3

Error Handling

Errors are a fact of life. Mr. Murphy has an entire collection of laws detailing the prevalence and inescapability of errors. In programming, errors come in two basic flavors:

- **External errors**—These are errors in which the code takes an unanticipated path due to a part of the program not acting as anticipated. For example, a database connection failing to be established when the code requires it to be established successfully is an external error.
- **Code logic errors**—These errors, commonly referred to as *bugs*, are errors in which the code design is fundamentally flawed due to either faulty logic ("it just doesn't work that way") or something as simple as a typo.

These two categories of errors differ significantly in several ways:

- External errors will always occur, regardless of how "bug free" code is. They are not bugs in and of themselves because they are external to the program.
- External errors that aren't accounted for in the code logic can be bugs. For example, blindly assuming that a database connection will always succeed is a bug because the application will almost certainly not respond correctly in that case.
- Code logic errors are much more difficult to track down than external errors because by definition their location is not known. You can implement data consistency checks to expose them, however.

PHP has built-in support for error handling, as well as a built-in severity system that allows you to see only errors that are serious enough to concern you. PHP has three severity levels of errors:

- `E_NOTICE`
- `E_WARNING`
- `E_ERROR`

`E_NOTICE` errors are minor, nonfatal errors designed to help you identify possible bugs in your code. In general, an `E_NOTICE` error is something that works but may not do what you intended. An example might be using a variable in a non-assignment expression before it has been assigned to, as in this case:

```
<?php
    $variable++;
?>
```

This example will increment `$variable` to `1` (because variables are instantiated as 0/false/empty string), but it will generate an `E_NOTICE` error. Instead you should use this:

```
<?php
    $variable = 0;
    $variable++;
?>
```

This check is designed to prevent errors due to typos in variable names. For example, this code block will work fine:

```
<?
    $variable = 0;
    $variabel++;
?>
```

However, `$variable` will not be incremented, and `$variabel` will be. `E_NOTICE` warnings help catch this sort of error; they are similar to running a Perl program with `use warnings` and `use strict` or compiling a C program with `-Wall`.

In PHP, `E_NOTICE` errors are turned off by default because they can produce rather large and repetitive logs. In my applications, I prefer to turn on `E_NOTICE` warnings in development to assist in code cleanup and then disable them on production machines.

`E_WARNING` errors are nonfatal runtime errors. They do not halt or change the control flow of the script, but they indicate that something bad happened. Many external errors generate `E_WARNING` errors. An example is getting an error on a call to `fopen()` to `mysql_connect()`.

`E_ERROR` errors are unrecoverable errors that halt the execution of the running script. Examples include attempting to instantiate a non-existent class and failing a type hint in a function. (Ironically, passing the incorrect number of arguments to a function is only an `E_WARNING` error.)

PHP supplies the `trigger_error()` function, which allows a user to generate his or her own errors inside a script. There are three types of errors that can be triggered by the user, and they have identical semantics to the errors just discussed:

- `E_USER_NOTICE`
- `E_USER_WARNING`
- `E_USER_ERROR`

You can trigger these errors as follows:

```
while(!feof($fp)) {
  $line = fgets($fp);
  if(!parse_line($line)) {
    trigger_error("Incomprehensible data encountered", E_USER_NOTICE);
  }
}
```

If no error level is specified, `E_USER_NOTICE` is used.

In addition to these errors, there are five other categories that are encountered somewhat less frequently:

- **`E_PARSE`**—The script has a syntactic error and could not be parsed. This is a fatal error.
- **`E_COMPILE_ERROR`**—A fatal error occurred in the engine while compiling the script.
- **`E_COMPILE_WARNING`**—A nonfatal error occurred in the engine while parsing the script.
- **`E_CORE_ERROR`**—A fatal runtime error occurred in the engine.
- **`E_CORE_WARNING`**—A nonfatal runtime error occurred in the engine.

In addition, PHP uses the `E_ALL` error category for all error reporting levels.
You can control the level of errors that are percolated up to your script by using the `php.ini` setting `error_reporting`. `error_reporting` is a bit-field test set that uses defined constants, such as the following for all errors:

```
error_reporting = E_ALL
```

`error_reporting` uses the following for all errors except for `E_NOTICE`, which can be set by XOR'ing `E_ALL` and `E_NOTICE`:

```
error_reporting = E_ALL ~ E_NOTICE
```

Similarly, `error_reporting` uses the following for only fatal errors (bitwise OR of the two error types):

```
error_reporting = E_ERROR | E_USER_ERROR
```

Note that removing `E_ERROR` from the `error_reporting` level does not allow you to ignore fatal errors; it only prevents an error handler from being called for it.

Handling Errors

Now that you've seen what sort of errors PHP will generate, you need to develop a plan for dealing with them when they happen. PHP provides four choices for handling errors that fall within the `error_reporting` threshold:

- Display them.
- Log them.
- Ignore them.
- Act on them.

None of these options supersedes the others in importance or functionality; each has an important place in a robust error-handling system. Displaying errors is extremely beneficial in a development environment, and logging them is usually more appropriate in a production environment. Some errors can be safely ignored, and others demand reaction. The exact mix of error-handling techniques you employ depends on your personal needs.

Displaying Errors

When you opt to display errors, an error is sent to the standard output stream, which in the case of a Web page means that it is sent to the browser. You toggle this setting on and off via this `php.ini` setting:

```
display_errors = On
```

`display errors` is very helpful for development because it enables you to get instant feedback on what went wrong with a script without having to tail a logfile or do anything but simply visit the Web page you are building.

What's good for a developer to see, however, is often bad for an end user to see. Displaying PHP errors to an end user is usually undesirable for three reasons:

- It looks ugly.
- It conveys a sense that the site is buggy.
- It can disclose details of the script internals that a user might be able to use for nefarious purposes.

The third point cannot be emphasized enough. If you are looking to have security holes in your code found and exploited, there is no faster way than to run in production with `display_errors` on. I once saw a single incident where a bad INI file got pushed out for a couple errors on a particularly high-traffic site. As soon as it was noticed, the corrected file was copied out to the Web servers, and we all figured the damage was mainly to our pride. A year and a half later, we tracked down and caught a cracker who had been maliciously defacing other members' pages. In return for our not trying to prosecute him, he agreed to disclose all the vulnerabilities he had found. In addition to the standard bag of JavaScript exploits (it was a site that allowed for a lot of user-developed content), there were a couple particularly clever application hacks that he had developed from perusing the code that had appeared on the Web for mere hours the year before.

We were lucky in that case: The main exploits he had were on unvalidated user input and nondefaulted variables (this was in the days before `register_global`). All our database connection information was held in libraries and not on the pages. Many a site has been seriously violated due to a chain of security holes like these:

- Leaving display_errors on.
- Putting database connection details (`mysql_connect()`) in the pages.
- Allowing nonlocal connections to MySQL.

These three mistakes together put your database at the mercy of anyone who sees an error page on your site. You would (hopefully) be shocked at how often this occurs.

I like to leave `display_errors` on during development, but I never turn it on in production.

Production Display of Errors

How to notify users of errors is often a political issue. All the large clients I have worked for have had strict rules regarding what to do when a user incurs an error. Business rules have ranged from display of a customized or themed error page to complex logic regarding display of some sort of cached version of the content they were looking for. From a business perspective, this makes complete sense: Your Web presence is your link to your customers, and any bugs in it can color their perceptions of your whole business.

Regardless of the exact content that needs to be returned to a user in case of an unexpected error, the last thing I usually want to show them is a mess of debugging information. Depending on the amount of information in your error messages, that could be a considerable disclosure of information.

One of the most common techniques is to return a 500 error code from the page and set a custom error handler to take the user to a custom error page. A 500 error code in HTTP signifies an internal server error. To return one from PHP, you can send this:

```
header("HTTP/1.0 500 Internal Server Error");
```

Then in your Apache configuration you can set this:

```
ErrorDocument 500 /custom-error.php
```

This will cause any page returning a status code of 500 to be redirected (internally—meaning transparently to the user) to `/custom-error.php`.

In the section “Installing a Top-Level Exception Handler,” later in this chapter, you will see an alternative, exception-based method for handling this.

Logging Errors

PHP internally supports both logging to a file and logging via `syslog` via two settings in the `php.ini` file. This setting sets errors to be logged:

```
log_errors = On
```

And these two settings set logging to go to a file or to `syslog`, respectively:

```
error_log = /path/to/filename

error_log = syslog
```

Logging provides an auditable trace of any errors that transpire on your site. When diagnosing a problem, I often place debugging lines around the area in question.
In addition to the errors logged from system errors or via `trigger_error()`, you can manually generate an error log message with this:

```
error_log("This is a user defined error");
```

Alternatively, you can send an email message or manually specify the file. See the PHP manual for details. `error_log` logs the passed message, regardless of the `error_reporting` level that is set; `error_log` and `error_reporting` are two completely different entries to the error logging facilities.

If you have only a single server, you should log directly to a file. `syslog` logging is quite slow, and if any amount of logging is generated on every script execution (which is probably a bad idea in any case), the logging overhead can be quite noticeable.

If you are running multiple servers, though, `syslog`'s centralized logging abilities provide a convenient way to consolidate logs in real-time from multiple machines in a single location for analysis and archival. You should avoid excessive logging if you plan on using `syslog`.

Ignoring Errors

PHP allows you to selectively suppress error reporting when you think it might occur with the `@` syntax. Thus, if you want to open a file that may not exist and suppress any errors that arise, you can use this:

```
$fp = @fopen($file, $mode);
```

Because (as we will discuss in just a minute) PHP's error facilities do not provide any flow control capabilities, you might want to simply suppress errors that you know will occur but don't care about.

Consider a function that gets the contents of a file that might not exist:

```
$content = file_get_content($sometimes_valid);
```

If the file does not exist, you get an `E_WARNING` error. If you know that this is an expected possible outcome, you should suppress this warning; because it was expected, it's not really an error. You do this by using the `@` operator, which suppresses warnings on individual calls:

```
$content = @file_get_content($sometimes_valid);
```

In addition, if you set the `php.ini` setting `track_errors = On`, the last error message encountered will be stored in `$php_errormsg`. This is true regardless of whether you have used the `@` syntax for error suppression.

Acting On Errors

PHP allows for the setting of custom error handlers via the `set_error_handler()` function. To set a custom error handler, you define a function like this:

```
<?php
require "DB/Mysql.inc";
function user_error_handler($severity, $msg, $filename, $linenum) {
   $dbh = new DB_Mysql_Prod;
   $query = "INSERT INTO errorlog
                 (severity, message, filename, linenum, time)
                 VALUES(?,?,?,?, NOW())";
   $sth = $dbh->prepare($query);
   switch($severity) {
   case E_USER_NOTICE:
      $sth->execute('NOTICE', $msg, $filename, $linenum);
      break;
   case E_USER_WARNING:
      $sth->execute('WARNING', $msg, $filename, $linenum);
      break;
   case E_USER_ERROR:
      $sth->execute('FATAL', $msg, $filename, $linenum);
      echo "FATAL error $msg at $filename:$linenum<br>";
      break;
   default:
      echo "Unknown error at $filename:$linenum<br>";
      break;
   }
}
?>
```

You set a function with this:

```
set_error_handler("user_error_handler");
```

Now when an error is detected, instead of being displayed or printed to the error log, it will be inserted into a database table of errors and, if it is a fatal error, a message will be printed to the screen. Keep in mind that error handlers provide no flow control. In the case of a nonfatal error, when processing is complete, the script is resumed at the point where the error occurred; in the case of a fatal error, the script exits after the handler is done.

Mailing Oneself

It might seem like a good idea to set up a custom error handler that uses the `mail()` function to send an email to a developer or a systems administrator whenever an error occurs. In general, this is a very bad idea.

Errors have a way of clumping up together. It would be great if you could guarantee that the error would only be triggered at most once per hour (or any specified time period), but what happens more often is that when an unexpected error occurs due to a coding bug, many requests are affected by it. This means that your nifty mailing `error_handler()` function might send 20,000 mails to your account before you are able to get in and turn it off. Not a good thing.

If you need this sort of reactive functionality in your error-handling system, I recommend writing a script that parses your error logs and applies intelligent limiting to the number of mails it sends.

Handling External Errors

Although we have called what we have done so far in this chapter *error handling*, we really haven't done much handling at all. We have accepted and processed the warning messages that our scripts have generated, but we have not been able to use those techniques to alter the flow control in our scripts, meaning that, for all intents and purposes, we have not really handled our errors at all. Adaptively handling errors largely involves being aware of where code can fail and deciding how to handle the case when it does. External failures mainly involve connecting to or extracting data from external processes.

Consider the following function, which is designed to return the `passwd` file details (home directory, shell, gecos information, and so on) for a given user:

```
<?php
function get_passwd_info($user) {
    $fp = fopen("/etc/passwd", "r");
    while(!feof($fp)) {
        $line = fgets($fp);
        $fields = explode(";", $line);
        if($user == $fields[0]) {
            return $fields;
        }
    }
    return false;
}
?>
```

As it stands, this code has two bugs in it: One is a pure code logic bug, and the second is a failure to account for a possible external error. When you run this example, you get an array with elements like this:

```
<?php
    print_r(get_passwd_info('www'));
?>
```

```
Array
        (
            [0] => www:*:70:70:World Wide Web Server:/Library/WebServer:/noshell
        )
```

This is because the first bug is that the field separator in the `passwd` file is `:`, not `;`. So this:

```
$fields = explode(";", $line);
```

needs to be this:

```
$fields = explode(":", $line);
```

The second bug is subtler. If you fail to open the `passwd` file, you will generate an `E_WARNING` error, but program flow will proceed unabated. If a user is not in the `passwd` file, the function returns `false`. However, if the `fopen` fails, the function also ends up returning `false`, which is rather confusing.

This simple example demonstrates one of the core difficulties of error handling in procedural languages (or at least languages without exceptions): How do you propagate an error up to the caller that is prepared to interpret it?

If you are utilizing the data locally, you can often make local decisions on how to handle the error. For example, you could change the password function to format an error on return:

```
<?php
function get_passwd_info($user) {
    $fp = fopen("/etc/passwd", "r");
    if(!is_resource($fp)) {
        return "Error opening file";
    }
    while(!feof($fp)) {
        $line = fgets($fp);
        $fields = explode(":", $line);
        if($user == $fields[0]) {
            return $fields;
        }
    }
    return false;
}
?>
```

Alternatively, you could set a special value that is not a normally valid return value:

```
<?php
function get_passwd_info($user) {
    $fp = fopen("/etc/passwd", "r");
    if(!is_resource($fp)) {
        return -1;
    }
    while(!feof($fp)) {
        $line = fgets($fp);
        $fields = explode(":", $line);
        if($user == $fields[0]) {
            return $fields;
        }
    }
    return false;
}
?>
```

You can use this sort of logic to bubble up errors to higher callers:

```
<?php
function is_shelled_user($user) {
    $passwd_info = get_passwd_info($user);
    if(is_array($passwd_info) && $passwd_info[7] != '/bin/false') {
        return 1;
    }
    else if($passwd_info === -1) {
        return -1;
    }
    else {
        return 0;
    }
}
?>
```

When this logic is used, you have to detect all the possible errors:

```
<?php
$v = is_shelled_user('www');
if($v === 1) {
    echo "Your Web server user probably shouldn't be shelled.\n";
}
else if($v === 0) {
    echo "Great!\n";
}
else {
    echo "An error occurred checking the user\n";
}
?>
```

If this seems nasty and confusing, it's because it is. The hassle of manually bubbling up errors through multiple callers is one of the prime reasons for the implementation of exceptions in programming languages, and now in PHP5 you can use exceptions in PHP as well. You can somewhat make this particular example work, but what if the function in question could validly return any number? How could you pass the error up in a clear fashion then? The worst part of the whole mess is that any convoluted error-handling scheme you devise is not localized to the functions that implement it but needs to be understood and handled by anyone in its call hierarchy as well.

Exceptions

The methods covered to this point are all that was available before PHP5, and you can see that this poses some critical problems, especially when you are writing larger applications. The primary flaw is in returning errors to a user of a library. Consider the error checking that you just implemented in the `passwd` file reading function.

When you were building that example, you had two basic choices on how to handle a connection error:

- Handle the error locally and return invalid data (such as `false`) back to the caller.
- Propagate and preserve the error and return it to the caller instead of returning the result set.

In the `passwd` file reading function example, you did not select the first option because it would have been presumptuous for a library to know how the application wants it to handle the error. For example, if you are writing a database-testing suite, you might want to propagate the error in high granularity back to the top-level caller; on the other hand, in a Web application, you might want to return the user to an error page.

The preceding example uses the second method, but it is not much better than the first option. The problem with it is that it takes a significant amount of foresight and planning to make sure errors can always be correctly propagated through an application. If the result of a database query is a string, for example, how do you differentiate between that and an error string?

Further, propagation needs to be done manually: At every step, the error must be manually bubbled up to the caller, recognized as an error, and either passed along or handled. You saw in the last section just how difficult it is to handle this.

Exceptions are designed to handle this sort of situation. An *exception* is a flow-control structure that allows you to stop the current path of execution of a script and unwind the stack to a prescribed point. The error that you experienced is represented by an object that is set as the exception.

Exceptions are objects. To help with basic exceptions, PHP has a built-in `Exception` class that is designed specifically for exceptions. Although it is not necessary for exceptions to be instances of the `Exception` class, there are some benefits of having any class that you want to throw exceptions derive from `Exception`, which we'll discuss in a moment. To create a new exception, you instantiate an instance of the `Exception` class you want and you throw it.

When an exception is thrown, the `Exception` object is saved, and execution in the current block of code halts immediately. If there is an exception-handler block set in the current scope, the code jumps to that location and executes the handler. If there is no handler set in the current scope, the execution stack is popped, and the caller's scope is checked for an exception-handler block. This repeats until a handler is found or the main, or top, scope is reached.

Running this code:

```
<?php
    throw new Exception;
?>
```

returns the following:

```
> php uncaught-exception.php

Fatal error: Uncaught exception 'exception'! in Unknown on line 0
```

An uncaught exception is a fatal error. Thus, exceptions introduce their own maintenance requirements. If exceptions are used as warnings or possibly nonfatal errors in a script, every caller of that block of code must know that an exception may be thrown and must be prepared to handle it.

Exception handling consists of a block of statements you want to try and a second block that you want to enter if and when you trigger any errors there. Here is a simple example that shows an exception being thrown and caught:

```
try {
    throw new Exception;
    print "This code is unreached\n";
}
catch (Exception $e) {
    print "Exception caught\n";
}
```

In this case you throw an exception, but it is in a `try` block, so execution is halted and you jump ahead to the `catch` block. `catch` catches an `Exception` class (which is the class being thrown), so that block is entered. `catch` is normally used to perform any cleanup that might be necessary from the failure that occurred.

I mentioned earlier that it is not necessary to throw an instance of the `Exception` class. Here is an example that throws something other than an `Exception` class:

```
<?php

class AltException {}

try {
        throw new AltException;
}
catch (Exception $e) {
        print "Caught exception\n";
}
?>
```

Running this example returns the following:

```
> php failed_catch.php
Fatal error: Uncaught exception 'altexception'! in Unknown on line 0
```

This example failed to catch the exception because it threw an object of class `AltException` but was only looking to catch an object of class `Exception`.

Here is a less trivial example of how you might use a simple exception to facilitate error handling in your old favorite, the factorial function. The simple factorial function is valid only for natural numbers (integers > 0). You can incorporate this input checking into the application by throwing an exception if incorrect data is passed:

```
<?php
// factorial.inc
// A simple Factorial Function
function factorial($n) {
    if(!preg_match('/^\d+$/',$n) || $n < 0 ) {
        throw new Exception;
    } else if ($n == 0 || $n == 1) {
        return $n;
    }
    else {
        return $n * factorial($n - 1);
    }
}
?>
```

Incorporating sound input checking on functions is a key tenant of defensive programming.

Why the regex?

It might seem strange to choose to evaluate whether `$n` is an integer by using a regular expression instead of the `is_int` function. The `is_int` function, however, does not do what you want. It only evaluates whether `$n` has been typed as a string or as integer, not whether the value of `$n` is an integer. This is a nuance that will catch you if you use `is_int` to validate form data (among other things). We will explore dynamic typing in PHP in Chapter 20, "PHP and Zend Engine Internals."

When you call `factorial`, you need to make sure that you execute it in a `try` block if you do not want to risk having the application die if bad data is passed in:

```
<html>
<form method="POST">
Compute the factorial of
```

```
<input type="text" name="input" value="<?= $_POST['input'] ?>"><br>
<?php
include "factorial.inc";
if($_POST['input']) {
    try {
        $input = $_POST['input'];
        $output = factorial($input);
        echo "$_POST[input]! = $output";
    }
    catch (Exception $e) {
        echo "Only natural numbers can have their factorial computed.";
    }
}
?>
<br>
<input type=submit name=posted value="Submit">
</form>
```

Using Exception Hierarchies

You can have `try` use multiple `catch` blocks if you want to handle different errors differently. For example, we can modify the factorial example to also handle the case where `$n` is too large for PHP's math facilities:

```
class OverflowException {}
class NaNException {}
function factorial($n)
{
    if(!preg_match('/^\d+$/', $n) || $n < 0 ) {
        throw new NaNException;
    }
    else if ($n == 0 || $n == 1) {
        return $n;
    }
    else if ($n > 170 ) {
        throw new OverflowException;
    }
    else {
        return $n * factorial($n - 1);
    }
}
```

Now you handle each error case differently:

```
<?php
if($_POST['input']) {
    try {
        $input = $_POST['input'];
        $output = factorial($input);
        echo "$_POST[input]! = $output";
    }
    catch (OverflowException $e) {
        echo "The requested value is too large.";
    }
    catch (NaNException $e) {
        echo "Only natural numbers can have their factorial computed.";
    }
}
?>
```

As it stands, you now have to enumerate each of the possible cases separately. This is both cumbersome to write and potentially dangerous because, as the libraries grow, the set of possible exceptions will grow as well, making it ever easier to accidentally omit one.

To handle this, you can group the exceptions together in families and create an inheritance tree to associate them:

```
class MathException extends Exception {}
class NaNException extends MathException {}
class OverflowException extends MathException {}
```

You could now restructure the `catch` blocks as follows:

```
<?php
if($_POST['input']) {
    try {
        $input = $_POST['input'];
        $output = factorial($input);
        echo "$_POST[input]! = $output";
    }
    catch (OverflowException $e) {
        echo "The requested value is too large.";
    }
    catch (MathException $e) {
        echo "A generic math error occurred";
    }
    catch (Exception $e) {
        echo "An unknown error occurred";
    }
}
?>
```

In this case, if an `OverflowException` error is thrown, it will be caught by the first `catch` block. If any other descendant of `MathException` (for example, `NaNException`) is thrown, it will be caught by the second `catch` block. Finally, any descendant of `Exception` not covered by any of the previous cases will be caught.

This is the benefit of having all exceptions inherit from `Exception`: It is possible to write a generic `catch` block that will handle all exceptions without having to enumerate them individually. Catchall exception handlers are important because they allow you to recover from even the errors you didn't anticipate.

A Typed Exceptions Example

So far in this chapter, all the exceptions have been (to our knowledge, at least) attribute free. If you only need to identify the type of exception thrown and if you have been careful in setting up our hierarchy, this will satisfy most of your needs. Of course, if the only information you would ever be interested in passing up in an exception were strings, exceptions would have been implemented using strings instead of full objects. However, you would like to be able to include arbitrary information that might be useful to the caller that will catch the exception.

The base exception class itself is actually deeper than indicated thus far. It is a *built-in class*, meaning that it is implemented in C instead of PHP. It basically looks like this:

```
class Exception {
    Public function _ _construct($message=false, $code=false) {
        $this->file = _ _FILE_ _;
        $this->line = _ _LINE_ _;
        $this->message = $message; // the error message as a string
        $this->code = $code;  // a place to stick a numeric error code
    }
    public function getFile() {
        return $this->file;
    }
    public function getLine() {
        return $this->line;
    }
    public function getMessage() {
        return $this->message;
    }
    public function getCode() {
       return $this->code;
    }
}
```

Tracking `_ _FILE_ _` and `_ _LINE_ _` for the last caller is often useless information. Imagine that you decide to throw an exception if you have a problem with a query in the `DB_Mysql` wrapper library:

```
class DB_Mysql {
    // ...
    public function execute($query) {
      if(!$this->dbh) {
        $this->connect();
      }
      $ret = mysql_query($query, $this->dbh);
      if(!is_resource($ret)) {
        throw new Exception;
      }
      return new MysqlStatement($ret);
    }
}
```

Now if you trigger this exception in the code by executing a syntactically invalid query, like this:

```
<?php
        require_once "DB.inc";
        try {
        $dbh = new DB_Mysql_Test;
        // ... execute a number of queries on our database connection
        $rows = $dbh->execute("SELECT * FROM")->fetchall_assoc();
        }
        catch (Exception $e) {
                print_r($e);
        }
?>
```

you get this:

```
exception Object
(
    [file] => /Users/george/Advanced PHP/examples/chapter-3/DB.inc
    [line] => 42
)
```

Line 42 of `DB.inc` is the `execute()` statement itself! If you executed a number of queries within the `try` block, you would have no insight yet into which one of them caused the error. It gets worse, though: If you use your own exception class and manually set `$file` and `$line` (or call `parent::_ _construct` to run `Exception`'s constructor), you would actually end up with the first callers `_ _FILE_ _` and `_ _LINE_ _` being the constructor itself! What you want instead is a full backtrace from the moment the problem occurred.

You can now start to convert the `DB` wrapper libraries to use exceptions. In addition to populating the backtrace data, you can also make a best-effort attempt to set the `message` and `code` attributes with the MySQL error information:

```
class MysqlException extends Exception {
  public $backtrace;
  public function _ _construct($message=false, $code=false) {
    if(!$message) {
      $this->message = mysql_error();
    }
    if(!$code) {
      $this->code = mysql_errno();
    }
    $this->backtrace = debug_backtrace();
  }
}
```

If you now change the library to use this exception type:

```
class DB_Mysql {
  public function execute($query) {
    if(!$this->dbh) {
      $this->connect();
    }
    $ret = mysql_query($query, $this->dbh);
    if(!is_resource($ret)) {
      throw new MysqlException;
    }
    return new MysqlStatement($ret);
  }
}
```

and repeat the test:

```
<?php
        require_once "DB.inc";
        try {
        $dbh = new DB_Mysql_Test;
        // ... execute a number of queries on our database connection
        $rows = $dbh->execute("SELECT * FROM")->fetchall_assoc();
        }
        catch (Exception $e) {
                print_r($e);
        }
?>
```

you get this:

```
mysqlexception Object
(
  [backtrace] => Array
    (
      [0] => Array
        (
          [file] => /Users/george/Advanced PHP/examples/chapter-3/DB.inc
          [line] => 45
          [function] => _ _construct
          [class] => mysqlexception
```

```
            [type] => ->
            [args] => Array
              (
              )
          )
        [1] => Array
          (
            [file] => /Users/george/Advanced PHP/examples/chapter-3/test.php
            [line] => 5
            [function] => execute
            [class] => mysql_test
            [type] => ->
            [args] => Array
              (
                [0] => SELECT * FROM
              )
          )
    )

  [message] => You have an error in your SQL syntax near '' at line 1
  [code] => 1064
)
```

Compared with the previous exception, this one contains a cornucopia of information:

- Where the error occurred
- How the application got to that point
- The MySQL details for the error

You can now convert the entire library to use this new exception:

```
class MysqlException extends Exception {
  public $backtrace;
  public function _ _construct($message=false, $code=false) {
    if(!$message) {
      $this->message = mysql_error();
    }
    if(!$code) {
     $this->code = mysql_errno();
    }
    $this->backtrace = debug_backtrace();
  }
}
class DB_Mysql {
  protected $user;
  protected $pass;
  protected $dbhost;
  protected $dbname;
  protected $dbh;

  public function _ _construct($user, $pass, $dbhost, $dbname) {
    $this->user = $user;
    $this->pass = $pass;
    $this->dbhost = $dbhost;
    $this->dbname = $dbname;
  }
  protected function connect() {
    $this->dbh = mysql_pconnect($this->dbhost, $this->user, $this->pass);
    if(!is_resource($this->dbh)) {
      throw new MysqlException;
    }
    if(!mysql_select_db($this->dbname, $this->dbh)) {
      throw new MysqlException;
    }
  }
  public function execute($query) {
    if(!$this->dbh) {
      $this->connect();
    }
    $ret = mysql_query($query, $this->dbh);
    if(!$ret) {
      throw new MysqlException;
    }
    else if(!is_resource($ret)) {
      return TRUE;
    } else {
      return new DB_MysqlStatement($ret);
    }
  }
  public function prepare($query) {
    if(!$this->dbh) {
      $this->connect();
    }
    return new DB_MysqlStatement($this->dbh, $query);
  }
}
class DB_MysqlStatement {
  protected $result;
  protected $binds;
  public $query;
  protected $dbh;
  public function _ _construct($dbh, $query) {
    $this->query = $query;
    $this->dbh = $dbh;
    if(!is_resource($dbh)) {
      throw new MysqlException("Not a valid database connection");
    }
  }
  public function bind_param($ph, $pv) {
    $this->binds[$ph] = $pv;
  }
  public function execute() {
    $binds = func_get_args();
    foreach($binds as $index => $name) {
      $this->binds[$index + 1] = $name;
    }
    $cnt = count($binds);
    $query = $this->query;
    foreach ($this->binds as $ph => $pv) {
      $query = str_replace(":$ph", "'".mysql_escape_string($pv)."'", $query);
    }
    $this->result = mysql_query($query, $this->dbh);
    if(!$this->result) {
      throw new MysqlException;
    }
  }
  public function fetch_row() {
    if(!$this->result) {
      throw new MysqlException("Query not executed");
    }
    return mysql_fetch_row($this->result);
  }
  public function fetch_assoc() {
    return mysql_fetch_assoc($this->result);
  }
  public function fetchall_assoc() {
    $retval = array();
    while($row = $this->fetch_assoc()) {
      $retval[] = $row;
    }
    return $retval;
  }
}

? >
```

Cascading Exceptions

Sometimes you might want to handle an error but still pass it along to further error handlers. You can do this by throwing a new exception in the `catch` block:

```
<?php
try {
        throw new Exception;
}
catch (Exception $e) {
        print "Exception caught, and rethrown\n";
        throw new Exception;
}
?>
```

The `catch` block catches the exception, prints its message, and then throws a new exception. In the preceding example, there is no `catch` block to handle this new exception, so it goes uncaught. Observe what happens as you run the code:

```
> php re-throw.php
Exception caught, and rethrown

Fatal error: Uncaught exception 'exception'! in Unknown on line 0
```

In fact, creating a new exception is not necessary. If you want, you can rethrow the current `Exception` object, with identical results:

```
<?php
try {
        throw new Exception;
}
catch (Exception $e) {
        print "Exception caught, and rethrown\n";
        throw $e;
}
?>
```

Being able to rethrow an exception is important because you might not be certain that you want to handle an exception when you catch it. For example, say you want to track referrals on your Web site. To do this, you have a table:

```
CREATE TABLE track_referrers (
    url varchar2(128) not null primary key,
    counter int
);
```

The first time a URL is referred from, you need to execute this:

```
INSERT INTO track_referrers VALUES('http://some.url/', 1)
```

On subsequent requests, you need to execute this:

```
UPDATE track_referrers SET counter=counter+1 where url = 'http://some.url/'
```

You could first select from the table to determine whether the URL's row exists and choose the appropriate query based on that. This logic contains a race condition though: If two referrals from the same URL are processed by two different processes simultaneously, it is possible for one of the inserts to fail.

A cleaner solution is to blindly perform the insert and call `update` if the insert failed and produced a unique key violation. You can then catch all `MysqlException` errors and perform the update where indicated:

```
<?php
include "DB.inc";

function track_referrer($url) {
        $insertq = "INSERT INTO referrers (url, count) VALUES(:1, :2)";
        $updateq = "UPDATE referrers SET count=count+1 WHERE url = :1";
        $dbh = new DB_Mysql_Test;
        try {
                $sth = $dbh->prepare($insertq);
                $sth->execute($url, 1);
        }
        catch (MysqlException $e) {
                if($e->getCode == 1062) {
                        $dbh->prepare($updateq)->execute($url);
                }
                else {
                        throw $e;
                }
        }
}
?>
```

Alternatively, you can use a purely typed exception solution where `execute` itself throws different exceptions based on the errors it incurs:

```
class Mysql_Dup_Val_On_Index extends MysqlException {}
//...
class DB_Mysql {
  // ...
  public function execute($query) {
    if(!$this->dbh) {
      $this->connect();
    }
    $ret = mysql_query($query, $this->dbh);
    if(!$ret) {
      if(mysql_errno() == 1062) {
        throw new Mysql_Dup_Val_On_Index;
      else {
        throw new MysqlException;
      }
    }
    else if(!is_resource($ret)) {
       return TRUE;
    } else {
      return new MysqlStatement($ret);
    }
  }
}
```

Then you can perform your checking, as follows:

```
function track_referrer($url) {
  $insertq = "INSERT INTO referrers (url, count) VALUES('$url', 1)";
  $updateq = "UPDATE referrers SET count=count+1 WHERE url = '$url'";
  $dbh = new DB_Mysql_Test;
  try {
    $sth = $dbh->execute($insertq);
  }
  catch (Mysql_Dup_Val_On_Index $e) {
    $dbh->execute($updateq);
  }
}
```

Both methods are valid; it's largely a matter of taste and style. If you go the path of typed exceptions, you can gain some flexibility by using a factory pattern to generate your errors, as in this example:

```
class MysqlException {
  // ...
  static function createError($message=false, $code=false) {
     if(!$code) {
       $code = mysql_errno();
     }
     if(!$message) {
       $message = mysql_error();
     }
     switch($code) {
       case 1062:
         return new Mysql_Dup_Val_On_Index($message, $code);
         break;
       default:
         return new MysqlException($message, $code);
         break;
```

```
        }
      }
}
```

There is the additional benefit of increased readability. Instead of a cryptic constant being thrown, you get a suggestive class name. The value of readability aids should not be underestimated.
Now instead of throwing specific errors in your code, you just call this:

```
throw MysqlException::createError();
```

Handling Constructor Failure

Handling constructor failure in an object is a difficult business. A class constructor in PHP *must* return an instance of that class, so the options are limited:

- You can use an initialized attribute in the object to mark it as correctly initialized.
- You can perform no initialization in the constructor.
- You can throw an exception in the constructor.

The first option is very inelegant, and we won't even consider it seriously. The second option is a pretty common way of handling constructors that might fail. In fact, in PHP4, it is the preferable way of handling this.

To implement that, you would do something like this:

```
class ResourceClass {
  protected $resource;
  public function _ _construct() {
    // set username, password, etc
  }
  public function init() {
    if(($this->resource = resource_connect()) == false) {
      return false;
    }
    return true;
  }
}
```

When the user creates a new `ResourceClass` object, there are no actions taken, which can mean the code fails. To actually initialize any sort of potentially faulty code, you call the `init()` method. This can fail without any issues.

The third option is usually the best available, and it is reinforced by the fact that it is the standard method of handling constructor failure in more traditional object-oriented languages such as C++. In C++ the cleanup done in a `catch` block around a constructor call is a little more important than in PHP because memory management might need to be performed. Fortunately, in PHP memory management is handled for you, as in this example:

```
class Stillborn {
  public function _ _construct() {
    throw new Exception;
  }
  public function _ _destruct() {
    print "destructing\n";
  }
}
try {
  $sb = new Stillborn;
}
catch(Stillborn $e) {}
```

Running this generates no output at all:

```
>php stillborn.php
>
```

The `Stillborn` class demonstrates that the object's destructors are not called if an exception is thrown inside the constructor. This is because the object does not really exist until the constructor is returned from.

Installing a Top-Level Exception Handler

An interesting feature in PHP is the ability to install a default exception handler that will be called if an exception reaches the top scope and still has not been caught. This handler is different from a normal `catch` block in that it is a single function that will handle *any* uncaught exception, regardless of type (including exceptions that do not inherit from `Exception`).

The default exception handler is particularly useful in Web applications, where you want to prevent a user from being returned an error or a partial page in the event of an uncaught exception. If you use PHP's output buffering to delay sending content until the page is fully generated, you gracefully back out of any error and return the user to an appropriate page.

To set a default exception handler, you define a function that takes a single parameter:

```
function default_exception_handler($exception) {}
```

You set this function like so:

```
$old_handler = set_exception_handler('default_exception_handler');
```

The previously defined default exception handler (if one exists) is returned.

User-defined exception handlers are held in a stack, so you can restore the old handler either by pushing another copy of the old handler onto the stack, like this:

```
set_exception_handler($old_handler);
```

or by popping the stack with this:

```
restore_exception_handler();
```

An example of the flexibility this gives you has to do with setting up error redirects for errors incurred for generation during a page. Instead of wrapping every questionable statement in an individual `try` block, you can set up a default handler that handles the redirection. Because an error can occur after partial output has been generated, you need to make sure to set output buffering on in the script, either by calling this at the top of each script:

```
ob_start();
```

or by setting the `php.ini` directive:

```
output_buffering = On
```

The advantage of the former is that it allows you to more easily toggle the behavior on and off in individual scripts, and it allows for more portable code (in that the behavior is dictated by the content of the script and does not require any nondefault `.ini` settings). The advantage of the latter is that it allows for output buffering to be enabled in every script via a single setting, and it does not require adding output buffering code to every script. In general, if I am writing code that I know will be executed only in my local environment, I prefer to go with `.ini` settings that make my life easier. If I am authoring a software product that people will be running on their own servers, I try to go with a maximally portable solution. Usually it is pretty clear at the beginning of a project which direction the project is destined to take.

The following is an example of a default exception handler that will automatically generate an error page on any uncaught exception:

```
<?php
function redirect_on_error($e) {
  ob_end_clean();
  include("error.html");
}
set_exception_handler("redirect_on_error");
ob_start();
// ... arbitrary page code goes here
?>
```

This handler relies on output buffering being on so that when an uncaught exception is bubbled to the top calling scope, the handler can discard all content that has been generated up to this point and return an HTML error page instead.

You can further enhance this handler by adding the ability to handle certain error conditions differently. For example, if you raise an `AuthException` exception, you can redirect the person to the login page instead of displaying the error page:

```
<?php
function redirect_on_error($e) {
  ob_end_clean();
  if(is_a($e, "AuthException")) {
    header("Location: /login/php");
  }
  else {
    include("error.html");
  }
}
set_exception_handler("redirect_on_error");
ob_start();
// ... arbitrary page code goes here
? >
```

Data Validation

A major source of bugs in Web programming is a lack of validation for client-provided data. *Data validation* involves verification that the data you receive from a client is in fact in the form you planned on receiving. Unvalidated data causes two major problems in code:

- Trash data
- Maliciously altered data

Trash data is information that simply does not match the specification of what it should be. Consider a user registration form where users can enter their geographic information. If a user can enter his or her state free form, then you have exposed yourself to getting states like

- New Yrok (typo)
- Lalalala (intentionally obscured)

A common tactic used to address this is to use drop-down option boxes to provide users a choice of state. This only solves half the problem, though: You've prevented people from accidentally entering an incorrect state, but it offers no protection from someone maliciously altering their `POST` data to pass in a non-existent option.

To protect against this, you should always validate user data in the script as well. You can do this by manually validating user input before doing anything with it:

```
<?php
$STATES = array('al' => 'Alabama',
                 /* ... */,
                'wy' => 'Wyoming');
function is_valid_state($state) {
  global $STATES;
  return array_key_exists($STATES, $state);
}
?>
```

I often like to add a validation method to classes to help encapsulate my efforts and ensure that I don't miss validating any attributes. Here's an example of this:

```
<?php

class User {
  public id;
  public name;
  public city;
  public state;
  public zipcode;
  public function _ _construct($attr = false) {
    if($attr) {
      $this->name = $attr['name'];
      $this->email = $attr['email'];
      $this->city = $attr['city'];
      $this->state = $attr['state'];
      $this->zipcode = $attr['zipcode'];
    }
  }
  public function validate() {
    if(strlen($this->name) > 100) {
      throw new DataException;
    }
    if(strlen($this->city) > 100) {
      throw new DataException;
    }
    if(!is_valid_state($this->state)) {
      throw new DataException;
    }
    if(!is_valid_zipcode($this->zipcode)) {
      throw new DataException;
    }
  }
}

?>
```

The `validate()` method fully validates all the attributes of the `User` object, including the following:

- Compliance with the lengths of database fields
- Handling foreign key data constraints (for example, the user's U.S. state being valid)
- Handling data form constraints (for example, the zip code being valid)

To use the `validate()` method, you could simply instantiate a new `User` object with untrusted user data:

```
$user = new User($_POST);
```

and then call validate on it

```
try {
    $user->validate();
}
catch (DataException $e) {
    /* Do whatever we should do if the users data is invalid */
}
```

Again, the benefit of using an exception here instead of simply having `validate()` return `true` or `false` is that you might not want to have a `try` block here at all; you might prefer to allow the exception to percolate up a few callers before you decide to handle it.

Malicious data goes well beyond passing in nonexistent state names, of course. The most famous category of bad data validation attacks are referred to as *cross-site scripting attacks*. Cross-site scripting attacks involve putting malicious HTML (usually client-side scripting tags such as JavaScript tags) in user-submitted forms.

The following case is a simple example. If you allow users of a site to list a link to their home page on the site and display it as follows:

```
<a href="<?= $url ?>">Click on my home page</a>
```

where `url` is arbitrary data that a user can submit, they could submit something like this:

```
$url ='http://example.foo/" onClick=bad_javascript_func foo="';
```

When the page is rendered, this results in the following being displayed to the user:

```
<a href="'http://example.foo/" onClick=bad_javascript_func foo="">
  Click on my home page
</a>
```

This will cause the user to execute `bad_javascript_func` when he or she clicks the link. What's more, because it is being served from your Web page, the JavaScript has full access to the user's cookies for your domain. This is, of course, really bad because it allows malicious users to manipulate, steal, or otherwise exploit other users' data.

Needless to say, proper data validation for any user data that is to be rendered on a Web page is essential to your site's security. The tags that you should filter are of course regulated by your business rules. I prefer to take a pretty draconian approach to this filtering, declining any text that even appears to be JavaScript. Here's an example:

```
<?php
$UNSAFE_HTML[] = "!javascript\s*:!is";
$UNSAFE_HTML[] = "!vbscri?pt\s*:!is";
$UNSAFE_HTML[] = "!<\s*embed.*swf!is";
$UNSAFE_HTML[] = "!<[^>]*[^a-z]onabort\s*=!is";
```

```
$UNSAFE_HTML[] = "!<[^>]*[^a-z]onblur\s*=!is";
$UNSAFE_HTML[] = "!<[^>]*[^a-z]onchange\s*=!is";
$UNSAFE_HTML[] = "!<[^>]*[^a-z]onfocus\s*=!is";
$UNSAFE_HTML[] = "!<[^>]*[^a-z]onmouseout\s*=!is";
$UNSAFE_HTML[] = "!<[^>]*[^a-z]onmouseover\s*=!is";
$UNSAFE_HTML[] = "!<[^>]*[^a-z]onload\s*=!is";
$UNSAFE_HTML[] = "!<[^>]*[^a-z]onreset\s*=!is";
$UNSAFE_HTML[] = "!<[^>]*[^a-z]onselect\s*=!is";
$UNSAFE_HTML[] = "!<[^>]*[^a-z]onsubmit\s*=!is";
$UNSAFE_HTML[] = "!<[^>]*[^a-z]onunload\s*=!is";
$UNSAFE_HTML[] = "!<[^>]*[^a-z]onerror\s*=!is";
$UNSAFE_HTML[] = "!<[^>]*[^a-z]onclick\s*=!is";

function unsafe_html($html) {
  global $UNSAFE_HTML;
  $html = html_entities($html, ENT_COMPAT, ISO-8859-1_
  foreach ( $UNSAFE_HTML as $match ) {
    if( preg_match($match, $html, $matches) ) {
      return $match;
    }
  }
  return false;
}
?>
```

If you plan on allowing text to be directly integrated into tags (as in the preceding example), you might want to go so far as to ban any text that looks at all like client-side scripting tags, as in this example:

```
$UNSAFE_HTML[] = "!onabort\s*=!is";
$UNSAFE_HTML[] = "!onblur\s*=!is";
$UNSAFE_HTML[] = "!onchange\s*=!is";
$UNSAFE_HTML[] = "!onfocus\s*=!is";
$UNSAFE_HTML[] = "!onmouseout\s*=!is";
$UNSAFE_HTML[] = "!onmouseover\s*=!is";
$UNSAFE_HTML[] = "!onload\s*=!is";
$UNSAFE_HTML[] = "!onreset\s*=!is";
$UNSAFE_HTML[] = "!onselect\s*=!is";
$UNSAFE_HTML[] = "!onsubmit\s*=!is";
$UNSAFE_HTML[] = "!onunload\s*=!is";
$UNSAFE_HTML[] = "!onerror\s*=!is";
$UNSAFE_HTML[] = "!onclick\s*=!is";
```

It is often tempting to turn on `magic_quotes_gpc` in you `php.ini` file. `magic_quotes` automatically adds quotes to any incoming data. I do not care for `magic_quotes`. For one, it can be a crutch that makes you feel safe, although it is simple to craft examples such as the preceding ones that are exploitable even with `magic_quotes` on.

With data validation (especially with data used for display purposes), there is often the option of performing filtering and conversion inbound (when the data is submitted) or outbound (when the data is displayed). In general, filtering data when it comes in is more efficient and safer. Inbound filtering needs to be performed only once, and you minimize the risk of forgetting to do it somewhere if the data is displayed in multiple places. The following are two reasons you might want to perform outbound filtering:

- You need highly customizable filters (for example, multilingual profanity filters).
- Your content filters change rapidly.

In the latter case, it is probably best to filter known malicious content on the way in and add a second filtering step on the way out.

Further Data Validation

Web page display is not the only place that unvalidated data can be exploited. Any and all data that is received from a user should be checked and cleaned before usage. In database queries, for instance, proper quoting of all data for insert should be performed. There are convenience functions to help perform these conversion operations.

A high-profile example of this are the so-called SQL injection attacks. A SQL injection attack works something like this: Suppose you have a query like this:

```
$query = "SELECT * FROM users where userid = $userid";
```

If `$userid` is passed in, unvalidated, from the end user, a malicious user could pass in this:

```
$userid = "10; DELETE FROM users;";
```

Because MySQL (like many other RDBMS systems) supports multiple queries inline, if this value is passed in unchecked, you will have lost your `user's` table. This is just one of a number of variations on this sort of attack. The moral of the story is that you should always validate any data in queries.

When to Use Exceptions

There are a number of views regarding when and how exceptions should be used. Some programmers feel that exceptions should represent fatal or should-be-potentially-fatal errors only. Other programmers use exceptions as basic components of logical flow control. The Python programming language is a good representative of this latter style: In Python exceptions are commonly used for basic flow control.

This is largely a matter of style, and I am inherently distrustful of any language that tries to mandate a specific style. In deciding where and when to use exceptions in your own code, you might reflect on this list of caveats:

- Exceptions are a flow-control syntax, just like `if{}`, `else{}`, `while{}`, and `foreach{}`.
- Using exceptions for nonlocal flow control (for example, effectively long-jumping out of a block of code into another scope) results in non-intuitive code.
- Exceptions are bit slower than traditional flow-control syntaxes.
- Exceptions expose the possibility of leaking memory.

Further Reading

An authoritative resource on cross-site scripting and malicious HTML tags is CERT advisory CA-2000-02, available at `www.cert.org/advisories/CA-2000-02.html`.

Because exceptions are rather new creatures in PHP, the best references regarding their use and best practices are probably Java and Python books. The syntax in PHP is very similar to that in Java and Python (although subtlely different—especially from Python), but the basic ideas are the same.