



Professional **PHP4** **Multimedia** **Programming**

Allan Kent, Devon H. O'Dell, Andy Chase, Jan Rosa,
Sanjay Abraham, Iman S.H. Suyoto, Kapil Apshankar



Wrox technical support at: support@wrox.com

Updates and source code at: www.wrox.com

Peer discussion at: p2p.wrox.com

Summary of Contents

Preface		1
Chapter 1:	PHP Multimedia Installation	13
Chapter 2:	PHP Refresher	25
Chapter 3:	Ming Fundamentals	71
Chapter 4:	Ming with ActionScript	105
Chapter 5:	Manipulating Images with GD	143
Chapter 6:	ImageMagick	193
Chapter 7:	PDFs with PDFlib	217
Chapter 8:	FDF – Sending Data To and From PDF Forms	283
Chapter 9:	Case Study – Ming Headline Grabber	333
Chapter 10:	Case Study – Using GD on WAP Sites and PDA	359
Chapter 11:	Case Study – Image Gallery	399
Chapter 12:	Case Study: PDF Template System	421
Appendix A:	PHP Ming Language Reference	463
Appendix B:	PHP GD Language Reference	475
Appendix C:	HAWHAW Reference	483
Appendix D:	PHP PDF Language Reference	501
Appendix E:	PHP FDF Language Reference	513
Appendix F:	PHP and ImageMagick Language Reference	517
Appendix G:	Color Screenshots Gallery	531
Appendix H:	Setting Up Palm Desktop, POSE, and AvantGo on a Windows Desktop	547
Index		557

6

ImageMagick

In the first two chapters we refreshed our PHP programming skills. We then went on to look at how we could create dynamic Flash content using Ming and how we could add interactivity to our Flash movies using ActionScript. During the course of these chapters we saw how we could use the basic drawing methods of Ming to draw simple shapes. We also saw how we could incorporate existing images within our Flash content. But what happens if we have an existing image that we want to modify? This is where **ImageMagick** comes in.

What is ImageMagick?

Written by ImageMagick Studio (<http://www.imagemagick.org>), ImageMagick is a collection of tools to modify and gather data about images, and also an API to functions that perform those tasks. It allows manipulation of over a hundred different image and document formats and at the same time allows us to apply complex transformations and filters to images.

Though ImageMagick can work with a large variety of document formats, since we are working in a web environment, we will focus only on JPEG (Joint Photographic Experts Group) and PNG (Portable Network Graphics) formats.

In this chapter will focus on ImageMagick tools that allow us to manipulate web images – **Convert** (converts images in different formats), **Mogrify** (transforms images), and **Identify** (identifies the format and characteristics of an image).

In this chapter, we will cover:

- ❑ Installing ImageMagick
- ❑ Some of the tools distributed with the ImageMagick distribution

- ❑ How to use these tools with PHP
- ❑ How to create a class library that interfaces with these tools
- ❑ How to implement the class library functions

Getting Started

ImageMagick can be used on a number of platforms. Besides the usual Windows and UNIX versions, there are binaries available for Macintosh and VMS (Virtual Memory System) operating systems as well. The sourcecode is available for ImageMagick, so if we wish, we can compile it ourselves for our chosen platform. Here we will look at installing from binaries on a Windows system and compiling from source on a UNIX system.

Installing on UNIX

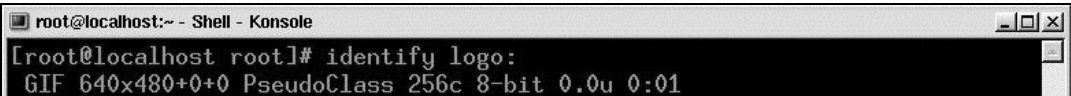
For UNIX we will install ImageMagick from source. Download the latest sourcecode from one of the sites listed on the ImageMagick web site (<http://www.imagemagick.org/>). Here we will use the ImageMagick-5.4.6-3.tar.gz file. All that we need to do is extract, configure, and compile the source. Use the following commands at the system prompt:

1. `tar -zxvf ImageMagick-5.4.6-3.tar.gz`
2. `./configure --prefix=/usr/local`
3. `make`
4. `make install`

The configure program will determine what our system architecture is and what libraries we have installed. The binaries will be installed to `/usr/local/bin`. We can now test that the binaries have installed correctly and that `/usr/local/bin` is in our path by issuing the command:

```
# identify logo:
```

which gives this result:

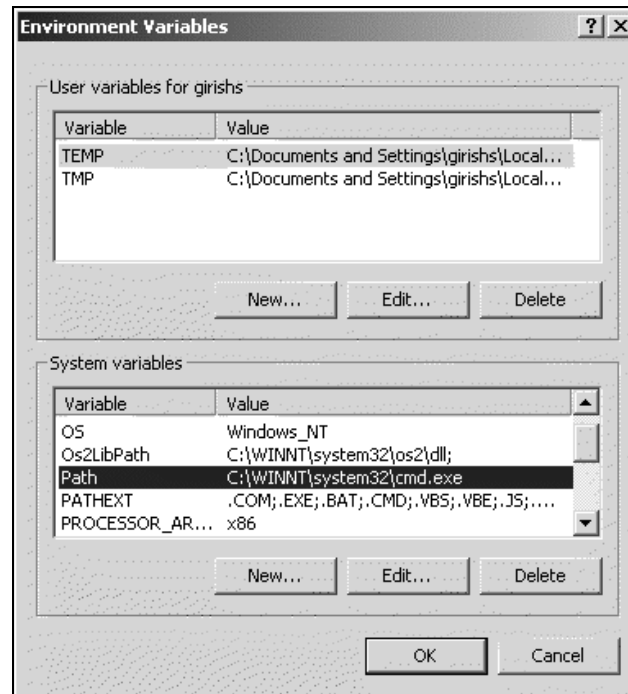
A terminal window titled "root@localhost:~ - Shell - Konsole" showing the command `[root@localhost root]# identify logo:` and its output: `GIF 640x480+0+0 PseudoClass 256c 8-bit 0.0u 0:01`.

```
root@localhost:~ - Shell - Konsole
[root@localhost root]# identify logo:
GIF 640x480+0+0 PseudoClass 256c 8-bit 0.0u 0:01
```

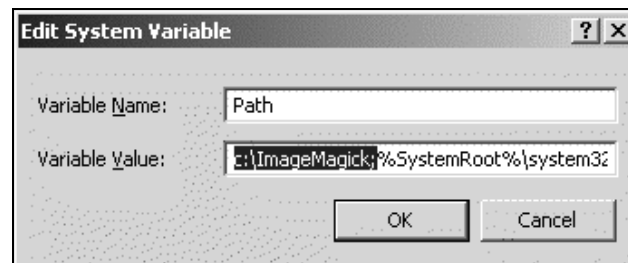
Installing on Windows

Download the Windows binary distribution (ImageMagick-win2k.zip) from one of the sites listed on the ImageMagick web site. Unzip the contents into a folder on the hard drive. It is always better to keep all the utility programs at the same place, for example, `c:\bin\ImageMagick`. Since we want PHP to have access to the ImageMagick tools, we have to add the ImageMagick folder to the Path environment variable.

In Windows 2000 or Windows NT, we can alter this variable by right-clicking My Computer, and selecting **Properties** from the menu, and then selecting the **Advanced** option. Click the **Environment Variables** button to bring up the following dialog box:



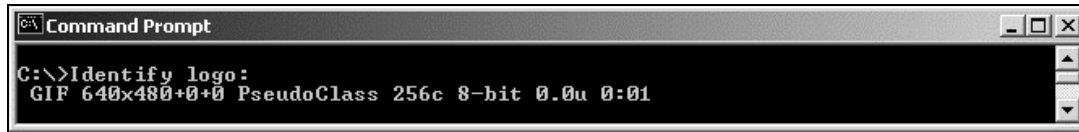
Choose the **Path** option under **System variables**, click **Edit**, and add the full path to ImageMagick and a semicolon to the beginning of the existing value:



Click **OK** on all of the open dialogs to accept the change. To test whether our installation was successful or not, open the **Command Prompt** and type the following command:

```
C:\>Identify logo:
```

to see this output:



Windows NT has a program called `convert.exe` that converts files to NTFS file systems; since we don't want this tool to run on our images we have added the ImageMagick folder at the beginning of the environmental variable path.

ImageMagick Tools

ImageMagick tools are used to access the ImageMagick functions directly from a command-line prompt. The tools that we will be using in this chapter are:

- ☐ Convert
- ☐ Mogrify
- ☐ Identify

For a full list of the tools available in ImageMagick, refer to <http://www.imagemagick.org/www/tools.html>.

Convert

The `Convert` tool converts images from one format to another, while optionally passing the image through filters, and produces no output message on success. Converting between file formats is very straightforward – we just have to specify the input and output filenames. The `Convert` tool will use the extension of the output file name to determine the output format:

```
Convert glitter.jpg glitter.png
```

The ImageMagick library contains filters that can be used to add sound to images, remove pixel aliasing, add blurs, change image sizes, and even simulate a charcoal drawing.

Suppose we want to change the size of an image, for example, let's say from 320x240 pixels to 160x120 pixels, using this command:

```
convert -geometry "160x120!" -draw "text 0,0 'glitter'"  
-negate glitter.jpg glitter.png
```

Here, we add text that says `glitter` at point (0, 0), and make a negative image, we then change the format again from JPEG to PNG.

The Convert utility has some pretty nifty functions that can be used on our images:

- ☐ Add a Gaussian blur
- ☐ Colorize the image
- ☐ Add comments to it
- ☐ Crop, emboss, rotate, or sharpen the image

We can even make our own online image manipulation studio application using the Convert utility.

Mogrify

Like Convert, Mogrify also gives no output message if successful. The Mogrify tool works mostly the same way as the Convert tool, the only exception being that Mogrify works on the input image and doesn't save the image to another file. The Mogrify utility can be used to do the same tasks as the Convert utility but without changing the output file type, like this:

```
mogrify -geometry '160x120!' -draw "text 0,0 'glitter'" -negate glitter.jpg
```

It is also possible to change the file format with the Mogrify utility. To do this, we use the `format` option and specify the target format. However, the specified target format should be one of the ImageMagick-supported formats. We will see how we can do this when we create our class later on.

Some of the file formats require additional libraries and support files to be installed before ImageMagick can use them. For example, we will need the **FreeType** software in order to add annotations to images. On Windows machines, where no X server is available, ImageMagick will use **GhostScript** to annotate images.

For more information refer to the listing at <http://www.imagemagick.org/www/install.html#extr>.

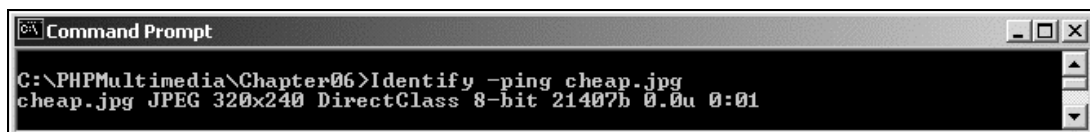
Identify

The Identify tool reads and outputs information about a specified image, and has some options to specify image geometries for images that are raw CMYK, RGB, or GREY values.

Here's an example that shows the usage and the output:

```
Identify -ping cheap.jpg
```

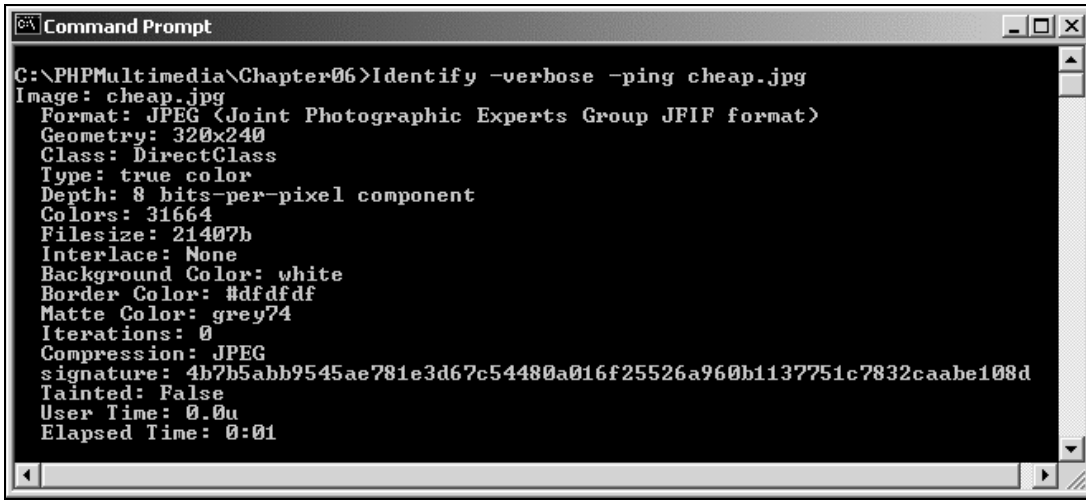
Here is the output:



And if we use:

```
$ Identify -verbose -ping cheap.jpg
```

This is the output:



```
Command Prompt
C:\PHPMultimedia\Chapter06>Identify -verbose -ping cheap.jpg
Image: cheap.jpg
Format: JPEG (Joint Photographic Experts Group JFIF format)
Geometry: 320x240
Class: DirectClass
Type: true color
Depth: 8 bits-per-pixel component
Colors: 31664
Filesize: 21407b
Interlace: None
Background Color: white
Border Color: #dfdfdf
Matte Color: grey74
Iterations: 0
Compression: JPEG
signature: 4b7b5abb9545ae781e3d67c54480a016f25526a960b1137751c7832caabe108d
Tainted: False
User Time: 0.0u
Elapsed Time: 0:01
```

As we can see, using the `-ping` option with `Identify` is more efficient, as it does not read the entire image into memory. Although the load and CPU time taken by this program is already minimal, it is recommended for servers with heavier loads.

Now that we have learned how to use the ImageMagick command-line tools, let's look at how we can use them within our PHP scripts.

Using ImageMagick with PHP

At the time of writing, there is no PHP extension library that interfaces with ImageMagick's API. Therefore, we must execute the small and efficient pre-built programs that come with the ImageMagick distribution. Work has begun on an extension using **PECL** (PHP Extension C Layer), but it is not yet stable enough to be used in a working environment. The extension manual can be found at <http://pear.php.net/manual/en/pecl.imagick.php>.

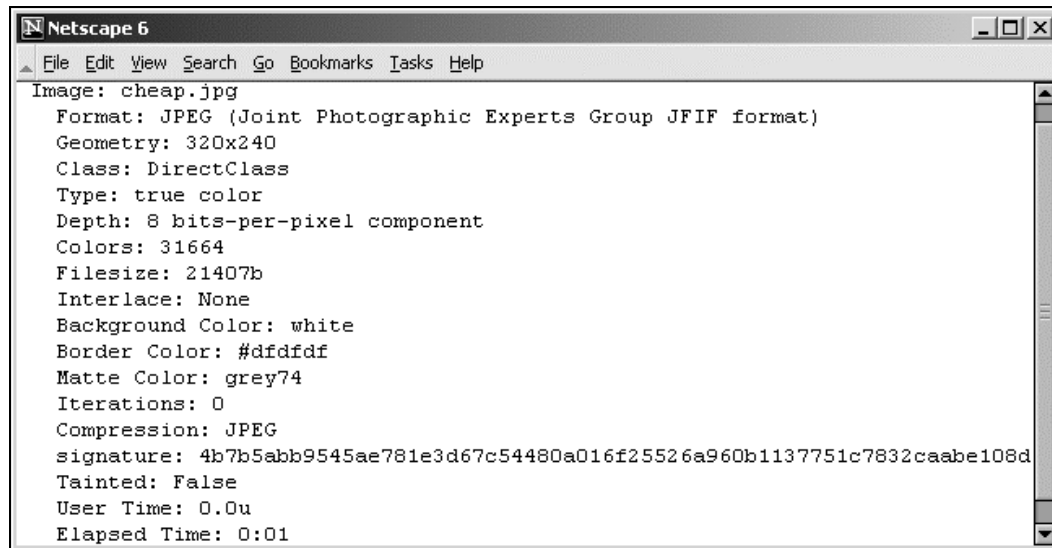
Currently, the way that we execute these tools is by calling them directly from the PHP script. There are a number of ways in which we can do this in PHP, but not all of them will work in this case. Remember when we looked at the `Identify` tool – it returned a number of lines detailing the image format. Obviously we would want all of that information to be returned to us. The PHP function that will do this is the `shell_exec()`. It takes a single string argument of the `shell` command to run, and returns the complete output of that command. There is a shortcut for this, known as the execution operator or backtick. The backtick looks like a single quote, but slants backwards like a backslash.

Let's take a look at an example to illustrate this. Save the following code as `backtick.php`. We will also have to have a JPEG file called `cheap.jpg` in the same folder as the script.

```
<pre>
<?php
    $output = `identify -verbose cheap.jpg`;
    print $output;
?>
</pre>
```

Nothing complicated here – all we're doing is using the backtick operator to execute the command and have the output of the command stored in the `$output` variable.

This is the output of the script:



The ImageMagick Class Library

So far we have seen what we can do with ImageMagick and how we can interface it with PHP. Now we can create a class to interface with the command-line tools. Our class will allow us to easily use the Convert, Identify, and Mogrify tools.

All the options provided by the Convert, Mogrify, and Identify tools are explained in great detail inside their respective 'man' pages (<http://www.imagemagick.org>). The options that we will be using in this chapter and in the Case Study that follows later in the book will be explained here though.

Save this file as `class.imagemagick.php`:

```
<?php

/*****
class.imagemagick.php
```

```

Class library interface to the ImageMagick toolkit accessing
Convert, Identify, and Mogrify
/*****

class ImageMagick
{

```

First, we will set some variables that will help us with basic error handling:

```

var $errorLevel = 0;
var $errorExitLevel = 2;
var $lastError = '';

```

Next, we initialize an array that provides a list of available image file formats for the user to convert to. ImageMagick can recognize and output these document types:

```

var $imageTypeList = array

("AVS" => "AVS X image file", "BIE" => "Joint Bi-level
      Image Experts Group file interchange format",
"BMP" => "Microsoft Windows bitmap image file",
"BMP24" => "Microsoft Windows 24-bit bitmap image file",
"CGM" => "Computer Graphics Metafile",
"DCX" => "ZSoft IBM PC multi-page Paintbrush file",
"DIB" => "Microsoft Windows bitmap image file",
"DICOM" => "Medical image file",

"EPDF" => "Encapsulated Portable Document Format",
"EPI" => "Adobe Encapsulated PostScript Interchange format",
"EPS" => "Adobe Encapsulated PostScript file",
"EPS2" => "Adobe Level II Encapsulated PostScript file",
"EPSF" => "Adobe Encapsulated PostScript format",
"EPSI" => "Adobe Encapsulated PostScript Interchange format",
"EPT" => "Adobe Encapsulated PostScript Interchange
      format with TIFF preview",
"FAX" => "Group 3",
"FIG" => "TransFig image format",
"FITS" => "Flexible Image Transport System",
"FPX" => "FlashPix Format",
"GIF" => "CompuServe graphics interchange format; 8-bit color",
"GIF87" => "CompuServe graphics interchange format;
      8-bit color (version 87a)",
"HDF" => "Hierarchical Data Format",
"HPGL" => "HP-GL plotter language",

"JBIG" => "Joint Bi-level Image experts Group file interchange
      format",
"JPEG" => "Joint Photographic Experts Group JFIF format;
      compressed 24-bit color",
"ICO" => "Microsoft icon",
"LABEL" => "text image",
"MIF" => "Magick image file format",

```

```

"MNG" => "Multiple-image Network Graphics",
"MONO" => "Bi-level bitmap in least-significant byte (LSB)
          first order",
"MPEG" => "Motion Picture Experts Group file interchange format",
"M2V"  => "Motion Picture Experts Group file interchange format
          (version 2)",

"NETSCAPE" => "Netscape 216 color cube",
"NULL" => "NULL image",
"PBM" => "Portable bitmap format (B&W)",
"PCD" => "Photo CD, maximum resolution of 512x768",
"PCDS" => "Photo CD. Decode with sRGB color table",
"PCL" => "Page Control Language",
"PCX" => "ZSoft IBM PC Paintbrush file",
"PDF" => "Portable Document Format",
"PGM" => "Portable graymap format (grayscale)",
"PICT" => "Apple Macintosh QuickDraw/PICT file",
"PIX" => "Alias/Wavefront RLE image format",
"PNG" => "Portable Network Graphics",
"PNG" => "Portable Network Graphics",
"PNM" => "Portable anymap",
"PPM" => "Portable pixmap format (color)",
"PS"  => "Adobe PostScript file",
"PS2" => "Adobe Level II PostScript file",
"RAD" => "Radiance image file",
"RLA" => "Alias/Wavefront image file (read-only)",
"RLE" => "Utah Run length encoded image file (read-only)",
"SGI" => "Irix RGB image file",
"SUN"  => "SUN Rasterfile",
"SVG"  => "Scalable Vector Graphics",

"TGA" => "Truevision Targa image file",
"TIFF" => "Tagged Image File Format",
"TIFF32" => "24-bit Tagged Image File Format",
"TIM"  => "PSX TIM file",
"TTF"  => "TrueType font file",
"UIL"  => "X-Motif UIL table",
"VID"  => "Visual Image Directory",
"VIFF" => "Khoros Visualization image file",
"XBM"  => "X Windows system bitmap, black and white only",
"XPM"  => "X Windows system pixmap file (color)",
"XWD"  => "X Windows system window dump file (color)";

```

The `$resizeFlag` variable is used to pass different values to `Convert` and `Mogrify`, to specify when, or when not to resize the manipulated image by using the `-geometry` option. When we create a new object, we are going to grab the image properties and store them for our functions to use later:

```
var $resizeFlag;
```

Let's declare the variables:

```
var $imageBits;  
var $imageBytes;  
var $imageChannels;  
var $imageClass;  
var $imageHeight;  
var $imageName;  
var $imageType;  
var $imageWidth;
```

Each of these variables describes the properties of the image initialized when the library is called.

Name	Description
\$imageBits	Describes the bits per pixel in each image (usually 8 or 16)
\$imageBytes	Provides the file size of the image
\$imageChannels	Contains the number of channels in the image
\$imageClass	Describes the class of the image (can be either 'DirectClass' or 'PseudoClass')
\$imageHeight	Height of the image (\$imageWidth provides the image width)
\$imageName	Stores the name of the image (if specified, when the class is initialized, full path)
\$imageType	Stores the type of the image (JPEG, or PNG for example)

The `$imageComments` variable is not initialized inside the constructor of the class library (it is not necessary for any of the ImageMagick functions). However, a class variable is reserved since the image comments are part of the image's headers, and are useful when describing an image in the user interface:

```
var $imageComments;
```

If we are working on a machine where we do not have administrative rights and ImageMagick is not in our path, we specify the full path of the folder where we have installed ImageMagick, like this:

```
var $imageMagickPath = 'c:/bin/ImageMagick/';
```

ImageMagick()

Now we can create the ImageMagick constructor function. When we create the object we can pass the filename as an optional argument to the constructor; we will do this in our example:

```
function ImageMagick()  
{
```

Test for the number of arguments that have been passed through; if we have only a single argument then we must test to see if the file exists:

```
if (func_num_args() == 1) {
    if (@file_exists(func_get_arg(0))) {
```

If the file exists then we can call `ImageProperties()`. As we will see later, the `ImageProperties()` function sets the variables we declared earlier:

```
        $this->ImageProperties(func_get_arg(0));
    } else {
```

If the filename does not exist then we output an error message using the `Error()` member function:

```
        $filename = func_get_arg(0);
        $this->error
            (2,'Fatal error trying to use file "'. $filename.': file
              doesn't exist or has wrong permissions to be accessed');
    }
```

If the class was initialized without passing a filename, then we can continue creating the object, but we will not call the `ImageProperties()` function. In this case the script creating the `ImageMagick` object will have to set the `$imageName` variable and call `ImageProperties()` function manually:

```
    } elseif(func_num_args() == 0) {
```

If we get any other number of arguments, it means the class has been invoked incorrectly and we generate an error:

```
    } else {
        $this->error
            (2,"Fatal error invoking class ImageMagick: we may use a
              maximum of one argument (an image filename) when
              invoking this class");
    }
} // ImageMagick()
```

That is it for the `ImageMagick` constructor function. The next function that we will create is the `Error()` function.

Error()

This function takes two arguments – the level of the error and the message. Earlier we defined a variable called `$errorExitLevel` that defined a threshold for the type of error that we can get. With certain errors we would want the script to exit but with minor errors or warnings we would like to continue:

```
function Error($level,$message)
{
    switch ($level) {
        case 2 :
            $this->lastError = "<b>Fatal Error:</b> $message<br />";
            break;
        case 1 :
            $this->lastError = "<b>Warning:</b> $message<br />";
            break;
        default :
            $this->lastError = "<b>Information:</b> $message<br />";
    }
}
```

The `$errorLevel` variable holds the current error level of the script. It was initialized to 0 and we want it to be at the highest error level that has been generated. We therefore test to see if the current error level (`$level`) is higher than `$this->errorLevel`, then we update `$this->errorLevel` with `$level`:

```
if ($this->errorLevel <= $level) {
    print $this->lastError;
    flush();
}
```

If the error level is greater than or equal to the threshold we have defined, then we exit from the script:

```
if ($this->errorExitLevel <= $level) {
    exit;
}
} // Error()
```

ImageProperties()

In the constructor function we called the `ImageProperties()` member function. It is the first function that we are creating, which will call one of the ImageMagick tools. As mentioned earlier, the `Identify` tool returns the image properties:

```
function ImageProperties($image)
{
```

Although in the constructor function (`ImageMagick()`) we tested whether the file existed or not, we also said that if no arguments are passed to it then we would have to manually set the `$imageName` variable and call the `ImageProperties()` function. Therefore, there is a possibility that we could call this function without setting the `$imageName` variable, hence we must test again:

```
if (file_exists($this->imageName)) {
    $this->imageName = $image;
```

We call the `Exec()` member function to call the program. This function will use a combination of the path to the ImageMagick installation and the command line, like this:

```
$identify = $this->Exec("identify $this->imageName");
```

The output of `Identify` will be parsed using a regular expression match, but before we do that we need to build the pattern:

```
$match = '#'.preg_quote($this->imageName).' (.+) (\d+)x(\d+)
          (\w+) (.+) (\d+) [b|kb|mb]#';
```

We can then execute the regular expression and store the results in `$matches`:

```
preg_match($match, $identify, $matches);
```

The output from a valid image file parsed with our regular expression will return an array of seven elements. We can now test to see whether we have 7 elements or not:

```
if (sizeof($matches) != 7) {
    $this->Error(2, "This ($this->imageName) is not an image
                    file.");
}
```

We don't have to exit the script here, as the `Error()` function will do that for us. We can now set the member variables to the values returned by `preg_match()`.

```
$this->imageType    = $matches[1];
$this->imageWidth   = $matches[2];
$this->imageHeight  = $matches[3];
$this->imageClass   = $matches[4];
$this->imageBits    = $matches[5];
$this->imageBytes   = $matches[6];
```

We can then clean up any variables that we no longer need:

```
unset($match);
unset($identify);
```

If the file does not exist then we generate an error to that effect:

```
    } else {
        $this->error(2, "File $this->imageName doesn't exist");
    }
} // ImageProperties()
```

We can now start writing the functions that allow us to modify our images. The first set of functions that we will deal with is the `Convert` functions.

Convert()

We will create a `Convert()` function that acts as a raw interface to the `Convert` tool. Here, we will have to provide all of the options for the `Convert` tool:

```
function Convert($outFile, $options)
{
```

The function will take two arguments – the name of the output file and the options that we will be passing to the Convert tool. All we do is build a command line with these arguments and pass them to the `Exec()` member function:

```
$output = $this->Exec("convert $options $this->imageName
                     $outFile");
```

If this is successful, the Convert tool will not generate any message. Therefore, we can test the length of `$output` – if it has something in it, then we get an error message:

```
if (strlen($output) > 0) {
    $this->Error(1, "convert probably didn't finish
                  successfully. Convert says: $output");
    return -1;
}

$this->ImageProperties($outFile);
return 0;
} // Convert()
```

The next set of functions that we will write also uses the Convert tool, but each function will execute only one set of the more popular options that we can pass to the Convert tool. We prefix each of these functions with `C_`.

`C_Comment()`

The first function will allow us to add a comment to the image file. The comment can be any string, except for some `%[letter]` values that are ImageMagick macros. These values are listed in the table below:

Values	Description
<code>%b</code>	Image file size (in bytes)
<code>%c</code>	If the comment is <code>V1.jpg</code> and we <code>-comment</code> to <code>%c v2</code> , we get <code>V1.jpg v2</code>
<code>%d</code>	Directory of image
<code>%e</code>	Filename extension (for example, <code>lalou.jpg</code> to <code>.jpg</code>)
<code>%f</code>	Filename (for example, <code>lalou.jpg</code>)
<code>%h</code>	Image height (for example, <code>1024</code> , representing 1024 pixels)
<code>%i</code>	Input filename
<code>%k</code>	Number of unique image colors
<code>%l</code>	Image label
<code>%m</code>	ImageMagick file format type (for example, JPEG, GIF, PNG, and MNG)
<code>%n</code>	Number of scenes
<code>%o</code>	Output filename

Values	Description
%p	Page number (useful for PostScript, PDF documents, and so on)
%q	Quantum depth
%s	Scene number
%t	Prefix of filename (for example, lalou from lalou.jpg)
%u	Unique temporary filename
%w	Width
%x	x resolution
%y	y resolution
\n	New line
\r	Carriage return

Here is the function:

```
function C_Comment($outFile, $comment)
{
    $this->Convert($outFile, "-comment '$comment'");
} // C_Comment()
```

C_Resize

Another popular action to perform on a file is to resize it. So we pass the new width and height that we want for the image to the function:

```
function C_Resize($outFile, $newWidth, $newHeight)
{
    $this->imageWidth = $newWidth;
    $this->imageHeight = $newHeight;

    $this->Convert($outFile, "-geometry {$newWidth}x{$newHeight}!");
} // C_Resize()
```

The next set of functions interface with the Mogrify tool.

Mogrify()

As we did with the Convert tool, we will create a Mogrify() function that will act as a raw interface to the Mogrify tool, and some functions for the more common options that we can pass to it. Remember the Mogrify tool works with the existing image and does not need an output file to be specified:

```
function Mogrify($options)
{
    $output = $this->Exec("mogrify $options $this->imageName");
    if (strlen($output) > 0) {
```

```
        $this->Error(1,"mogrify probably didn't finish successfully.  
        Mogrify says: $output<br />");  
        return -1;  
    }  
  
    return 0;  
} // Mogrify()
```

ChangeFormat()

This function changes the format of the file, for example, this will allow us to convert from a JPEG format to a PNG format. The format will have to be one of the ImageMagick-supported formats:

```
function ChangeFormat($newFormat)  
{  
    if (isset($this->imageTypeList[strToUpper($newFormat)])) {  
        $this->Mogrify("-format $newFormat");  
    } else {  
        $this->Error(0,"This image format is not supported by  
        ImageMagick");  
        return -1;  
    }  
}  
} // ChangeFormat()
```

If the format is not listed, then we output an error message, which says that the format is not supported.

Resize()

This function will write an interface to resize an image. There are a number of constraints that we can use when resizing an image, and we will have to take these into account. These constraints can be accessed using the \$resizeFlag variable. The function will create the options string and then pass that to the Mogrify() member function:

```
function Resize($newWidth, $newHeight)  
{  
    $this->imageWidth = $newWidth;  
    $this->imageHeight = $newHeight;  
    $this->Mogrify("-geometry '{$newWidth}x{$newHeight}  
        {$this->resizeFlag}'");  
}  
} // Resize()
```

These functions allow us to specify the restraints on the resize. All that they do is set the \$resizeFlag member variable, and call the Resize() function.

```
function Resize_F($newWidth, $newHeight)  
{  
    $this->resizeFlag = "!";  
    $this->Resize($newWidth, $newHeight);  
} // Resize_F
```

The ! character forces the resize to use the exact dimensions we have specified:

```
function Resize_Gt($newWidth, $newHeight)
{
    $this->resizeFlag = ">";
    $this->Resize($newWidth, $newHeight);
} // Resize_Gt()
```

The > character tells the Resize() to resize the image only if the new size is smaller than the existing size – it will only allow us to make the image smaller:

In the same way the < character will tell the Mogrify tool to only resize the image if the new width and height are larger than the existing width and height:

```
function Resize_Lt($newWidth, $newHeight)
{
    $this->resizeFlag = "<";
    $this->Resize($newWidth, $newHeight);
} // Resize_Lt()
```

The @ character specifies the maximum area in pixels, of an image:

```
function Resize_Max($newWidth, $newHeight)
{
    $this->resizeFlag = "@";
    $this->Resize($newWidth, $newHeight);
} // Resize_Max()
```

By specifying a % sign in the options to the Mogrify tool we tell the tool to expect a percentage as the width and height values, rather than a pixel value:

```
function Resize_P($pWidth, $pHeight)
{
    $this->imageWidth = $this->imageWidth * $pWidth / 100;
    $this->imageHeight = $this->imageHeight * $pHeight / 100;
    $this->resizeFlag = "%";
    $this->resize($this->imageWidth, $this->imageHeight);
} // Resize_P()
```

Identify()

The final set of functions interface with the Identify tool:

```
function Identify($options)
{
    $output = $this->Exec("identify $options $this->imageName");
    return $output;
} // Identify()
```

IdentifyPing()

It is best to specify the `-ping` as an option to the `Identify` tool, since the entire file is not read into memory:

```
function IdentifyPing($options)
{
    $output = $this->Exec("identify -ping $options
                        $this->imageName");
    return $output;
} // IdentifyPing()
```

IdentifyVerbose()

Specifying `-verbose` as an option will return a full set of information about the image:

```
function IdentifyVerbose($options)
{
    $output = $this->Exec("identify -verbose $options
                        $this->imageName");
    return $output;
} // IdentifyVerbose()
```

When we ask for verbose image information, one piece of information that is returned is the user-specified comments. The next function uses a regular expression to parse the output and return only the comments.

ReadComments()

Here is the function:

```
function ReadComments()
{
    $output = $this->Exec("identify -verbose -ping
                        $this->imageName");

    $match = "/comment: (.+?)\n signature: /s";
    preg_match($match, $output, $matches);

    //We must make sure that we got a valid comment back:
    if (sizeof($matches) != 2) {
        return $matches[1];
    } else {
        return -1;
    }
} // ReadComments()
```

The final function in our class is the `Exec()` function.

Exec()

This is the member function that is used to call the actual command line. The single argument of the command line along with its options that we created is appended to the full path to the ImageMagick installation and the results returned to the calling function:

```

function Exec($cmd)
{
    $command = $this->imageMagickPath.$cmd;
    return ` $command `;
} // Exec()

} // End of class ImageMagick
?>

```

Implementing the Class Library Functions

In this section, we will look at how to implement the class library through several examples in the following code snippets. We will create a simple page that will allow us to dynamically resize an image, change its format, and apply one of several different effect filters to the image.

Save the following code as `imagemagick.php`:

```

<?php

require_once "class.imagemagick.php";

$path = '';
$ORIGINAL_FILE = "cheap.jpg";

```

First, we check to see whether we have an image name to work with:

```

if (isset($_POST['pName'])) {

```

Here we use the `$_POST` variable as `register_globals` is turned off by default since PHP 4.1; this means `$pName` won't work unless we specifically turn `register_globals` on.

Next, we create a new `ImageMagick` object and pass the filename to it:

```

$iObj = new ImageMagick($path.$_POST['pName']);

```

There are three possible actions that we can perform in this script: `resize`, `convert`, and `produce effects`.

We will first look at resizing the image:

```

if ($_POST['action'] == "resize") {

    $iName = $path.$_POST['tmpName'].'.'.strtolower(
        ($iObj->imageType);

    $iObj->C_Resize($iName, $_POST['width'], $_POST['height']);

    if ($_POST['pName'] != $ORIGINAL_FILE)
        unlink($_POST['pName']);
}

```

Here, first we build a temporary filename based on the tmpName form field with an extension of the file type, then we resize the temporary file. Also, if we are not working with the original image then we can delete the previous temporary file.

Next we will look at converting the image:

```
    } elseif ($_POST['action'] == "convert") {

        $iObj->ChangeFormat(strToLower($_POST['format']));

        $iName = preg_replace('/^([^\.]+)\.(\w+)$/i',
            '\1.' . strToLower($_POST['format']), $iObj->imageName);

        $iObj->ImageProperties($iName);

        if ($_POST['pName'] != $ORIGINAL_FILE)
            unlink($_POST['pName']);
    }
```

The Convert action changes the format of the file. We call our ChangeFormat() member function and pass it the format that we want to change to. We also change the file extension to the new type. Once the image has been converted we re-initialize the image properties and again delete the temporary file.

This section of the script adds effects to the image. The effect to add is selected from a select box - Add button in our form:

```
    } elseif ($_POST['action'] == "effect") {
        $iName = $_POST['tmpName'] . strToLower($iObj->imageType);
        $iObj->Convert($iName, $_POST['effect']);

        if ($_POST['pName'] != $ORIGINAL_FILE)
            unlink($_POST['pName']);
    }
```

If we have not specified an image name, then we must create the image from our existing base image:

```
    } else {
        $iObj = new ImageMagick($ORIGINAL_FILE);
        $iName = $ORIGINAL_FILE;
    }

    ?>
```

Let's now look at the HTML script:

```
<html>

<head>
    <title>ImageMagick Manipulation Examples</title>
</head>

<body bgcolor="#FFFFFF">
    <table width="80%" bgcolor="#000000" border="0"
        cellspacing="1" cellpadding="5">
```

The PHP script is used to display the current image:

```
<tr>
  <td bgcolor="#FFFFFF" colspan="3" align="center">
    <?php
      print ("<img src=\"\${iObj->imageName}\">");
    ?>
  </td>
</tr>
```

The form lets us resize the image. We will need two input boxes – one for the width and one for the height. We also need to store some values for the temporary filename, the current filename, and the action to perform:

```
<tr>
  <td bgcolor="#FFFFFF" width="33%" align="center">

    <form action="<?php print $PHP_SELF; ?>" method="POST">
      <input type="hidden" name="pName"
        value= "<?php print ${iName}; ?>">

      <input type="hidden" name="tmpName"
        value= "<?php print uniqid(time()); ?>">

      <input type="hidden" name="action" value="resize">

      <b><u>Resize</u></b><br />

      W: <input type="text" size="4" name="width"> x
      H: <input type="text" size="4" name="height"><p />

      <input type="submit" name="submit" value="Resize"></form>
    </td>
```

Our second form is for changing the format of the image. Again, we need to store the current and temporary filenames, and the action to perform. Instead of input boxes for the width and height, we will build a selection box of the supported file formats:

```
<td bgcolor="#FFFFFF" width="33%" align="center">
  <form action="<?php print $PHP_SELF; ?>" method="POST">

    <input type="hidden" name="pName"
      value= "<?php print ${iName}; ?>">

    <input type="hidden" name="tmpName"
      value= "<?php print time(); ?>">

    <input type="hidden" name="action" value="convert">

    <b><u>Convert Format</u></b><br />

    <select name="format">
```

```

        <?php
            foreach($iObj->imageTypeList as $cKey=>$cVal) {
                print ("<option value=\"{$cKey}\"> {$cKey} ($cVal)</option>");
            }
        ?>

    </select><p /><input type="submit" name="submit" value="Change">
</form>
</td>

```

Our third form allows us to apply effects to the image. Our selection box this time contains a list of possible effects that we can apply to the image:

```

<td bgcolor="#FFFFFF" width="33%" align="center">

    <form action="<?php print $PHP_SELF; ?>" method="POST">

        <input type="hidden" name="pName"
            value= "<?php print $iName ;?>">

        <input type="hidden" name="tmpName"
            value= "<?php print time(); ?>">

        <input type="hidden" name="action" value="effect">

        <b><u>Add Effect</u></b><br />

        <select name="effect">
            <option value="-cycle 16">Cycle Colormap 16 Positions</option>
            <option value="-despeckle">De-speckle image </option>
            <option value="-enhance">Enhance Noisy Image</option>
            <option value="-flip">Vertical Mirror </option>
            <option value="-flop">Horizontal Mirror </option>
            <option value="-monochrome">B&W</option>
            <option value="-negate">Negative</option>
            <option value="-normalize">Normalize Image      </option>
        </select><p />

        <input type="submit" name="submit" value="Add">

    </form>

</td>

</tr>

</table>
</body>
</html>

```

To see the resulting image, after implementing the class library functions and opening `imagemagick.php` in a browser, refer to *Fig 6.1 – ImageMagick Manipulation Examples* in *Appendix G*.

To see the resulting image, when we apply a negative effect to the image, refer to *Fig 6.2 – ImageMagick Manipulation Examples* in *Appendix G*.

Summary

In this chapter we started by taking a look at what ImageMagick is and what it can do. We looked at how we could get ImageMagick installed on UNIX and Windows machines. Then we discussed the three tools – `Convert`, `Mogrify`, and `Identify` – that we use in the chapter.

We went on to see how we can interface between ImageMagick and PHP – there is currently no extension for ImageMagick, so we had to pass information between ImageMagick and PHP with the `backtick` operator.

Once we had seen how we could do this, we created a class to perform some basic ImageMagick functions and then implemented the class with a simple form-based interface.

For a more realistic example of how to use the class library and the ImageMagick toolkit, refer to Chapter 11 which uses the class library to implement a fully functional Image Gallery application.

