:active{
    color: #3300CC;
    background-color: #FAEBF7;
    text-decoration: none;
}
```

Because these hyperlinks look like buttons, it makes sense to assign the same characteristics to each of the different states represented by ❶ the pseudo-classes: link, visited, hover, and active.

Finally, you differentiate inactive links from active ones by changing the background and the font style. For example, in Figure 8-1, because page 3 is the current page, it is disabled and has a gray background and italic font style.

```
span.inactive{
    background-color :#EEEEEE;
    font-style:italic;
}
```

You can, of course, style your own navigator much differently, using different CSS styles and, really, that's the whole point.

Paging with Class

In Chapter 6, we created a web page to loop through a directory of images and display a thumbnail of each image. We're going to do the same thing here, but this time we'll incorporate the page navigator in order to display a limited number of images per page.

The very first thing you need to do is include the classes you'll be using. This is done with two `require` statements:

```
require 'PageNavigator.php';
require 'DirectoryItems.php';
```

The `PERPAGE` variable defines how many images to display on each page. Define it as a constant (`5`), because it is used in a number of different places on this page and you don't want to change its value accidentally:

```
//max per page
define("PERPAGE", 5);
```

Recall that within the `PageNavigator`, the variable called `$firstparam` is assigned a default value of `offset`—the name for the first name/value pair of the query string associated with the URL of each hyperlink in the navigator. Each page needs to retrieve the `offset` value in order to determine which group of images to display:

```
//name of first parameter in query string
define(❶"OFFSET", "offset");
/*get query string - name should be same as first parameter name
passed to the page navigator class*/
$offset = @$_GET[OFFSET];
```

Like `PERPAGE`, ❶ `OFFSET` is defined as a constant because you do not want its value to change. You want to ensure that the variable you're requesting matches the variable passed into this page by the navigator.

You also want the flexibility to open this page even when no query string has been passed. For this reason, you should check the value of `$offset`:

```
//check variable
if(!isset($offset)){
    ❶$totaloffset = 0;
}
else{
    //then calculate offset
    $totaloffset = $offset * ❷PERPAGE;
}
```

If no query string is passed into the page, you want to begin displaying images at the beginning of your list, so you set ❶ `$totaloffset` to `0`. If `$offset` does have a value, multiplying it by ❷ the `PERPAGE` value calculates the start position within the array of image filenames.

The name of the directory you want to use is assigned to the variable `$directory`:

```
$directory = "graphics";
$di = new DirectoryItems($directory);
```

Listing 8-1: Hard-coded directory name

Because you want to display the directory of images in the `graphics` directory, pass the value `graphics` to the constructor of the `DirectoryItems` class.

The `imagesOnly` method filters out all non-images, and the method `naturalCaseInsensitiveOrder` ignores case and orders numerically where appropriate.

```
$di->imagesOnly();
$di->naturalCaseInsensitiveOrder();
```

In Chapter 6, when you displayed all the thumbnails on one page, you retrieved the entire list of filenames from the `DirectoryItems` class instance. Since the page navigator controls the starting position and since you can retrieve a slice of the array, you need only retrieve a specific number of items here. Do this by passing the `getArraySlice` method a start position and the number of items you wish to display.

```
//get portion of array
$filearray = $di->getFileArraySlice($totaloffset, PERPAGE);
```

Displaying an Array Slice

You retrieve each filename and pass it to the `getthumb.php` file so it can serve as the file source for an `img` tag. You don't need to make any changes to the version of the `getthumb.php` file you used in Chapter 6—it includes the `ThumbnailImage` class and uses it to create a reduced image.

The code to loop through the thumbnail images hasn't changed from Chapter 6 either. For ease of reference, it's reproduced in Listing 8-2.

```
echo "<div style=\"text-align:center;\">";
echo "Click the file name to view full-sized version.<br />";
$path = "";
//specify size of thumbnail
$size = 100;
foreach (❶$filearray as $key => $value){
    $path = "$directory/".$key;
    /*errors in getthumb or in class will result in broken links
    - error will not display*/
    echo "<img src=\"getthumb.php?path=$path&amp;size=$size\" ".
    "style=\"border:1px solid black;margin-top:20px;\" ".
    "alt= \"$value\" /><br />\n";
    echo "<a href=\"$path\" target=\"_blank\" >";
    echo "Title: $value</a> <br />\n";
}
echo "</div><br />";
```

Listing 8-2: Code to loop through thumbnail images

This code differs from the code in Chapter 6 only in that ❶ the `$filearray` variable that contains the image filenames is the portion of the total array retrieved by the `getArraySlice` method and not all the filenames.

Creating the PageNavigator Object

In order to create the page navigator, you need the current page name and also the total number of image files; the global `$_SERVER` array supplies the name of the current page and `getCount` the total number of images.

```
$pagename = basename($_SERVER["PHP_SELF"]);
$totalcount = $di->getCount();
```

You only need to create the navigator if there is more than one page, so calculate that number first, as shown in the code in Listing 8-3.

```
$numpages = ceil($totalcount/PERPAGE);
//create if needed
if($numpages > 1){
    //create navigator
    $nav = new PageNavigator(❶$pagename, $totalcount, PERPAGE, $totaloffset);
    //is the default but make explicit
    ❷$nav->setFirstParamName(OFFSET);
    echo ❸$nav->getNavigator();
}
```

Listing 8-3: Creating the navigator if there's more than one page

When constructing the PageNavigator instance, you pass it ❶ the four required parameters and let the two additional parameters—$maxpagesshown and $params—default to 4 and an empty string, respectively. This means that the navigator will show links to a maximum of four pages and that there are no additional name/value pairs for the query string. (As promised in Chapter 7, you'll learn more about $params in Chapter 9. However, you may already have surmised that this variable can be used to replace the hard-coded directory name given in Listing 8-1.)

You do not need to set ❷ the first parameter name; it has a default value of offset. However, by setting the name here, you make it clear that this is the name of the one required name/value pair, and that it can be changed if desired.

Finally, the HTML code that makes up ❸ the navigator is returned and displayed in the web page.

Where to Go from Here

Using the PageNavigator class solves two problems: it alleviates the demand on server resources and it improves the aesthetics of the web page display. Only a limited number of images are displayed at any one time, thus reducing the demands on the server. Aesthetic requirements are satisfied by reduced web page length.

As noted on many occasions, the real value of objects is in their reusability. Through the use of CSS, you're able to adjust the appearance of the page navigator to match it to a variety of situations. By using span and div tags, you can manipulate the look and feel of the navigator so that it blends easily with any design. The number of items shown on each page and the number of pages accessible at any one time can be set to any number desired.

We've seen that the PageNavigator class's design is adaptable and that you can use it to step through an array of images, but what about its use in other situations? A navigator is much more commonly required with database queries that return a large number of records. In the next chapter, we'll develop a database class and then see how well the PageNavigator class can handle a large result set. The ability to reuse the navigator class in various and different circumstances will be a true test of its robustness.