Techniques Every Expert Programmer Needs to Know

WHAT'S IN THIS CHAPTER?

- Understanding Object-oriented fundamentals in PHP
- Understanding `INNER` and `OUTER` JOINs
- Other `JOIN` syntax you should know
- Using MySQL Unions
- Using `GROUP BY` in MySQL queries
- Implementing MySQL Logical Operations and flow control
- Maintaining MySQL relational integrity
- Using subqueries in MySQL
- Utilizing advanced PHP regular expressions

This chapter covers the techniques that you, the proficient PHP and MySQL developer, should know and use before you tackle more advanced domain features in PHP and MySQL. The chapter starts with an in-depth overview of object-oriented programming techniques in PHP and object-oriented design patterns. As a PHP developer, you then become familiar with a number of core MySQL requirements for retrieving data including the different types of joins, `UNION`, `GROUP BY`, and subqueries syntax. This chapter also details the logic operators and flow control and techniques for maintaining relational integrity in MySQL. The chapter concludes with an in-depth review of advanced regular expressions in both PHP and MySQL.

OBJECT-ORIENTED PHP

Object-orientation has become a key concept behind proper PHP software design. This book follows the idea that in properly designed software, all business logic (the rules that drive how an application behaves) should be object oriented. The only exception is when small scripts act as a view or a way to display data returned from other objects.

Taking this approach solves a few problems because it:

- Makes it easy to extend the functionality of existing code.
- Allows for type hinting, which gives greater control over what variables are passed into functions.
- Allows for established design patterns to be used to solve common software design problems and makes debugging much easier.

This section covers object-oriented PHP and key design patterns in depth. Later chapters cover even more advanced object-oriented topics.

Instantiation and Polymorphism

The two key benefits of object-oriented programming in PHP are the ability to abstract data into classes and for the application to act on those structures. It is important to understand polymorphism, which is when one object appears and can be used in the same way as another object of a related type. It stands to reason that if B is a descendant of A and a function can accept A as a parameter, it can also accept B.

Three classes are used in this chapter while covering polymorphism:

- `Node`
- `BlogEntry`
- `ForumTopic`

In this application both `BlogEntry` and `ForumTopic` are siblings of each other and descendants of `Node`. This is a good time to become familiar with the example code that comes with the book. The code archive contains a folder for every chapter. Each folder has a `.class.php` file and a `.test.php` file for every class. SQL files aren't used in this section, but when they are, they have a `.sql` extension.

The typical class looks a lot like this:

```
class ClassNameHere extends AnotherClass {
  public function someFunction() {
    parent::someFunction();
  }
};
```

The `parent` keyword is used to directly reference a variable or method in the parent class, bypassing any variables or methods of the same name in the current class. The method is marked as `public`, which is a familiar concept for object-oriented programming but relatively new to PHP.

Older PHP applications will not define a visibility for the member methods. When the visibility is not defined it is assumed to be public. A member variable or method (function inside a class) can have one of three visibilities:

- **`public`** indicates that the member is accessible globally across all of PHP.
- **`private`** indicates that a member can be accessed only from within the class in which it is defined. Private members cannot be overridden in later classes because those classes too do not have access to the member.
- **`protected`** indicates that the member can be accessed only by the class in which it is defined and all descending classes.

Additionally, three other keywords can augment `private`, `public`, and `protected`. They are `static`, `abstract`, and `final`:

- **`static`** members are not tied to particular instances of a class and can be accessed by any instance. They should be used sparingly but are very useful for shared variables across all instances. The `static` keyword can also be used inside methods and functions to define a variable that is global to all calls to that function. Both uses are relied upon by later examples in this chapter.
- **`abstract`** methods must be implemented in all classes that descend from that class that defines it. Abstract methods can only be defined in classes that are marked as abstract. It is not possible to directly instantiate an abstract class because of the nature of abstraction.
- **`final`** methods can never be redefined in descending classes and therefore their functionality cannot be changed.

Variables inside a class can also be declared constant using `const`. Constants are always `public static` and their value can never be changed at run time. Unlike normal variables, constants cannot have a dollar sign in front of them and by convention are always capitalized.

Each and every type of visibility is used throughout this book. The next section covers most of them by using the three classes described previously.

Polymorphism in Action

The three classes mentioned previously need to be defined in order to be useful. The goal of this section is not to create a fully functioning application but rather to demonstrate techniques that are the core of the rest of the book. The first class to be defined is the `Node` class as shown in Listing 1-1.

LISTING 1-1: NODE.CLASS.PHP

Available for download on Wrox.com

```
<?php
abstract class Node {
  private $debugMessages;

  public function __construct() {
    $this->debugMessages = array();
    $this->debug(__CLASS__." constructor called.");
  }

  public function __destruct() {
    $this->debug(__CLASS__." destructor called.");
    $this->dumpDebug();
  }

  protected function debug( $msg ) {
    $this->debugMessages[] = $msg;
  }

  private function dumpDebug( ) {
    echo implode( "\n", $this->debugMessages);
  }

  public abstract function getView();
}
?>
```

The `Node` class is abstract and therefore cannot be instantiated. However, it can have private members. The descendant classes will not be able to access the private members directly but the members can be accessed from other more visible methods inside of `Node`. In the node class, the member variable `$debugMessage` is being accessed from several methods and `dumpDebug()` is a private method being called from the destructor. For the purpose of this example, both `ForumTopic` and `BlogEntry` are identical in all regards except name. The magic constant `__CLASS__` will be used to tell them apart as shown in Listing 1-2.

Available for download on Wrox.com

LISTING 1-2: ForumTopic.class.php

```
<?php
class ForumTopic extends Node {
  private $debugMessages;

  public function __construct() {
    parent::__construct();
    $this->debug(__CLASS__." constructor called.");
  }

  public function __destruct() {
    $this->debug(__CLASS__." destructor called.");
    parent::__destruct();
  }

  public function getView() {
    return "This is a view into ".__CLASS__;
  }
}
?>
```

Now it is time to run some tests and see what happens. The first test is to create an instance of each subclass and observe the debug output. The entire test is just one line of code but has a several lines of output:

```
$forum = new ForumTopic();
/* Output:
Node constructor called.
ForumTopic constructor called.
ForumTopic destructor called.
Node destructor called.
*/
```

The output shows that the constructor for each class is called and that it bubbles down appropriately to the parent class before adding its own debug message. The opposite is true for the destructor. Whether the parent class is called first, last, or in the middle of a method can be determined at design time for each specific class. However, in general, because the constructors and destructors for descendant classes often reference the variables from the parent, it is a good practice to call the parent at the beginning of the constructor and end of the destructor.

Almost as important is the output demonstrating that the `__CLASS__` variable is always equal to the name of the class in which the function being called is defined. It is not necessarily the same as the output of `get_class($this)`. The `get_class()` method returns the name of the class that was instantiated. In non-technical terms this method always returns the class name that directly follows the `new` keyword when instantiating the object.

A WORD ON THE DESTRUCTOR

The destructor, in this case, was never explicitly called. Unless the script ends in a fatal error, the destructor for any remaining objects will always be executed when the script completes. The garbage collector will also fire the destructor immediately if the number of references to an object goes to zero. In this case the destructor is what dumps the debug output to the screen, so it is simple to test to see if the garbage collector is doing its job:

```
$topic = new ForumTopic();

echo "---------------------\n";
$topic = null;
echo "---------------------\n";

/* Output:
---------------------
Node constructor called.
ForumTopic constructor called.
ForumTopic destructor called.
Node destructor called.
---------------------
*/
```

continues

However, if there is another variable thrown into the mix the situation becomes much different:

```
$topic = new ForumTopic();
$reference = $topic;

echo "---------------------\n";
$topic = null;
echo "---------------------\n";

/* Output:
---------------------
---------------------
Node constructor called.
ForumTopic constructor called.
ForumTopic destructor called.
Node destructor called.
*/
```

Class instances are always passed by reference unless explicitly cloned. Using the reference operator on a class variable like `$reference = &$topic;` will not increase the reference count for the object and will therefore not prevent it from being garbage collected. The code, in effect, is creating a reference to a reference.

Handling Terminal Types and Type Hinting

One practical application of `get_class()` is explored later in this book; however, it is not always helpful to determine just the terminal type of an object. For example, in almost every case it is wrong to execute code only if the output of `get_class()` matches a string. After all, what happens if the class is subclassed? Shouldn't the subclasses also pass the test?

The keyword that solves the issue is `instanceof`. It evaluates to `true` if the operand on the left is of the type on the right or any subclasses of that type. For example, if a method takes an arbitrary parameter but should execute specific code if the variable is a `Node` object, it can be written like this:

```
if ( $foo instanceof Node ) ...
```

In this case `$foo` can be an instance of `ForumTopic` or `BlogEntry`. It cannot be an instance of `Node` only because `Node` is abstract and cannot be instantiated. PHP also supports type hinting, which allows a method or function to take only an object of a set type or its descendants. Type hinting, unfortunately, is not available for primitive types such as string and integer:

```
function print_view( Node $node ) {
  echo "Printing the view for ".get_class($node)."\n";
  echo $node->getView()."\n";
}
```

Class methods should use type hinting whenever possible to improve the maintainability of the code. Code that uses type hinting is less error-prone and partially self-documenting.

Interfaces

Interfaces are structures for defining functionality that a class must implement. An interface does not dictate the inner workings of that functionality. Think of interfaces as templates that classes need to adhere to. Chapter 2 makes heavy use of some of PHP's built-in interfaces.

Classes do not inherit interfaces because only the method signatures and return types are defined within them. Instead they *implement* the interface. However, it is occasionally useful to derive one interface from another. For example, an interface called `Iterator` may be used as the base interface for a new interface called `RecursiveIterator` that defines all the functionality of the standard `Iterator` interface but also defines new functionality.

An interface is never instantiated directly. However, variables can be tested against interfaces. Testing against an interface ensures that an object implements all the methods of the interface before attempting to call a method. For example, say the interface `PageElement` defines a `getXML()` method:

```
if ( $object instanceof PageElement )
  $body->appendChild( $object->getXML( $document ) );
```

Interfaces are defined in a similar way to classes. Instead of `class` the keyword `interface` is used. Two other important distinctions are that all methods inside an interface must always be defined as public and methods do not have a body. Consider Listing 1-3 which shows a new interface called `ReadableNode`:

LISTING 1-3: ReadableNode.interface.php

Available for download on Wrox.com

```
<?php
interface ReadableNode {
  public function isRead();
  public function markAsRead();
  public function markAsUnread();
};
?>
```

You can then create a reusable utility function `markNodeAsRead()` to check if a node is readable and to call the `markAsRead()` method if it is.

```
function markNodeAsRead( $node ) {
  if ( $node instanceof ReadableNode )
    $node->markAsRead();
}
```

Interfaces are useful for defining sets of functionality when it is not important how the methods are implemented. Because PHP doesn't have multi-inheritance they are also useful for defining classes that have a collection of disparate functionality but still need the benefits of polymorphism and type hinting. Unlike inheritance, a class can implement as many interfaces as it desires. Also, an interface can extend multiple other interfaces. For example, if a class is both `Readable` and `Deletable`:

```
<?php
interface MessagingNode extends Readable, Deletable {
};

class ForumTopic extends Node implements Readable, Deletable {
  …
};

class BlogEntry extends Node implements MessagingNode {
  …
}
?>
```

In this case both the classes `ForumTopic` and `BlogEntry` must implement every method found in both the interface `Readable` and `Deletable`. In this case the new `MessagingNode` interface is little more than shorthand.

Magic Methods and Constants

Before diving into design patterns it is necessary to review magic methods inside PHP. *Magic methods* are specially named methods that can be defined in any class and are executed via built-in PHP functionality. Magic methods always begin with a double underscore. In fact, the magic methods `__destruct()` and `__construct()` have already been used several times in this chapter. It is not good practice to write user-defined functions and methods that begin with the double underscore in case PHP implements methods with those names in future versions.

Magic constants are used to access certain read-only properties inside PHP. Magic constants both begin and end with a double underscore and are always capitalized. The constant `__CLASS__` has been used several times in this chapter to output the name of the class in which the code is defined.

Practical Use of Magic Constants

It is often useful to determine where in the code output originates. This is the purpose of all of the magic constants and is particularly useful when writing custom logging functions. The seven magic constants are as follows:

- **`__CLASS__`** equates to the class in which the constant is referenced. As noted earlier, this variable is always equal to the class in which it is defined, which is not always the class that was instantiated. In the previous example, `__CLASS__` as defined inside `Node` always returns `Node` even if the method is part of an object that was instantiated as a descendant class. In addition to debugging, the class constant is also useful for static callback functions.
- **`__FILE__`** is always equal to the filename where the constant is referenced.
- **`__LINE__`** is used in conjunction with `__FILE__` in order to output a location in code. For example:

  ```
  error_log('Notice: Placeholder class. Don't forget to change before
  release! In '.__FILE__.' on line '.__LINE__);
  ```

Both `_FILE_` and `_LINE_` are relative to the file currently executing regardless of whether that file was included or required from a different file.

- **`__DIR__`** functions exactly like `dirname(__FILE__)` and returns the absolute directory in which the file is located. It is useful for specifying absolute paths, which are sometimes faster than relative paths; particularly when including scripts.
- **`__FUNCTION__`** and **`__METHOD__`** make it possible to determine the function or method name, respectively, using magic constants. When possible, these constants should be used in place of hard-coding the function name.
- **`__NAMESPACE__`** is the seventh and final magic constant. As the name suggests, it is equal to the current namespace.

As a debugging mechanism using the magic constants is very basic. More advanced techniques for debugging are discussed in depth in Chapter 16.

Adding Magic Functionality to Classes

Although the magic methods `__construct()` and `__destruct()` are the most commonly used, several more exist. When using design patterns it becomes necessary to expand on certain built-in functionality of PHP. This section first covers the cases where each magic method is useful and then illustrates the use of the method. The first set of methods has to do with data representation.

In many cases it is useful to have a string representation of an object so you can output it to the user or another process. Referencing an object as a string will, by default, evaluate to the object's ID in memory, which in most cases is less than ideal. PHP provides a standard way of overriding this default functionality and returning any desirable string representation. A numeric class might return the number as a string, a user class might return a username, a node class might return a node title, an XML node might return the text content of the node, and so on. The magic method used for this functionality is `__toString()`. The method is triggered in any situation where an object is used as a string, for example: `echo "Hello $obj";`. It can also be called directly like any other normal public method, which is preferable to hacks such as appending an empty string to force coercion.

Serialization is another process integral to PHP applications that store state or cache entire objects. It generates a string representation of an object. Serialization is done manually by calling `serialize()` and is reversed with `unserialize()`. Both methods work on any PHP variable (except a resource such as a MySQL handle) without any modification. However, sometimes it is necessary to clean up a complex object prior to serialization.

Classes can implement the magic method `__sleep()`, which is called immediately before serialization. It is expected to return an array where the values are the member variables that should be saved. Member variables can be `public`, `private`, or `protected`. Likewise, `__wakeup()` is called when you restore the object. One use for these functions is to ignore a resource handle on sleep and to then reopen the handle on restoration as shown in Listing 1-4.

LISTING 1-4: FileLog.class.php

Available for download on Wrox.com

```
<?php
class FileLog {
  private $fpointer;
  private $filename;

  function __construct( $filename ) {
    $this->filename = $filename;
    $this->fpointer = fopen($filename,'a');
  }

  function __destruct() {
    fclose($this->fpointer);
  }

  function __sleep() {
    return array( "filename" );
  }

  function __wakeup() {
    $this->fpointer = fopen($this->filename,'a');
  }

  public function write( $line ) {
    fwrite( "$line\n", $this->fpointer );
  }
};

/*
  Example usage:
    $log = new FileLog( "debug.txt" );
    $data = serialize( $log );
    $log = null;
    $log = unserialize($data);
    echo $data;
  Example output:
    O:7:"FileLog":1:{s:17:"FileLogfilename";s:9:"debug.txt"}
*/
?>
```

The serialized data, as seen in the comments of the previous example, contains the data type followed by the length of the data and then the data itself. A semicolon separates multiple members and each member has two variables. The first variable is the name and the second is the value.

When serializing, private member variables have the class name prepended to them, whereas protected variables have an asterisk prepended. In both cases the prefix is surrounded by two null bytes. The bytes cannot be seen in print, however. Looking closely at the string `s:17:"FileLogfilename"` it becomes apparent that the string is only 15 printable characters in length. The remaining two characters are the null bytes before and after the word `FileLog`.

The next four magic methods have to do with retrieving, inspecting, and storing inaccessible member variables. They are `__set()`, `__unset()`, `__get()`, and `__isset()`. Each is invoked when trying to access a member variable that is not available to the context that is requesting it. That can mean that a variable marked as private or protected and accessed outside the scope or that a member variable does not exist. Both `__unset()` and `__isset()` are triggered by the functions with the same name (sans the underscores) in PHP. All these methods are used extensively in the section "Design Patterns" so they won't be covered in any more detail in this section.

Similarly, the method __call() is invoked when you try to call a method that is either undefined or inaccessible. A similar method named __callStatic() is called for static methods.

Three magic methods won't be covered in this chapter. __set_state() is used when you import a class via a call to var_export() and is worth looking into if an application does a lot of dynamic code evaluation. __clone() is invoked if you try to make a clone of an object and you can use it for various processes. The third method, __invoke(), is used when an object is being called as if it were a function; it is covered more in Chapter 2.

The next section discusses the eight design patterns that you can use in applications for cleaner and more readable code as well as to solve common problems in software design.

Design Patterns

This section covers design patterns in PHP. The eight patterns that are covered in this section are:

- Singleton
- Multiton
- Proxy
- Façade
- Decorator
- Factory
- Observer Pattern
- Publisher/subscriber

Singleton and Multiton Patterns

The singleton and less common multiton patterns control the number of instances of a class in an application. As the names imply, a *singleton* can be instantiated only once and a *multiton* any number of times. In the case of the latter, there can be only one instance for any given key.

Because of the nature of singletons they are often used for configuration and for variables that need to be accessed from anywhere in the application. Using singletons is sometimes considered poor practice because it creates a global state and does not encapsulate all the functionality of the system in a single root object. In many cases this can make unit testing and debugging more difficult. This book leaves the reader to make his or her own decision regarding these patterns. In general, some object orientation is better than none. Listing 1-5 shows an example of a singleton pattern.

LISTING 1-5: SingletonExample.class.php

Available for download on Wrox.com

```
<?php
class SingletonExample {
  public static function getInstance() {
    static $instance = null;
    if ( $instance == null ) {
      $instance = new SingletonExample();
    }
    return $instance;
  }
};
?>
```

The singleton class makes use of both functions of the keyword `static`. The first is in the method definition, indicating that the method is not associated with any particular instance of the class. The second is in the method itself. The keyword `static`, when placed in front of a local variable in a function or method indicates that all calls to that method, regardless of what object the call is made to, will share that variable.

In the case of the singleton, the variable `$instance` is initialized to `null` and retains whatever value is set to it across all calls to the method. On first execution it is always `null`. On later executions it is always the same instance of the `SingletonExample` object. Making a single static method call ensures retrieval of the same instance every time:

```
$singleton = SingletonExample::getInsance();
```

A multiton is similar except that it requires a key to be passed to the `getInstance()` function. For a given key there can be only one instance of the object. This pattern is useful when dealing with many nodes that have unique identifiers that can appear multiple times in a single execution (such as a node in a Content Management System). Multitons save memory and ensure that there aren't multiple conflicting instances of the same object. The `SingletonExample` class can be quickly modified to be a multiton instead as shown in Listing 1-6.

LISTING 1-6: MultitonExample.class.php

Available for download on Wrox.com

```
<?php
class MultitonExample {
  public static function getInstance( $key ) {
    static $instances = array();
    if ( !array_key_exists( $key, $instances ) ) {
      $instances[$key] = new MultitonExample();
    }
    return $instance[$key];
  }
};
?>
```

Because PHP objects are always passed by reference it is ensured that each instance returned from a multiton or singleton object is consistent throughout the application. You must be careful when using these patterns with serialization or with the `clone` keyword because either action may result in multiple versions of what should be the same object.

Multitons and singletons are similar in concept to lazy initialization. In *lazy initialization*, object initialization that requires a significant amount of processing or memory is delayed until the object is needed. This usually consists of a conditional to check to see if the object exists, followed by a return of either the existing object or a new one — much like in the two previous patterns. The book sometimes uses lazy initialization for database handles or data sets to avoid spending resources that are not needed by the application.

Proxy and Façade Patterns

Proxy and façade patterns are grouped together because they each provide abstraction for more complex functionality. How abstraction is achieved differs for both patterns.

In the case of a proxy, all methods and member variables are routed to the destination object. The proxy can, if it is desirable, modify or inspect the data as it passes through. The magic methods make implementing this pattern very easy in PHP. One use for this pattern is to log method access. It could also be used to determine code coverage or to just debug an issue (see Listing 1-7):

Available for download on Wrox.com

LISTING 1-7: LoggingProxy.class.php

```
<?php

class LoggingProxy {
  private $target;

  function __construct( $target ) {
    $this->target = $target;
  }

  protected function log( $line ) {
    error_log($line);
  }

  public function __set( $name, $value ) {
    $this->target->$name = $value;
    $this->log("Setting value for $name: $value");
  }

  public function __get( $name ) {
    $value = $this->target->$name;
    $this->log( "Getting value for $name: $value" );
    return $value;
  }

  public function __isset( $name ) {
    $value = isset($this->target->$name);
    $this->log( "Checking isset for $name: ".($value?"true":"false") );
    return $value;
  }

  public function __call( $name, $arguments ) {
    $this->log( "Calling method $name with: ".implode(",",$arguments) );
    return call_user_func_array( array($this->target,$name), $arguments );
  }

};

?>
```

The `LoggingProxy` example uses callback functions in the `__call()` method. The purpose of the method is to call a defined function within the target class. The class also makes liberal use of variable member names. Variable member names and callbacks are covered in greater detail in Chapter 2.

A proxy can be as simple as the preceding one or as complex as needed. In most cases a proxy should not change the behavior of the class that it is a proxy for; however, it is possible to do that as well. It is also possible for the proxy to be an interface into an entirely different system. For example, it may be useful to have a MySQL database proxy that executes stored procedures or a proxy that is an interface to XML Remote Procedure Calls.

One disadvantage of a proxy is that it is not of the same type as the class it is a proxy for. Therefore it cannot be used in situations where type hinting is necessary or when the code checks to ensure that an object is of a certain type.

The façade pattern serves a different purpose. It is meant to abstract complex functionality so that the application does not need to know the details around which subsystem handles each request. For example, if making a typical API request requires that a user be authenticated via a user subsystem, the request is made to a remote server with an API subsystem, and then the response is decoded via a function from a different API, the resulting façade method looks like this:

```
public function apiRequestJson( $method, $parameters ) {
  $user = User::getAuthenticatedUser();
  if ( $user->hasPermission( $method ) ) {
    $result = $this->api->$method( $parameters );
    return json_decode( $result );
  }
}
```

Façades do not add new functionality but rather delegate the responsibilities to the appropriate subsystem. The subsystems do not need to know of the existence of a façade and the application does not need to know about the existence of the subsystems.

Sometimes it becomes necessary to extend the functionality of a class while maintaining object integrity and allowing for type hinting. The ideal pattern for that is the decorator pattern.

Decorator Pattern

The decorator pattern extends the functionality of a class similar to standard inheritance. Unlike standard inheritance, the decorator pattern can add functionality dynamically at run time if an object has already been instantiated. This action is referred to as *decorating* the object. One benefit of decoration is that it allows any combination of decorators to extend the same object. For example, a car might have an option for an in-car navigation system and an option for leather seats. A customer may want just the seats or may want just the navigation system. Using this pattern the combination can be dynamic.

Taking the car example a step further, you can create a series of classes for decorating the car. To make things easier, all car decorations will extend from a `CarDecorator` class. It is also possible for decorators to extend other decorators. For instance, the user may be able to upgrade from a basic radio to a CD player/radio combination to a multi-disc CD player with radio. A chain of inheritance can be created because a multi-disk CD player with radio shares all the functionality of a basic radio. For simplicity, the following examples assume that the only two methods in the class Car are getPrice() and getManufacturer() as shown in Listing 1-8:

Available for download on Wrox.com

LISTING 1-8: AbstractCar.class.php

```
<?php

abstract class AbstractCar {
  public abstract function getPrice();
  public abstract function getManufacturer();
};

?>
```

The car class extends the AbstractCar class and must implement all the methods in the abstract class. The result is the car without any decorators added as shown in Listing 1-9.

Available for download on Wrox.com

LISTING 1-9: Car.class.php

```
<?php

class Car extends AbstractCar {
  private var $price = 16000;
  private var $manufacturer = "Acme Autos";

  public function getPrice() { return $this->price; }
  public function getManufacturer() { return $this->manufacturer; }
};

?>
```

The CarDecorator class also extends AbstractCar. It serves as the base class for all future decorators. The purpose of the class is to act as a proxy into the real implementation, which in this case is called the target. Because the base price for the car exists not in the decorator object but in the target, it is necessary for getPrice() to query the price from the target object as shown in Listing 1-10.

Available for download on Wrox.com

LISTING 1-10: CarDecorator.class.php

```
<?php

class CarDecorator extends AbstractCar {
  private var $target;

  function __construct( Car $target ) { $this->target = $target; }

  public function getPrice() { return $target->getPrice(); }
  public function getManufacturer() { return $target->getManufacturer(); }
};

?>
```

The first step is complete, creating a `CarDecorator` class that extends from `AbstractCar`. The car decorator could be used directly but it wouldn't serve much purpose. For now all it does is forward all requests to the target `Car` object. Extending both `Car` and its decorator from an abstract class allows the decorators to avoid the overhead of extending a complete `Car` object but still maintain its polymorphic properties.

The next step is to define a *concrete decorator*. Once the base decorator is created it becomes easy to implement new decorators as shown in Listing 1-11:

Available for download on Wrox.com

LISTING 1-11: NavigationSystem.class.php

```
<?php

class NavigationSystem extends CarDecorator {
  public function getPrice() { return parent::getPrice()+1000; }
};

?>
```

The pattern can be particularly useful in ecommerce applications but it is also commonly used in graphical applications. An icon may decorate a text box; or a scroll bar may decorate a canvas. In the previous example getting the price of a car that has a navigation system and leather seats is just three lines of code:

```
<?php

$car = new Car();
$car = new NavigationSystem( $car );
$car = new LeatherSeats( $car );
echo $car->getPrice();

?>
```

When using decorators in this manner, it is technically possible for multiple instances of the same decorator to decorate an object. Having two navigation systems in one car doesn't make any sense. A simple function can be added to the `CarDecorator` class to check to see if a decorator is being used:

```
public function hasDecoratorNamed( $name ) {
  if ( get_class($this) == $name )
    return true;
  else if ( $this->target instanceof CarDecorator )
    return $this->target->hasDecoratorNamed( $name );
  else
    return false;
}
```

The decorator can be combined with a proxy pattern to create additional functionality at run time. For example, if the code were to implement all the functionality of a car, the `NavigationSystem` class may add a `turnOnNavigation()` method. Because the method to turn on navigation would only be available in the navigation decorator it becomes necessary to proxy call to unknown methods through to the target.

Factory Method

The factory method pattern is a creational pattern much like singletons, multitons, and lazy initialization. *Factory methods* are used to return an instance of an object that is a subclass of the object containing the factory method. One simple example is a class called `GDImage` that will take a valid image filename and return an appropriate image object as shown in Listing 1-12.

LISTING 1-12: GDImage.class.php

Available for download on Wrox.com

```
<?php

abstract class GDImage {
  public static function createImage( $filename ) {
    $info = getimagesize( $filename );
    $type = $info[2];

    switch ( $type ) {
      case IMAGETYPE_JPEG:
       new new JPEGImage( $filename );

     case IMAGETYPE_PNG:
       new new PNGImage( $filename );

      case IMAGETYPE_GIF:
       new new GIFImage( $filename );
    }

    return null;
  }
};

?>
```

In the GD example, the classes `PNGImage`, `GIFImage`, and `JPEGImage` would all descend from the common class `GDImage`. Pure implementations of the factory design pattern will always define factory methods as static. Additionally, `GDImage` should be treated as an abstract class and never be directly instantiated.

Another use for factory methods is for unit testing. A factory might return a working valid object under normal conditions but return a dummy object under test conditions. This is useful because using a live object both requires a fully functional data service and can possibly modify real data. For example, a class called `User` may return an `AuthenticatedUser` if the system is not in testing mode or a `TestUser` if the system is in testing mode and `AuthenticatedUser` is not the direct subject of the test.

The factory method can be implemented in nearly any situation where a different class needs to be instantiated depending on the type of data. There can also be more than one factory method per class. For example, the `GDImage` class in the previous example may have a second factory method called `createFromString()` that returns the appropriate object based on a binary input.

Observer and Publisher/Subscriber Patterns

The observer pattern and the publisher/subscriber pattern are more common in event-based architectures than they are in most stateless server-side Internet applications; however they do have uses in PHP. The observer pattern is simpler to implement and is sufficient in most cases so it is covered first.

In the observer pattern the observer must know what objects are broadcasting the events that they want to listen for. It is a sniper rifle approach to event handling. When an event happens on the publisher object, it notifies all observers at once. But if another object fires the same event, it is not broadcasted unless the observer is also watching that object. In Listing 1-13, a simple reusable observer system can be defined with one class and one interface:

LISTING 1-13: Observer.interface.php

Available for download on Wrox.com

```
<?php
interface Observer {
  public function notify( $event );
};
?>
```

The observable object then contains a method that can be used to register an observer as shown in Listing 1-14.

LISTING 1-14:OBSERVABLEOBJECT.CLASS.PHP

Available for download on Wrox.com

```
<?php
class ObservableObject {
  private function $observers = array();

  public function observe( Observer $observer ) {
    $this->observers[] = $observer;
  }

  public function dispatch( $event ) {
    foreach ( $this->observers as $observer )
      $observer->notify( $event );
  }
};
?>
```

A class that wants to broadcast events extends `ObservableObject` and any class that wants to listen to events can simply implement the interface `Observer`. In a more complex system the observer can specify the type of event that it wants to listen for.

The publisher/subscriber pattern is similar except that it decouples the subscribers (observers) from the publishers (observable objects). Instead a new class is introduced called an `Event`. The observer subscribes for notification whenever the event is triggered anywhere in the application instead of observing events on just a single class. The observer does not know or care what classes can publish the event.

Some systems implement the publisher/subscriber pattern using a controller as a delegate for all events. That method requires that all events be registered in a central location. For simplicity, the event object itself will act as a delegate (Listing 1-15):

LISTING 1-15: BroadcastingEvent.class.php

Available for download on Wrox.com

```
<?php
class BroadcastingEvent {
  private static $observers = array();

  public static function subscribe( Observer $observer ) {
    self::$observers[] = $observer;
  }

  public function publish() {
    foreach ( self::$observers as $observer )
      $observer->notify( $this );
  }
};
?>
```

The `BroadcastingEvent` class and the `ObservableObject` class both look very similar. Two changes are that the array of observers is now a static variable in the class instead of an instance variable and the event type no longer needs to be passed to the dispatching function because dispatching is a method of the event itself.

The major paradigm shift is that the observers and the dispatching object no longer need to have any knowledge of each other. This decoupling allows for an observer/subscriber to listen for all events of that type without needing specific application knowledge.

In a PHP application, this pattern is used in systems that can be dynamically extended such as Content Management Systems. For example, an object can listen for a user load event and take specific actions. A CMS should not require implementation-level knowledge of all its modules.

USING MYSQL JOINS

Retrieving data from a normalized relational database that contains many tables generally involves the use of joins in a `SELECT` statement. A join in MySQL queries enables you to select or manipulate data from multiple tables in a single SQL statement.

The SQL standard provides various different join operations such as `INNER JOIN`, `OUTER JOIN`, `STRAIGHT_JOIN`, and `NATURAL JOIN`. MySQL implements most common join syntax; however, your expectation may differ between different relational database products.

The following examples use two simple base tables to demonstrate various different joins. The first table contains colors, and second table contains the colors of country flags. The sample data includes the following rows:

For simplicity, this data is de-normalized to demonstrate the various possible MySQL join syntax. These table structures may not necessarily represent optimal database schema design (see Table 1-1). To construct this table and data for all examples in this section, see the code file create-tables.sql

TABLE 1-1: Schema Tables

TABLE	VALUES
Colors	Red, White, Blue, Green, Black
Flags	USA, Australia, Canada, Japan, Sweden

To understand joins with multiple tables, you can use the concept of sets and the mathematical visual approach of Venn diagrams. This shows the interaction between various sets and therefore the types of joins that can be used to retrieve information. See `http://en.wikipedia.org/wiki/Venn_diagram` for more background information on Venn diagrams.

Figure 1-1 shows the Venn diagram of two individual sets of information.

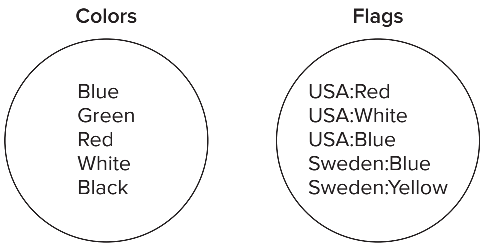

FIGURE 1-1

If you wanted to know the colors that are in the USA flag, you could use the following `SELECT` statement to retrieve the necessary rows as described in Figure 1-1. This is shown in Listing 1-16.

Available for download on Wrox.com

LISTING 1-16: simple-select.sql

```
SELECT color
FROM flags
WHERE country='USA';

+-------+
| color |
+-------+
| Blue  |
| Red   |
| White |
+-------+
```

Note the following from the previous listing:

- In line 1 you specify the column(s) you want to retrieve.
- In line 2 you specify which table you want to retrieve these column(s) from.
- In line 3 you specify any criteria or condition where you want to restrict the types of rows you want to retrieve.

Figure 1-2 shows the Venn diagram of the intersection of these two sets, and also two exception sets of information.

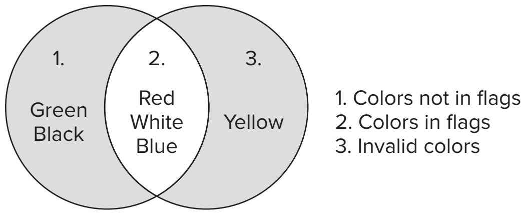

FIGURE 1-2

INNER JOIN

If you want to know more about the attributes of the colors for the USA flag, you can use an `INNER JOIN`, as shown in Listing 1-17, with the colors table to retrieve more information.

LISTING 1-17: inner-join.sql

Available for download on Wrox.com

```
SELECT flags.color, colors.is_primary, colors.is_dark, colors.is_rainbow
FROM   flags
INNER JOIN colors ON flags.color = colors.color
WHERE  flags.country='USA';

+-------+------------+---------+------------+
| color | is_primary | is_dark | is_rainbow |
+-------+------------+---------+------------+
| Blue  | yes        | yes     | yes        |
| Red   | yes        | no      | yes        |
| White | yes        | no      | no         |
+-------+------------+---------+------------+
```

Note the following for the previous Listing:

- Line 1 selects additional columns from the colors table.
- Line 3 specifies an `INNER JOIN` with the colors table and the flags table, and states that you want to join on the color column in flags with the color column in colors.

The Table Alias

When working with joins in MySQL it is common practice to alias tables used in the SQL query. You can very easily rewrite the previous example as shown in Listing 1-18.

Available for download on Wrox.com

LISTING 1-18: inner-join-alias.sql

```
SELECT f.color, c.is_primary, c.is_dark, c.is_rainbow
FROM   flags f
INNER JOIN colors c ON f.color = c.color
WHERE  f.country='USA';
```

For each table, you can optionally specify an alias after the table in the `FROM` clause. There are no general restrictions on the length of the alias; however it is best practice to use appropriate naming standards for your application. A table alias in MySQL has a maximum 256 characters in length, whereas a table name has only 64 characters.

ON and USING

For a join command, the `ON` syntax is of the format `table1.column_name = table2.column_name`.

When your schema design names columns in an identical fashion between join tables, you can shortcut the `ON` syntax with the `USING` syntax in the format `USING(column_name)`. For an example see Listing 1-19.

Available for download on Wrox.com

LISTING 1-19: inner-join-using.sql

```
SELECT f.color, c.is_primary, c.is_dark, c.is_rainbow
FROM   flags f
INNER JOIN colors c USING (color)
WHERE  f.country='USA';
```

In line 3 you will see the `USING` syntax as an alternative to the ON syntax in the previous SQL example.

> *When the column name between two tables is the same, you can simply use the* `ON` *syntax with the* `USING` *syntax. It is a good practice to use appropriate database naming standards, and specify columns with the same name when they contain the same data in different tables.*

An Alternative INNER JOIN Syntax

You can also use the comma (,) syntax for specifying an `INNER JOIN` as shown in Listing 1-20.

Available for download on Wrox.com

LISTING 1-20: inner-join-comma.sql

```
SELECT f.color, c.is_primary, c.is_dark, c.is_rainbow
FROM   flags f, colors c
WHERE  f.country='USA'
AND    f.color = c.color;
```

This comma syntax is a common and well-used approach; however, it does not provide the best readability for a software developer. With this comma syntax the join columns and restriction criteria are all specified in the `WHERE` clause, unlike with the `INNER JOIN` syntax where the `ON` or `USING` defines the join between each table when the table is specified, and the `WHERE` restricts the rows of results based on the table join. Overall this improves readability and decreases the possibility of missing a join column in a more complex multitable statement.

Listing 1-21 shows an example where you miss a join between two tables because it is not defined in the `WHERE` clause:

Available for download on Wrox.com

LISTING 1-21: missing-where-join.sql

```
SELECT f.color, c.is_primary, c.is_dark, c.is_rainbow
FROM   flags f, colors c
WHERE f.country='USA';

+-------+------------+---------+------------+
| color | is_primary | is_dark | is_rainbow |
+-------+------------+---------+------------+
| Blue  | no         | yes     | no         |
| Red   | no         | yes     | no         |
| White | no         | yes     | no         |
| Blue  | yes        | yes     | yes        |
| Red   | yes        | yes     | yes        |
| White | yes        | yes     | yes        |
| Blue  | yes        | yes     | yes        |
| Red   | yes        | yes     | yes        |
| White | yes        | yes     | yes        |
| Blue  | yes        | no      | yes        |
| Red   | yes        | no      | yes        |
| White | yes        | no      | yes        |
| Blue  | yes        | no      | no         |
| Red   | yes        | no      | no         |
| White | yes        | no      | no         |
+-------+------------+---------+------------+
```

Without the correct table join you are effectively retrieving a cartesian product of both tables.

OUTER JOIN

As you probably noticed with the Venn diagram in Figure 1-2, when looking at a cartesian product between two intersecting sets, you will see there are indeed three different possible sets of data. The first set is the intersection of both sets and retrieving these rows using the `INNER JOIN` syntax has been demonstrated with Listing 1-17. The other two sets are the exclusions, that is, the colors that are not in flags, and the countries that have colors that are not defined in the set of recorded colors. You can retrieve these rows using the `OUTER JOIN` syntax as shown in Listing 1-22.

LISTING 1-22: outer-join.sql

Available for download on Wrox.com

```
SELECT f.country, f.color
FROM   flags f
LEFT OUTER JOIN colors c USING (color)
WHERE  c.color IS NULL;

+---------+--------+
| country | color  |
+---------+--------+
| Sweden  | Yellow |
+---------+--------+

SELECT c.color, c.is_primary
FROM   colors c
LEFT OUTER JOIN  flags f USING (color)
WHERE f.country IS NULL;

+-------+------------+
| color | is_primary |
+-------+------------+
| Black | no         |
| Green | yes        |
+-------+------------+
```

As you have noticed in these queries, the syntax is not just `OUTER JOIN`, but it also includes the keyword `LEFT`. You should also note that `OUTER` is an optional keyword and it is generally a best practice to reduce the SQL syntax to just use `LEFT JOIN`.

An `OUTER JOIN` *is used for two primary reasons. The first is when a set of data values may be unknown yet you want to retrieve a full set of rows that match part of your criteria. The second reason is when a normalized database does not enforce referential integrity. In the preceding example, it's* logical *that colors may exist and are not a flag color. It is not logical that flag colors do not exist in the colors table. In this situation the use of an* `OUTER JOIN` *is identifying data that constitutes a lack of data integrity.*

RIGHT JOIN

If you were wondering if there was a companion `RIGHT OUTER JOIN` syntax, there is. It is possible to return the same results as shown in the preceding example using a `RIGHT JOIN` as shown in Listing 1-23.

LISTING 1-23: right-join.sql

Available for download on Wrox.com

```
SELECT c.color, c.is_primary
FROM   colors c
LEFT JOIN  flags f USING (color)
WHERE f.country IS NULL;
```

…can be written as

```
SELECT c.color, c.is_primary
FROM   flags f
RIGHT JOIN colors c USING (color)
WHERE f.country IS NULL;

+-------+------------+
| color | is_primary |
+-------+------------+
| Black | no         |
| Green | yes        |
+-------+------------+
```

LEFT JOIN

It is generally considered a good practice to write queries as `LEFT JOIN`, and to be consistent throughout all your SQL statements for your application.

In review of these two join examples you can conclude the following conditions:

- A join using `INNER JOIN` can be considered a mandatory condition, where a row in the left-side table must match a corresponding row in the right-side table.
- A join using `OUTER JOIN` can be considered an optional condition, where a row in the `LEFT` or `RIGHT` table as specified may or may not correspond to a row in the associated table.

Other JOIN Syntax

MySQL provides a number of other varieties of joins. For the `CROSS JOIN` in MySQL this is considered identical in operation to an `INNER JOIN`.

MySQL provides a `STRAIGHT_JOIN`, which is considered equivalent to the `JOIN` command. However, this acts more as a hint to the MySQL optimizer to determine processing tables in a given order. The `NATURAL [LEFT|RIGHT] JOIN` is similar to the corresponding `[INNER|LEFT|RIGHT] JOIN`; however, all matching column names between both tables are implied. In the previous `INNER JOIN` example with both the `ON` and `USING` syntax, you could have simply written:

LISTING 1-24: natural-join.sql

Available for download on Wrox.com

```
SELECT f.color, c.is_primary, c.is_dark, c.is_rainbow
FROM   flags f
NATURAL JOIN colors c
WHERE  f.country='USA';
```

The `NATURAL JOIN` can be dangerous because the columns are not specified, additional join columns may actually exist in your database design intentionally or unintentionally, and your table structures may change over time; the results of the query may in fact result in different rows returned. For more information on joins you can review the MySQL Reference Manual, which includes several different sections:

- Join Syntax `http://dev.mysql.com/doc/refman/5.1/en/join.html`
- Left Join and Right Join Optimization `http://dev.mysql.com/doc/refman/5.1/en/left-join-optimization.html`
- Outer Join Simplification `http://dev.mysql.com/doc/refman/5.1/en/outer-join-simplification.html`
- Join Types Index `http://dev.mysql.com/doc/refman/5.1/en/dynindex-jointype.html`

UPDATE and DELETE JOIN Syntax

Joins are not limited to `SELECT` statements in MySQL. You can use a join in MySQL `UPDATE` and `DELETE` statements as well. Listing 1-25 shows an example.

LISTING 1-25: update.sql

Available for download on Wrox.com

```
UPDATE flags INNER JOIN colors USING (color)
SET    flags.color = UPPER(color)
WHERE  colors.is_dark = 'yes';

SELECT color
FROM   flags
WHERE  country = 'USA';

+-------+
| color |
+-------+
| BLUE  |
| Red   |
| White |
+-------+
```

Case Sensitivity

MySQL by default performs case-insensitive comparison for string columns in a table join `ON`, `USING`, or `WHERE` comparison. This differs from other popular relational databases. In this case `'USA'` is equal to `'usa'`, for example. It is possible via either defining your table column with a case-sensitive collation or using a specific prequalifier to implement case-sensitive comparison. Listing 1-26 shows an example.

LISTING 1-26: case-sensitivity.sql

Available for download on Wrox.com

```
SELECT 'USA' = 'USA', 'USA' = 'Usa', 'USA' = 'usa',
       'USA' = 'usa' COLLATE latin1_general_cs AS different;
```

```
+---------------+---------------+---------------+-----------+
| 'USA' = 'USA' | 'USA' = 'Usa' | 'USA' = 'usa' | different |
+---------------+---------------+---------------+-----------+
|             1 |             1 |             1 |         0 |
+---------------+---------------+---------------+-----------+
```

Complex Joins

Although the basics of joins in MySQL have been described, to become a real expert is to understand the possibilities of joins. It is possible to write rather obfuscated SQL statements including subqueries and derived tables. However, the disadvantage is the lack of readability and maintainability of your SQL. Listing 1-27 is a simple multi-table join that combines joining to the same table multiple times, and combines `INNER JOIN` and `LEFT JOIN` syntax to return the population, state, and capital of all countries that have at least Red, White, and Blue in the flag:

LISTING 1-27: complex-join.sql

Available for download on Wrox.com

```
SELECT f1.country, c.population,
       IFNULL(ci.city,'Not Recorded') AS city, s.abbr, s.state
FROM   flags f1
INNER JOIN flags f2 ON f1.country = f2.country
INNER JOIN flags f3 ON f1.country = f3.country
INNER JOIN countries c ON f1.country = c.country
LEFT JOIN cities ci ON f1.country = ci.country AND ci.is_country_capital = 'yes'
LEFT JOIN states s  ON f1.country = s.country AND ci.state = s.state
WHERE f1.color = 'Red'
AND   f2.color = 'White'
AND   f3.color = 'Blue';

+-----------+------------+---------------+------+-------+
| country   | population | city          | abbr | state |
+-----------+------------+---------------+------+-------+
| Australia |   21888000 | Not Recorded  | NULL | NULL  |
| USA       |  307222000 | Washington DC | NULL | NULL  |
+-----------+------------+---------------+------+-------+
```

In this example, if you were to replace the `LEFT JOIN` with an `INNER JOIN`, the results of the data would change accordingly based on the recorded data.

When it is possible to write complex joins, in MySQL the combination of joins, subqueries, and derived tables can result in SQL statements that do not perform optimally. There must always be a balance between returning a result set in a single query and performance of the statement. Although writing multiple statements in MySQL, combined with the use of temporary tables, may introduce more SQL code, this may be more optimal for the speed of your web site.

MYSQL UNIONS

A UNION statement is used to combine the results of more than one SELECT statement into the results for one SQL query. For a valid UNION statement, all SELECT statements must have the same number of columns, and these columns must be of the same data type for each column in the SELECT statement. MySQL supports the UNION and UNION ALL constructs for joining SELECT results.

When learning to use UNION, you can first consider writing individual SELECT statements. All individual SELECT statements within a UNION statement are valid SELECT statements, except for the ORDER BY clause, which can be defined only once in a UNION statement and is used to order the results of all combined queries. Listing 1-28 shows an example.

Available for download on Wrox.com

LISTING 1-28: union.sql

```
SELECT f.country
FROM    flags f
INNER JOIN colors c USING (color)
WHERE c.is_dark = 'yes'
UNION
SELECT f.country
FROM    flags f
INNER JOIN colors c USING (color)
WHERE c.is_primary = 'yes';

+-----------+
| country   |
+-----------+
| Australia |
| Sweden    |
| USA       |
| Canada    |
| Japan     |
+-----------+
```

The UNION also supports the additional keywords ALL or DISTINCT. By default, the UNION syntax returns a unique set of rows for all SELECT sets, removing any duplicates. The ALL syntax, however, returns all rows from each SELECT statement combined, and the DISTINCT syntax returns all DISTINCT rows for each SELECT. Listing 1-29 shows an example of this.

Available for download on Wrox.com

LISTING 1-29: union-all.sql

```
SELECT f.country, 'Dark'
FROM    flags f
INNER JOIN colors c USING (color)
WHERE c.is_dark = 'yes'
UNION ALL
SELECT f.country, 'Primary'
FROM    flags f
INNER JOIN colors c USING (color)
```

```
WHERE c.is_primary = 'yes';

+-----------+---------+
| country   | Dark    |
+-----------+---------+
| Australia | Dark    |
| Sweden    | Dark    |
| USA       | Dark    |
| Australia | Primary |
| Sweden    | Primary |
| USA       | Primary |
| Australia | Primary |
| Canada    | Primary |
| Japan     | Primary |
| USA       | Primary |
| Australia | Primary |
| Canada    | Primary |
| Japan     | Primary |
| USA       | Primary |
+-----------+---------+
```

In Listing 1-30 you will see a different set of results using the `DISTINCT` keyword.

LISTING 1-30: union-distinct.sql

Available for download on Wrox.com

```
SELECT f.country, 'Dark'
FROM    flags f
INNER JOIN colors c USING (color)
WHERE c.is_dark = 'yes'
UNION DISTINCT
SELECT f.country, 'Primary'
FROM    flags f
INNER JOIN colors c USING (color)
WHERE c.is_primary = 'yes';

+-----------+---------+
| country   | Dark    |
+-----------+---------+
| Australia | Dark    |
| Sweden    | Dark    |
| USA       | Dark    |
| Australia | Primary |
| Sweden    | Primary |
| USA       | Primary |
| Canada    | Primary |
| Japan     | Primary |
+-----------+---------+
```

MySQL does not support the `INTERSECT` *or* `MINUS` *syntax that are additional* `UNION` *related constructs that can be found in other relational database products. Refer to* `http://dev.mysql.com/doc/refman/5.1/en/union.html` *for more information.*

GROUP BY IN MYSQL QUERIES

The `GROUP BY` syntax allows for the aggregation of rows selected and the use of scalar functions. In MySQL, it is possible to use scalar functions without a GROUP BY and produce what can be considered inconsistent results. Listing 1-31 shows an example.

Available for download on Wrox.com

LISTING 1-31: count-no-group.sql

```
SELECT country, COUNT(*)
FROM   flags;
+-----------+----------+
| country   | COUNT(*) |
+-----------+----------+
| Australia |       12 |
+-----------+----------+
```

MySQL provides the expected ANSI SQL syntax requiring a `GROUP BY` statement to contain all non-scalar function columns with the use of `sql_mode`. Listing 1-32 shows an example of this.

Available for download on Wrox.com

LISTING 1-32: count-group.sql

```
SET SESSION sql_mode=ONLY_FULL_GROUP_BY;

SELECT country, COUNT(*)
FROM   flags;
ERROR 1140 (42000): Mixing of GROUP columns (MIN(),MAX(),COUNT(),...)
   with no GROUP columns is illegal if there is no GROUP BY clause

SELECT country, COUNT(*) AS color_count
FROM   flags
GROUP  BY country;
+-----------+-------------+
| country   | color_count |
+-----------+-------------+
| Australia |           3 |
| Canada    |           2 |
| Japan     |           2 |
| Sweden    |           2 |
| USA       |           3 |
+-----------+-------------+
```

One scalar function exists that does not return a numeric value; this is the `GROUP_CONCAT()` function shown in Listing 1-33.

Available for download on Wrox.com

LISTING 1-33: count-group-concat.sql

```
SELECT country, GROUP_CONCAT(color) AS colors
FROM   flags
GROUP BY country;
```

```
+-----------+----------------+
| country   | colors         |
+-----------+----------------+
| Australia | Blue,Red,White |
| Canada    | Red,White      |
| Japan     | Red,White      |
| Sweden    | Blue,Yellow    |
| USA       | Blue,Red,White |
+-----------+----------------+

SELECT country, GROUP_CONCAT(color) AS colors, COUNT(*) AS color_count
FROM   flags
GROUP BY country;
+-----------+----------------+-------------+
| country   | colors         | color_count |
+-----------+----------------+-------------+
| Australia | Blue,Red,White |           3 |
| Canada    | Red,White      |           2 |
| Japan     | Red,White      |           2 |
| Sweden    | Blue,Yellow    |           2 |
| USA       | Blue,Red,White |           3 |
+-----------+----------------+-------------+
5 rows in set (0.00 sec)
```

WITH ROLLUP

A feature of the `GROUP BY` syntax is the additional keywords `WITH ROLLUP`. With this syntax, the rows returned include aggregated rows for each `GROUP BY` column. This is represented by `NULL`. The output in Listing 1-34 shows a single-column and two-column example:

LISTING 1-34: count-with-rollup.sql

Available for download on Wrox.com

```
SELECT country, COUNT(*) AS color_count
FROM   flags
GROUP  BY country WITH ROLLUP;
+-----------+-------------+
| country   | color_count |
+-----------+-------------+
| Australia |           3 |
| Canada    |           2 |
| Japan     |           2 |
| Sweden    |           2 |
| USA       |           3 |
| NULL      |          12 |
+-----------+-------------+

SELECT c.color, c.is_dark, COUNT(*)
FROM    colors c, flags f
WHERE c.color = f.color
GROUP BY c.color, c.is_dark WITH ROLLUP;
+-------+---------+----------+
| color | is_dark | COUNT(*) |
+-------+---------+----------+
| Blue  | yes     |        3 |
| Blue  | NULL    |        3 |
| Red   | no      |        4 |
| Red   | NULL    |        4 |
| White | no      |        4 |
| White | NULL    |        4 |
| NULL  | NULL    |       11 |
+-------+---------+----------+
```

HAVING

To restrict the list of aggregated rows returned when using `GROUP BY` for any scalar functions, you use the `HAVING` clause to define the condition and not the `WHERE` clause. Listing 1-35 shows an example.

LISTING 1-35: having.sql

Available for download on Wrox.com

```
SELECT country, GROUP_CONCAT(color) AS colors
FROM   flags
GROUP BY country
HAVING COUNT(*) = 2;

+---------+-------------+
| country | colors      |
+---------+-------------+
| Canada  | Red,White   |
| Japan   | Red,White   |
| Sweden  | Blue,Yellow |
+---------+-------------+
```

You can use scalar functions that are not defined in the `SELECT` clause as shown in the preceding example. Unlike `ORDER BY` you must specify the name of the column; the numeric column order is not a permitted syntax.

LOGICAL OPERATIONS AND FLOW CONTROL IN MYSQL

MySQL has three states for any logic, `TRUE`, `FALSE`, or `NULL`:

```
mysql> SELECT TRUE,FALSE,NULL;
+------+-------+------+
| TRUE | FALSE | NULL |
+------+-------+------+
|    1 |     0 | NULL |
+------+-------+------+
```

Comparison operations such as =, <>, `IS`, `IS NOT`, `IN`, `ISNULL`, and so on will result in one of these three states:

```
mysql> SELECT 'A' IS NOT NULL, 'A' IS NULL, NULL = NULL, NULL IS NULL;
+-----------------+-------------+-------------+--------------+
| 'A' IS NOT NULL | 'A' IS NULL | NULL = NULL | NULL IS NULL |
+-----------------+-------------+-------------+--------------+
|               1 |           0 |        NULL |            1 |
+-----------------+-------------+-------------+--------------+
```

MySQL always returns 1 for a `TRUE` state, and any non-zero value evaluates to `TRUE`.

```
mysql> SELECT 5 IS TRUE, 0 IS FALSE, 10 IS NOT NULL;
+-----------+------------+----------------+
| 5 IS TRUE | 0 IS FALSE | 10 IS NOT NULL |
+-----------+------------+----------------+
|         1 |          1 |              1 |
+-----------+------------+----------------+
```

Logic Operators

MySQL has four logic control operators: `AND`, `OR`, `NOT`, and `XOR`. Three of these operators also have shorthand notations: `&& (AND)`, `|| (OR)`, `! (NOT)`. These shorthand notations should not be confused with the Bit operators, which are single characters of `&` and `|`.

```
mysql> SELECT TRUE AND TRUE, TRUE AND FALSE, TRUE AND NULL, NULL AND NULL;
+---------------+----------------+---------------+---------------+
| TRUE AND TRUE | TRUE AND FALSE | TRUE AND NULL | NULL AND NULL |
+---------------+----------------+---------------+---------------+
|             1 |              0 |          NULL |          NULL |
+---------------+----------------+---------------+---------------+

mysql> SELECT TRUE OR TRUE, TRUE OR FALSE, TRUE OR NULL, NULL OR NULL;
+--------------+---------------+--------------+--------------+
| TRUE OR TRUE | TRUE OR FALSE | TRUE OR NULL | NULL OR NULL |
+--------------+---------------+--------------+--------------+
|            1 |             1 |            1 |         NULL |
+--------------+---------------+--------------+--------------+

mysql> SELECT TRUE XOR TRUE, TRUE XOR FALSE, TRUE XOR NULL, NULL XOR NULL;
+---------------+----------------+---------------+---------------+
| TRUE XOR TRUE | TRUE XOR FALSE | TRUE XOR NULL | NULL XOR NULL |
+---------------+----------------+---------------+---------------+
|             0 |              1 |          NULL |          NULL |
+---------------+----------------+---------------+---------------+
```

Unlike `AND`, `OR`, and `XOR`, the `NOT` operator does not evaluate two values; it simply returns the inverse of the provided value:

```
mysql> SELECT NOT TRUE, NOT FALSE, NOT NULL;
+----------+-----------+----------+
| NOT TRUE | NOT FALSE | NOT NULL |
+----------+-----------+----------+
|        0 |         1 |     NULL |
+----------+-----------+----------+
```

You can change the `||` shorthand operator of MySQL using `sql_mode=PIPES_AS_CONCAT`, which will give unexpected results as shown in Listing 1-36.

Available for download on Wrox.com

LISTING 1-36: logic-operators.sql

```
mysql> SELECT TRUE OR FALSE, TRUE || FALSE;
+---------------+---------------+
| TRUE OR FALSE | TRUE || FALSE |
+---------------+---------------+
|             1 |             1 |
+---------------+---------------+

mysql> SET SESSION sql_mode=PIPES_AS_CONCAT;
mysql> SELECT TRUE OR FALSE, TRUE || FALSE;
+---------------+---------------+
| TRUE OR FALSE | TRUE || FALSE |
+---------------+---------------+
|             1 | 10            |
+---------------+---------------+
```

Flow Control

MySQL provides four functions for control flow: `IF()`, `CASE`, `IFNULL()`, and `NULLIF()`. The `IF()` function provides the syntax of a ternary operator with two possible outcomes for a given condition:

```
mysql> SELECT IF (2 > 1,'2 is greater than 1','2 is not greater than 1') AS answer;
+----------------------+
| answer               |
+----------------------+
| 2 is greater than 1 |
+----------------------+
```

The MySQL `CASE` statement operates in similar fashion to the PHP switch syntax where a single given condition of multiple options results in a true assignment. The `CASE` statement also includes a special default case when no conditions equate to a `TRUE` value.

```
mysql> SET @value=CONVERT(RAND()* 10, UNSIGNED INTEGER);

mysql> SELECT @value,
    -> CASE
    ->   WHEN @value < 3 THEN 'Value is < 3'
    ->   WHEN @value > 6 THEN 'Value is > 6'
    ->   WHEN @value = 3 OR @value = 6 THEN 'Value is 3 or 6'
    ->   ELSE 'Value is 4 or 5'
    ->   END;

+--------+-----------------+
|      3 | Value is 3 or 6  |
+--------+-----------------+
```

Though it is possible to perform complex flow control functions via SQL, the MySQL database is designed for storing and retrieving data. Where possible, complex rules should be written in the application layer to enable greater performance.

The remaining two functions IFNULL() and NULLIF() support conditional expressions for handling NULL. IFNULL() returns NULL if the provided expression equates to NULL, or the value of the expression. NULLIF() returns a NULL result if the two expressions result in a TRUE condition. Listing 1-37 shows an example of this.

LISTING 1-37: flow-control.sql

Available for download on Wrox.com

```
mysql> SELECT IFNULL(NULL,'Value is NULL') AS result1,
              IFNULL(1 > 2, 'NULL result') AS result2;
+---------------+---------+
| result1       | result2 |
+---------------+---------+
| Value is NULL | 0       |
+---------------+---------+

mysql> SELECT NULLIF(TRUE,TRUE) AS istrue,
              NULLIF(TRUE,FALSE) AS isfalse,
              NULLIF(TRUE,NULL) AS isnull;
+--------+---------+--------+
| istrue | isfalse | isnull |
+--------+---------+--------+
|   NULL |       1 |      1 |
+--------+---------+--------+
```

MAINTAINING RELATIONAL INTEGRITY

Although many developers consider relational integrity as using foreign keys to maintain referential integrity of your data, i.e. the Consistency part of the ACID properties, relational integrity in MySQL is available via a variety of means and at various different levels. These can be specified at the table structure level syntax of CREATE TABLE, ALTER TABLE or at the MySQL SESSION or GLOBAL VARIABLES level. MySQL can also provide a level of integrity that is storage engine specific.

Constraints

A constraint restricts the type of value that is stored in a given table column. There are various options for single column values including NOT NULL, UNSIGNED, ENUM, and SET. A UNIQUE KEY constraint applies to one or more columns of a single table. A FOREIGN KEY constraint involves a mandatory relationship between two tables.

NOT NULL

To ensure a column must contain a value, you can specify the NOT NULL constraint. It is important that the use of DEFAULT is not specified to enforce NOT NULL constraints. The DEFAULT attribute, as the name suggests, provides a default value when one is not specified. With a column definition of col1 CHAR(5) NOT NULL DEFAULT '', when col1 is not specified in an INSERT statement, an error is not returned for not specifying a mandatory column. Instead a blank value '' — not to be confused with a NULL value — is inserted into the column. This is even more confusing when the column is defined as nullable. In this instance, you have NULL and '' as possible values. These are not equal and this leads to confusion in your application. Should they be equal? When searching `col1 LIKE NULL` you would also need to include `OR col1 = ''`.

UNSIGNED

When an integer column only requires a non-negative number, the specification of the `UNSIGNED` constraint will ensure the column can only contain 0 or a positive value. For example:

LISTING 1-38: unsigned.sql

Available for download on Wrox.com

```
mysql> DROP TABLE IF EXISTS example;
mysql> CREATE TABLE example (
    ->   int_signed      INT NOT NULL,
    ->   int_unsigned    INT UNSIGNED NOT NULL
    -> ) ENGINE=InnoDB DEFAULT CHARSET latin1;

mysql> INSERT INTO example (int_signed, int_unsigned) VALUES ( 1, 1);
mysql> INSERT INTO example (int_signed, int_unsigned) VALUES ( 0, 0);
mysql> INSERT INTO example (int_signed, int_unsigned) VALUES ( -1, 1);
mysql> INSERT INTO example (int_signed, int_unsigned) VALUES ( 1, -1);

ERROR 1264 (22003): Out of range value for column 'int_unsigned' at row 1

mysql> SELECT * FROM example;

+------------+--------------+
| int_signed | int_unsigned |
+------------+--------------+
|          1 |            1 |
|          0 |            0 |
|         -1 |            1 |
+------------+--------------+
```

ENUM and SET

The `ENUM` data column and supporting `SET` column data types enable you to enforce integrity by enabling only a specific set of possible values. This can be of benefit when only a set range of values are possible for a column. Listing 1-39 shows an example.

LISTING 1-39: enum.sql

Available for download on Wrox.com

```
mysql> CREATE TABLE example (
    ->  currency  ENUM('USD','CAD','AUD') NOT NULL
    -> ) ENGINE=InnoDB DEFAULT CHARSET latin1;

mysql> INSERT INTO example (currency) VALUES ('AUD');
mysql> INSERT INTO example (currency) VALUES ('EUR');
```

```
ERROR 1265 (01000): Data truncated for column 'currency' at row 1

mysql> SELECT * FROM example;
+----------+
| currency |
+----------+
| AUD      |
+----------+
```

The SET data type operates similarly to ENUM except that one or more of the defined values are permitted as a valid value. The disadvantage of using ENUM or SET is that a DDL statement is required to change the range of possible values.

UNIQUE KEY

The UNIQUE KEY constraint ensures that all values in a given column are actually unique. A UNIQUE KEY constraint may also involve more than one column. It is possible for a UNIQUE KEY constraint to contain a nullable column, because NULL is considered a unique value. Listing 1-40 shows an example.

LISTING 1-40: unique-key.sql

Available for download on Wrox.com

```
mysql> CREATE TABLE example (
    ->  int_unique           INT UNSIGNED NOT NULL,
    ->  int_nullable_unique  INT UNSIGNED NULL,
    ->  UNIQUE KEY (int_unique),
    ->  UNIQUE KEY(int_nullable_unique)
    -> ) ENGINE=InnoDB DEFAULT CHARSET latin1;

mysql> INSERT INTO example (int_unique, int_nullable_unique) VALUES (1, 1);
mysql> INSERT INTO example (int_unique, int_nullable_unique) VALUES (2, NULL);
mysql> INSERT INTO example (int_unique, int_nullable_unique) VALUES (3, NULL);
mysql> INSERT INTO example (int_unique, int_nullable_unique) VALUES (1, NULL);
ERROR 1062 (23000): Duplicate entry '1' for key 'int_unique'

mysql> INSERT INTO example (int_unique, int_nullable_unique) VALUES (4, 1);
ERROR 1062 (23000): Duplicate entry '1' for key 'int_nullable_unique'

mysql> SELECT * FROM example;
+------------+---------------------+
| int_unique | int_nullable_unique |
+------------+---------------------+
|          2 |                NULL |
|          3 |                NULL |
|          1 |                   1 |
+------------+---------------------+
```

FOREIGN KEY

Developers will generally consider foreign keys as the basis of relational integrity; however, as shown in this chapter, other important factors exist for maintaining integrity. Foreign keys can ensure the Consistency portion of ACID compliance. Though it is possible to use manual procedures to maintain data integrity, this is a less than an ideal approach.

In the current production MySQL 5.1, foreign keys are supported only with the InnoDB storage engine. Some additional third-party storage engines do support foreign keys. Refer to Chapter 3 for additional information.

The MySQL Reference Manual defines the syntax for a `FOREIGN KEY` that can be used in `CREATE TABLE` or `ALTER TABLE` as:

```
[CONSTRAINT [symbol]] FOREIGN KEY
    [index_name] (index_col_name, ...)
    REFERENCES tbl_name (index_col_name,...)
    [ON DELETE reference_option]
    [ON UPDATE reference_option]

reference_option:
    RESTRICT | CASCADE | SET NULL | NO ACTION
```

As you've seen from earlier join examples, the countries table contains an invalid color. If you had defined the tables using foreign keys, you would not have experienced this data integrity problem. This provides a code example of what is necessary to correct bad data. First, you should attempt to create the missing foreign key integrity constraint as shown in Listing 1-41.

LISTING 1-41: foreign-key-alter.sql

Available for download on Wrox.com

```
mysql> ALTER TABLE flags
    -> ADD FOREIGN KEY (color)
    -> REFERENCES colors (color)
    -> ON DELETE CASCADE;
ERROR 1452 (23000): Cannot add or update a child row:a foreign key constraint fails
 ('chapter1'.'#sql-86f_1928bd', CONSTRAINT '#sql-86f_1928bd_ibfk_1' FOREIGN KEY
 ('color') REFERENCES 'colors' ('color') ON DELETE CASCADE)
```

Due to the error, you now need to identify the problem data in either the parent or child table. You could identify with a subquery or, as shown previously, an outer join to retrieve the invalid data. We know because of the small sample data that the color Yellow is the cause of the failure. Do you:

- Delete the offending row that contains the invalid data? This would then in turn produce invalid consistent data for the Swedish flag.
- Delete all flag data for Sweden? This would delete potentially valid data that you may use or that may be valuable elsewhere.
- Add the missing data to the colors base table?

These are important design decisions that affect how your application will run. In Listing 1-42, we make the decision to use the last option and add the missing color Yellow to the colors table to successfully add the foreign key.

Available for download on Wrox.com

LISTING 1-42: foreign-key-yellow.sql

```
mysql> INSERT INTO colors (color,is_primary,is_dark,is_rainbow)
                   VALUES ('Yellow','no','no','yes');

mysql> ALTER TABLE flags
       ADD FOREIGN KEY (color)
       REFERENCES colors (color)
       ON DELETE CASCADE;

mysql> SELECT *
       FROM colors
       WHERE color='Yellow';
+--------+------------+---------+------------+
| color  | is_primary | is_dark | is_rainbow |
+--------+------------+---------+------------+
| Yellow | no         | no      | yes        |
+--------+------------+---------+------------+

mysql> SELECT *
       FROM flags
       WHERE country IN (SELECT country
                         FROM flags
                         WHERE color='Yellow');
+---------+--------+
| country | color  |
+---------+--------+
| Sweden  | Blue   |
| Sweden  | Yellow |
+---------+--------+
```

You have now defined a `FOREIGN KEY` between the colors table and the flags table where the color for the flag must exist in the colors tables. You have also defined this rule to have a cascade `DELETE` rule, which states that if you delete a color, you will also delete all rows that use this color. Listing 1-43 shows an example:

Available for download on Wrox.com

LISTING 1-43: foreign-key-delete.sql

```
mysql> DELETE FROM colors WHERE color='Yellow';
mysql> SELECT * FROM flags WHERE color='Yellow';
Empty set (0.00 sec)

mysql> SELECT *
       FROM flags
       WHERE country IN (SELECT country
                         FROM flags
                         WHERE color='Yellow');

mysql> SELECT *
```

```
          FROM flags
          WHERE country = 'Sweden';
+---------+-------+
| country | color |
+---------+-------+
| Sweden  | Blue  |
+---------+-------+
```

Although the `FOREIGN KEY` constraint has ensured data integrity at the row level, it has not performed the type of integrity you would expect. The use of the `FOREIGN KEY` constraint will not ensure the level of application integrity you ideally wish to have.

Defining your foreign key definitions is a very important architectural design decision that should be performed before you add any data. It is far easier to remove a constraint later than to add it later.

A further benefit of InnoDB foreign key constraints is the requirement that both columns in the from table and the to table must use an identical data type. This improves the data integrity of the database.

You can find additional information on foreign keys in InnoDB at `http://dev.mysql.com/doc/refman/5.0/en/innodb-foreign-key-constraints.html`.

It is possible for foreign key constraints to be disabled within MySQL with the `SET foreign_key_checks = 0|1` option. This can further confuse the integrity of your database because permission to manipulate data via a DML statement can be overridden via the `SET` command at both the `SESSION` or `GLOBAL` level.

When using cascading foreign key constraints and the `REPLACE` command, your database may exhibit unexpected behavior or performance. The `REPLACE` command is generally understood and described as an `UPDATE` for the matching row. If no row is found then the `INSERT` command inserts the row. In implementation, however, `REPLACE` is actually a `DELETE` of the existing row, and then an `INSERT` of the new row. Be aware of this execution path of `REPLACE` when adding constraints that use cascading syntax.

Using Server SQL Modes

Introduced first in 4.1 and enhanced in 5.0, the Server SQL mode provides various features including different types of relational integrity. MySQL, by default, is very lax with data integrity and this can have unexpected results. For example, look at Listing 1-44.

Available for download on Wrox.com

LISTING 1-44: no-sql-mode.sql

```
mysql> CREATE TABLE example (
    ->  i TINYINT UNSIGNED NOT NULL,
    ->  c CHAR(2) NULL
    -> ) ENGINE=InnoDB DEFAULT CHARSET latin1;

mysql> INSERT INTO example (i) VALUES (0), (-1),(255), (9000);
Query OK, 4 rows affected, 2 warnings (0.00 sec)

mysql> SHOW WARNINGS;
+---------+------+---------------------------------------------+
| Level   | Code | Message                                     |
+---------+------+---------------------------------------------+
| Warning | 1264 | Out of range value for column 'i' at row 2 |
| Warning | 1264 | Out of range value for column 'i' at row 4 |
+---------+------+---------------------------------------------+
2 rows in set (0.00 sec)

mysql> INSERT INTO example (c) VALUES ('A'),('BB'),('CCC');
Query OK, 3 rows affected, 2 warnings (0.00 sec)

mysql> SHOW WARNINGS;
+---------+------+-----------------------------------------+
| Level   | Code | Message                                 |
+---------+------+-----------------------------------------+
| Warning | 1364 | Field 'i' doesn't have a default value |
| Warning | 1265 | Data truncated for column 'c' at row 3 |
+---------+------+-----------------------------------------+
2 rows in set (0.00 sec)

mysql> SELECT * FROM example;
+-----+------+
| i   | c    |
+-----+------+
|   0 | NULL |
|   0 | NULL |
| 255 | NULL |
| 255 | NULL |
|   0 | A    |
|   0 | BB   |
|   0 | CC   |
+-----+------+
7 rows in set (0.00 sec)
```

In these preceding SQL statements you find numerous actual errors in the data, yet no errors actually occurred.

MySQL issues only warnings, and most application developers actually ignore these warnings, never executing a `SHOW WARNINGS` to identify these silent data truncations. You expected to insert a value of 9,000; however, only 255 was stored. You expected to insert a string of three characters, yet only two characters were recorded. You didn't specify a value for a `NOT NULL` column, yet a default value was recorded.

The solution is to use a strict SQL mode available since MySQL 5.0. MySQL provides two strict types: `STRICT_ALL_TABLES` and `STRICT_TRANS_TABLES`. For the purposes of ensuring data integrity for all tables, this section only discusses `STRICT_ALL_TABLES`. When you re-run the previous SQL statements, you see the code in Listing 1-45:

LISTING 1-45: sql-mode-traditional.sql

Available for download on Wrox.com

```
mysql> TRUNCATE TABLE example;
Query OK, 0 rows affected (0.00 sec)

mysql> SET SESSION sql_mode='TRADITIONAL';
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO example (i) VALUES (0), (-1),(255), (9000);
ERROR 1264 (22003): Out of range value for column 'i' at row 2
mysql> INSERT INTO example (c) VALUES ('A'),('BB'),('CCC');
ERROR 1364 (HY000): Field 'i' doesn't have a default value
mysql> SELECT * FROM example;
Empty set (0.00 sec)

mysql> INSERT INTO example (i) VALUES (0);
mysql> INSERT INTO example (i) VALUES (-1);
ERROR 1264 (22003): Out of range value for column 'i' at row 1
mysql> INSERT INTO example (i) VALUES (255);
mysql> INSERT INTO example (i) VALUES (9000);
ERROR 1264 (22003): Out of range value for column 'i' at row 1
mysql> INSERT INTO example (c) VALUES ('A');
ERROR 1364 (HY000): Field 'i' doesn't have a default value
mysql> INSERT INTO example (i,c) VALUES (1,'A');
mysql> INSERT INTO example (i,c) VALUES (1,'BB');
mysql> INSERT INTO example (i,c) VALUES (1,'CCC');
ERROR 1406 (22001): Data too long for column 'c' at row 1
mysql> SELECT * FROM example;
+-----+------+
| i   | c    |
+-----+------+
|   0 | NULL |
| 255 | NULL |
|   1 | A    |
|   1 | BB   |
+-----+------+
4 rows in set (0.00 sec)
```

You will notice now the expected errors of a more traditional relational database system. You will also notice that the multiple `INSERT VALUES` statements fail unconditionally. It is possible to alter this behavior by using a nontransactional storage engine such as MyISAM and further confuse the possible lack of data integrity. Listing 1-46 shows an example.

LISTING 1-46: sql-mode-traditional-myisam.sql

Available for download on Wrox.com

```
mysql> ALTER TABLE example ENGINE=MyISAM;
mysql> TRUNCATE TABLE example;
mysql> SET SESSION sql_mode='TRADITIONAL';
mysql> INSERT INTO example (i) VALUES (0), (-1),(255), (9000);
ERROR 1264 (22003): Out of range value for column 'i' at row 2
mysql> INSERT INTO example (i,c) VALUES (1,'A'),(1,'BB'),(1,'CCC');
ERROR 1406 (22001): Data too long for column 'c' at row 3
mysql> SELECT * FROM example;
+---+------+
| i | c    |
+---+------+
| 0 | NULL |
| 1 | A    |
| 1 | BB   |
+---+------+
3 rows in set (0.00 sec)
```

sql_mode=TRADITIONAL

The use of `sql_mode` is essential in application development to providing an acceptable level of data integrity. Systems should ideally be defined with a minimum of `sql_mode=TRADITIONAL`. The MySQL Reference Manual provides the following description for `TRADITIONAL`.

"Make MySQL behave like a 'traditional' SQL database system. A simple description of this mode is 'give an error instead of a warning' when inserting an incorrect value into a column.

Equivalent to STRICT_TRANS_TABLES, STRICT_ALL_TABLES, NO_ZERO_IN_DATE, NO_ZERO_DATE, ERROR_FOR_DIVISION_BY_ZERO, NO_AUTO_CREATE_USER."

`TRADITIONAL` provides additional modes including important data integrity for date values.

It is important that changing the `sql_mode` for an application requires appropriate testing. It is dangerous to change `sql_mode` on a production system because functionality that may have operated previously may now operate differently.

sql_mode=NO_ENGINE_SUBSTITUTION

When using relational integrity that is engine specific, such as the InnoDB `FOREIGN KEY` constraint, it is important that a table is created with the intended storage engine as specified with the `CREATE TABLE` statement. Unfortunately, MySQL does not enforce this by default. Listing 1-47 shows an example.

LISTING 1-47: sql-mode-engine-myisam.sql

Available for download on Wrox.com

```
mysql> CREATE TABLE example (
    ->   col1 INT UNSIGNED NOT NULL,
    ->   col2 INT UNSIGNED NOT NULL
    -> ) ENGINE=InnoDB DEFAULT CHARSET latin1;
Query OK, 0 rows affected, 2 warnings (0.01 sec)

mysql> SHOW WARNINGS;
+---------+------+------------------------------------------------+
| Level   | Code | Message                                        |
+---------+------+------------------------------------------------+
| Warning | 1286 | Unknown table engine 'InnoDB'                  |
| Warning | 1266 | Using storage engine MyISAM for table 'example' |
+---------+------+------------------------------------------------+
2 rows in set (0.00 sec)

mysql> SHOW CREATE TABLE example\G
*************************** 1. row ***************************
       Table: example
Create Table: CREATE TABLE `example` (
  `col1` int(10) unsigned NOT NULL,
  `col2` int(10) unsigned NOT NULL
) ENGINE=MyISAM DEFAULT CHARSET=latin1
1 row in set (0.00 sec)
```

The table has been successfully created, yet the created storage of MyISAM is not the specified storage engine of InnoDB.

To ensure this does not occur, you need to use the `sql_mode` in Listing 1-48.

Available for download on Wrox.com

LISTING 1-48: sql-mode-engine-error.sql

```
mysql> SET SESSION sql_mode='NO_ENGINE_SUBSTITUTION';

mysql> CREATE TABLE example (
    ->   col1 INT UNSIGNED NOT NULL,
    ->   col2 INT UNSIGNED NOT NULL
    -> ) ENGINE=InnoDB DEFAULT CHARSET latin1;
ERROR 1286 (42000): Unknown table engine 'InnoDB'
```

Storage Engine Integrity

The `ARCHIVE` storage engine provides a unique feature that can be considered a level of integrity. In Listing 1-49, `DELETE` and `UPDATE` are not supported and they return an error:

Available for download on Wrox.com

LISTING 1-49: archive-engine.sql

```
mysql> CREATE TABLE example (
    ->   pk INT UNSIGNED NOT NULL AUTO_INCREMENT,
    ->   col2 VARCHAR(10) NOT NULL,
    ->   PRIMARY KEY(pk)
    -> ) ENGINE=ARCHIVE DEFAULT CHARSET latin1;

mysql> INSERT INTO example (col2) VALUES ('a'),('b'),('c');

mysql> UPDATE example SET col2='x' WHERE pk=1;
ERROR 1031 (HY000): Table storage engine for 'example' doesn't have this option

mysql> DELETE FROM example  WHERE pk=1;
ERROR 1031 (HY000): Table storage engine for 'example' doesn't have this option
```

What MySQL Does Not Tell You

You should also be aware that MySQL may perform silent column changes when you create a table in MySQL. Though subtle, it is important that you know about these changes because they may reflect an impact on relational integrity. The following is a summary of several important points; however, you should always refer to the MySQL manual for a complete list of version specific changes: `http://dev.mysql.com/doc/refman/5.1/en/silent-column-changes.html`.

- `VARCHAR` columns specified less than four characters are silently converted to `CHAR`.
- All `TIMESTAMP` columns are converted to `NOT NULL`.
- String columns defined with a binary `CHARACTER SET` are converted to the corresponding binary data type; for example, `VARCHAR` is converted to `VARBINARY`.

What's Missing?

MySQL does not support any check constraints on columns, for example the popular Oracle syntax that can restrict the range of values that can be recorded in a column:

```
CONSTRAINT country_id  CHECK (country_id BETWEEN 100 and 999)
```

SUBQUERIES IN MYSQL

The *subquery* is a powerful means of retrieving additional data in a single MySQL `SELECT` statement. With subqueries, it is possible to introduce other sets of information for varying purposes. The following examples show three different and popular forms of subqueries.

Subquery

A true subquery, also known as *dependent query*, is a standalone `SELECT` statement that you can execute independently to produce a set of results that are then used with the parent query. In this form, the subquery is executed first, and the results are used for comparison with the parent query.

Available for download on Wrox.com

LISTING 1-50: subquery.sql

```
SELECT color
FROM colors
```

```
WHERE color IN
 (SELECT color
  FROM flags);

+-------+
| color |
+-------+
| Blue  |
| Red   |
| White |
+-------+
```

Correlated Subquery

A *correlated subquery* performs a join between the parent query and the subquery resulting in a dependency during the process of retrieving results. In this situation, both sets of data must be determined independently, then compared to return the matching results:

Available for download on Wrox.com

LISTING 1-51: correlated-sub-query.sql

```
SELECT DISTINCT f.color
FROM flags f
WHERE EXISTS
 (SELECT 1
  FROM colors c
  WHERE c.color = f.color);

+-------+
| color |
+-------+
| Blue  |
| Red   |
| White |
+-------+
```

Derived Table

Though `SELECT` statements shown in this chapter have used tables and columns, it is possible for any table or column within a `SELECT` statement to actually be the result of a `SELECT` statement. This is known as a *derived table*.

You can use a `SELECT` statement to create a derived table that acts in the position as a normal table. For example:

Available for download on Wrox.com

LISTING 1-52: derived-table.sql

```
SELECT r.color, r.countries, c.is_dark, c.is_primary
FROM colors c,
     (SELECT color, GROUP_CONCAT(country) AS countries
      FROM   flags
      GROUP BY color) r
      WHERE c.color = r.color;
+-------+---------------------------+---------+------------+
| color | countries                 | is_dark | is_primary |
+-------+---------------------------+---------+------------+
| Blue  | Australia,Sweden,USA      | yes     | yes        |
| Red   | Australia,Canada,Japan,USA | no     | yes        |
| White | Australia,Canada,Japan,USA | no     | yes        |
+-------+---------------------------+---------+------------+
```

An earlier example used a `GROUP BY` statement to return a concatenated list of colors per country. This can also be retrieved using a column-based derived table as shown in Listing 1-53.

Available for download on Wrox.com

LISTING 1-53: derived-column.sql

```
SELECT DISTINCT f.country,
       (SELECT GROUP_CONCAT(color)
        FROM flags f2
        WHERE f2.country = f.country) AS colors
FROM   flags f;

+-----------+----------------+
| country   | colors         |
+-----------+----------------+
| Australia | Blue,Red,White |
| Sweden    | Blue,Yellow    |
| USA       | Blue,Red,White |
| Canada    | Red,White      |
| Japan     | Red,White      |
+-----------+----------------+
```

You can find a great example of the complexity of SQL and derived tables in the Blog Post by Shlomi Noach at `http://code.openark.org/blog/mysql/sql-pie-chart`.

Complex Sub Queries

Listing 1-54 is a 66-line SQL statement that includes combined examples of `UNION`, `GROUP BY`, `IF()` and `CASE()` flow control, and multiple subqueries including table and column derived tables:

Available for download on Wrox.com

LISTING 1-54: complex-sql.sql

```
SELECT
  group_concat(
    IF(round(sqrt(pow(col_number/@stretch-0.5-(@size-1)/2, 2) +
       pow(row_number-(@size-1)/2, 2))) BETWEEN @radius*2/3 AND @radius,
    (SELECT SUBSTRING(@colors, name_order, 1) FROM
      (
      SELECT
        name_order,
        name_column,
        value_column,
        accumulating_value,
        accumulating_value/@accumulating_value AS accumulating_value_ratio,
        @aggregated_data := CONCAT(@aggregated_data, name_column, ': ',
           value_column, ' (', ROUND(100*value_column/@accumulating_value), '%)',
           '|') AS aggregated_name_column,
        2*PI()*accumulating_value/@accumulating_value AS accumulating_value_radians
      FROM (
        SELECT
          name_column,
          value_column,
          @name_order := @name_order+1 AS name_order,
          @accumulating_value := @accumulating_value+value_column
            AS accumulating_value
        FROM (
          <strong>SELECT name AS name_column, value AS value_column
             FROM sample_values2 LIMIT 4</strong>
          ) select_values,
          (SELECT @name_order := 0) select_name_order,
          (SELECT @accumulating_value := 0) select_accumulating_value,
          (SELECT @aggregated_data := '') select_aggregated_name_column
        ) select_accumulating_values
      ) select_for_radians
    WHERE accumulating_value_radians &gt;= radians LIMIT 1
    ), ' ')
    order by col_number separator '') as pie
FROM (
  SELECT
    t1.value AS col_number,
    t2.value AS row_number,
    @dx := (t1.value/@stretch - (@size-1)/2) AS dx,
    @dy := ((@size-1)/2 - t2.value) AS dy,
    @abs_radians := IF(@dx = 0, PI()/2, (atan(abs(@dy/@dx)))) AS abs_radians,
    CASE
      WHEN SIGN(@dy) &gt;= 0 AND SIGN(@dx) &gt;= 0 THEN @abs_radians
      WHEN SIGN(@dy) &gt;= 0 AND SIGN(@dx) &lt;= 0 THEN PI()-@abs_radians
      WHEN SIGN(@dy) &lt;= 0 AND SIGN(@dx) &lt;= 0 THEN PI()+@abs_radians
      WHEN SIGN(@dy) &lt;= 0 AND SIGN(@dx) &gt;= 0 THEN 2*PI()-@abs_radians
    END AS radians
  FROM
    tinyint_asc t1,
    tinyint_asc t2,
    (select @size := 23) sel_size,
    (select @radius := (@size/2 - 1)) sel_radius,
    (select @stretch := 4) sel_stretch,
    (select @colors := '#;o:X"@+-=123456789abcdef') sel_colors
  WHERE
    t1.value &lt; @size*@stretch
    AND t2.value &lt; @size) select_combinations
  GROUP BY row_number
UNION ALL
  SELECT
    CONCAT(
      REPEAT(SUBSTRING(@colors, value, 1), 2),
      '  ',
      SUBSTRING_INDEX(SUBSTRING_INDEX(@aggregated_data, '|', value), '|', -1)
    )
  FROM
    tinyint_asc
  WHERE
    value BETWEEN 1 AND @name_order
;
```

Subqueries in MySQL were first available in version 5.0. In prior versions, the use of joins was necessary and in many instances they were able to achieve the same result.

USING REGULAR EXPRESSIONS

Regular expressions become indispensable as soon as application requirements include validation or parsing of complicated text data. This book does a lot of that and it all builds on the foundations in this chapter. It is vital for a developer to have a good working knowledge of the regular expression language in order to increase productivity and to save time by avoiding the need to write special-purpose text parsers.

This section starts with general practices regarding regular expressions and then finishes with some examples. The expressions in the book can sometimes be complicated and difficult to read. This is one of the downsides of using regular expressions, but when they are used properly they can replace hundreds of lines of traditional text-parsing code and will outperform native PHP on long or complex strings.

General Patterns

Regular expressions in PHP start and end with a boundary character. This is usually a slash but it can be any character as long as it is the first character of the expression. Regular expressions in MySQL, by contrast, do not have a boundary character. For ease of reading, this book uses slash as a boundary character for all regular expressions unless they appear directly in a MySQL query. It is also common to use a hash character as a boundary in PHP. When an expression has many slashes the hash effectively avoids the need to escape every single non-terminal slash. In web applications this approach is very useful for URIs. These two lines are both functionally identical and valid regular expressions:

```
/yin\/yang/i
#yin/yang#i
```

In all cases a regular expression will match the pattern inside the boundaries. Modifiers can be placed after the closing boundary to alter the behavior of the regular expression. In the previous example, the modifier “`i`” is used to make the expression case-insensitive.

The pattern can range from simple (a tiny set of possible strings) to complex (an infinite number of possible matches). Complex regular expressions should always be commented to avoid confusion down the road. It is not uncommon for developers to come across a regular expression and ask, “What is that supposed to be doing?” even if they wrote it themselves just a few days earlier.

Matching a Range of Characters

Regular expressions are often used to match a string where a finite character set is expected to occur (or not occur). This is where regular expressions save a lot of time. Enclosing a set of characters in square brackets `[like this]` will match any of the characters in the set. The example in the preceding sentence will match the letters `l`, `i`, `k`, `e`, `t`, `h`, and `s` as well as a space (ASCII 0x20). Putting a caret (`^`) after the opening bracket will give you any character that is not one of those seven. Using a dash inside the brackets can specify ranges, for example: a valid username contains only letters `A-Z`, numbers, dashes, and underscores. There are also several short codes for predefined and frequently used character sets. These two regular expressions both match the username:

```
/^[A-Za-z0-9_\-]{3,15}$/
/^[\w\-]{3,15}$/
```

By design, those regular expressions will also ensure that the username is between three and fifteen characters long as indicated by the braces. They force the regular expression to match that many instances of the pattern preceding them. The brackets, combined with the numbers inside of them, are called a *quantifier*.

This is the first time in the book that `\w` *is used.* `\w` *will match word characters in a regular expression. So* `[A-Za-z0-9_]` *can be simplified into just* `\w`*. Two other useful and related shorthand characters are:*

- `\s` *matches whitespace characters such as spaces, line feeds, and carriage returns:* `[ \t\n\r]`
- `\d` *matches digits:* `[0-9]`

Using the uppercase version of a shorthand character will negate it. For example: `\S` *will match any character that is not whitespace.*

Even slightly changing the regular expression alters the meaning dramatically. Changing the braces to `{3,}` instead of `{3,15}` will match usernames that are at least three characters long but can be any length. Likewise, changing it to `{3}` will only allow usernames that are exactly three characters long.

The expression is anchored by a caret at the front and a dollar sign at the end. This ensures that the entire string is matched. Remove both of them and the resulting regular expression would match any substring that has at least three consecutive characters and matches the pattern (allowing bogus usernames). Removing the dollar sign will match any string that starts with at least three of the allowed characters. The inverse is true if just the caret sign is removed.

A more complex task would be to match an email address. Matching an email address is useful for dumb validation of input. The application can ensure that the user at least tried to enter valid data (but not that the data is actually valid). RFC 5322 documents the proper format for an email address. The task can be as easy as `/\w@\w/` or very difficult.

The email address is divided into a local-part and a domain-part. The local-part can contain almost any printable ASCII character. It excludes all brackets except curly brackets. It also excludes the

`@` sign, colon, semicolon, and commas, with the exception being if the local-part is surrounded by quotes, it can contain the excluded characters. These first two addresses have valid local-parts and the third does not (note the commas):

```
Boston.MA@example.com
"Boston,MA"@example.com
Boston,MA@example.com
```

The quote syntax is rarely seen and the RFC for the Simple Mail Transfer Protocol (RFC 5321) warns against it in section 4.2.1. Two possible regular expressions for the local-part of the domain (with and without quotes) look like this:

```
/"[\w!#$%&'*+\/=?^`{|}~.@()[\]\\;:,<>-]+"/
/[\w!#$%&'*+\/=?^`{|}~.-]+/
```

The plus sign is a quantifier that tells the regular expression engine to look for one or more of the previous expressions. Using an asterisk as a quantifier tells the engine to look for zero or more.

The domain-part has more strict rules to follow (and thus is a little easier to validate against). The domain can be any number of subdomains separated by dots. The subdomain can contain alphanumeric characters and dashes as long as it doesn't start or end with a dash. The domain can also be an IP address enclosed in square brackets. The resulting regular expressions for the domain portion might look like this:

```
/([A-Za-z0-9-]+\.)+[A-Za-z0-9-]+/
/\[([0-9]{1,3}\.){3}[0-9]{1,3}\]/
```

Complex groups can be enclosed in parentheses like they are in the previous example for matching a valid IP address. The expression will match the first three octets followed by a dot and then the final octet (which does not have a trailing dot).

Now that all the pieces are there they need to be put together using *alternation*. Using the pipe character to separate parts of the regular expression tells the engine that it can accept any of the parts as input. You can group alternations together using parentheses. The almost final regular expression looks like this:

```
/^(
  "[\w!#$%&'*+\/=?^`{|}~.@()[\]\\;:,<>-]+"
| [\w!#$%&'*+\/=?^`{|}~.-]+
) @ (
  ([A-Za-z0-9-]+\.)+[A-Za-z0-9-]+
| \[([0-9]{1,3}\.){3}[0-9]{1,3}\]
)$/x
```

The `x` modifier in the preceding example can be used to indicate that whitespace should be ignored. It is useful for making long expressions easier to read by making them span multiple lines.

The regular expressions for both parts have glaring errors. The local-part allows for a dot at the beginning and the end as well as consecutive dots, and the domain-part of the regular expression allows for hyphens at the beginning and end of subdomains, none of which is allowed by the RFC. Those errors need to be fixed for the regular expression to be accurate.

Expert Regular Expressions

The errors in the email expression can be fixed using simple regular expression syntax to detect a more limited character set for the beginning and end. Ironically that will produce a more complicated and difficult to read expression. Instead, lookarounds can be used.

Lookaheads and Lookbehinds

Lookaheads and *lookbehinds* (collectively called *lookarounds*) can be used to assert the presence or absence of characters in a string. In the email example they can be used to assert that the first character in the local-part is not a dot and neither is the last character. They are each special types of groups. When the start of a group (opening parenthesis) is followed by a question mark, it indicates to the engine that the type for that group will follow. There are many types of groups, all of which are covered in this chapter.

Lookaheads use an equal sign and lookbehinds use a less-than sign followed by an equal sign. Using both, the engine can match the letter `b` that is immediately preceded by `a` and followed by `c`:

```
/(?<=a)b(?=c)/
```

Lookaheads and lookbehinds can also be negated using an exclamation point instead of an equal sign. The preceding regular expression can easily be modified to be the letter `b` that is not preceded by `a` or followed by `c`:

```
/(?<!a)b(?!c)/
```

The entire string will not match if any negative lookahead or negative lookbehind matches. So `abx` *and* `xbc` *both fail to match. A slightly more complicated regular expression that succeeds for both those strings but still fails for* `abc` *would be:*

```
/(?<!a)b|b(?!c)/
```

All lookarounds are zero-width, which means that they do not count toward the match. This can be useful for string replacement where you do not want the beginning or end of a string to be replaced. They can then be used to help out with the email problem as well. The problem can be simplified by ignoring the complexity of the local-part for now and saying that the expression only needs to match a word containing dots that does not start or end with a dot. The expression `[\w.]+` will match alphanumeric characters and dots. A negative lookahead and a negative lookbehind can be used together so that it doesn't match words that start or end with a dot:

```
/^(?!\.)[\w.]+(?<!\.)$/
```

Caution must be taken when using the dot character. It does not need to be escaped inside the character set, but outside it must be. Removing the slash before the first or last dot will read "not ending/beginning with any character," which is clearly not desirable. Changing the last exclamation point to an equal sign will only match strings that do end in a dot. Using negative lookarounds to catch leading and trailing dots in the local-part and hyphens in the domain-part lead to a new completed regular expression:

```
/^(
  "[\w!#$%&'*+\/=?^`{|}~.@()[\]\\;:,<>-]+"
| (?!\.)[\w!#$%&'*+\/=?^`{|}~.-]+(?<!\.)
) @ (
  ((?!-)[\w-]+(?<!-)\.)+(?!-)[\w-]+(?<!-)
| \[([0-9]{1,3}\.){3}[0-9]{1,3}\]
)$/x
```

The new expression will match any valid email address and will fail on an address that does not follow the rules outlined in the RFCs. Lookarounds are just one type of group. There is an entirely different type called capture groups that is also very common.

Capturing Data

Regular expressions have the ability to capture data. Starting a group without providing a type (a parenthesis that is not followed by unescaped question mark) will cause that group to be captured. Data from the capture group can be referenced both from inside the regular expression and PHP. When referenced from within the same expression it is referred to as a *back-reference*. Back-references can be achieved by using `\#` where `#` is the number of the captured groups. You can use back-references to match both a single and double-quoted string with the same regular expression:

```
/('|")[^\1]*?\1/
```

The back-references (`\1`) ensure that the end quote is of the same type as the opening quote and that the quoted string can contain other quotes as long as they are not the same type. It is important that the asterisk is made lazy using the question mark. Otherwise if there are multiple quoted strings inside the subject, the expression will match it as if it contains only one giant quoted string.

Any quantifier can be made lazy using the question mark (even the question mark quantifier itself). The question mark serves several purposes in regular expressions:

- *To mark the previous character, group, or character class as optional.*
- *To mark the previous quantifier as lazy. A lazy quantifier will quit matching as soon as it can. It will continue on to the next part of the regular expression if it can. In contrast, a greedy quantifier (no question mark) will keep matching as long as it can.*
- *To indicate the type of a group (if placed immediately after the opening parenthesis).*

Most programmers are accustomed to escaping quotes inside a quoted string to prevent the string from terminating. The previous regular expression does not behave properly in that situation.

By using a negative lookbehind and alternation the top example string can be matched using the bottom regular expression:

```
"Hello \"my\" world"
/('|")([^\1]|\\\1)*?(?<!\\)\1/
```

The alternation ensures that the engine behaves as intended when it encounters a backslash followed by the quote type. The negative lookbehind then ensures that the expression keeps looking for a closing quote instead of terminating lazily when it finds the first inner quote.

Sometimes it is undesirable to capture data. In those cases it can be avoided by putting `?:` at the beginning of the group. The colon turns it into a non-capturing group. Non-capturing groups are extremely useful for keeping the number of back-references available down to a minimum and makes writing code much easier and cleaner. Sometimes it is desirable to have a lot of back-references. In these cases it is useful to name them so as to avoid confusion ("Is that group `\4` or is it `\5`?").

Naming a capture group is as easy as putting `P<name_here>` after the question mark. A named group can then be back-referenced using `(?=name_here)` in the expression. A simple example pattern will discover Pseudo-Shakespearean questions in the subject text. The regular expression on the first line will match all subsequent subjects:

```
/(?P<word>(?:\w+\s?)+) or not (?P=word)\?$/i
To be or not to be?
PHP or not PHP?
Sleep or not sleep?
```

Named capture groups are used later in this chapter when writing a PHP script that verifies an email address. For now it is useful to go over documenting regular expressions.

Documenting Regular Expressions

Regular expressions can also be commented. The comment syntax is rarely used in this book. An alternative method is to use PHP comments above the regular expression. However, it is a good practice to comment individual alternations and subpatterns when the code contains complex regular expressions (like the email expression).

Comments are a special type of non-capturing group that starts with a `?#`. A comment can very easily be added into any expression but they are easiest to read in expressions where the `x` modifier is used and whitespace can be utilized liberally. A comment inside a regular expression will look like this:

```
(?# comment goes here)
```

The completed email regular expression from before is altered to include comments when it is used in the next section and in the code examples that accompany this book.

Putting It All Together in PHP

PHP uses Perl-style regular expressions via its `preg_` family of functions. PHP has also supported POSIX-style regular expressions via `ereg_`; however, those functions are deprecated in PHP 5.3 and should not be used anymore.

The PHP example in this section completely validates an email address. It supports two types of validation: lazy and complete. The lazy method simply returns `true` if the regular expression matches and if the string appears to be a valid email. However, that only serves to make using a fake email more difficult but not impossible. The complete method also checks DNS to make sure the domain name exists and then uses SMTP to connect to the Mail Transfer Agent (MTA) and make sure the user exists.

Each DNS zone for a domain can contain one or more Mail Exchange (MX) records that tell mail clients and transfer agents what server to connect to in order to send and retrieve mail. RFC 2810 states that a domain can receive email even if no MX records are found or valid for it. In that case, the mail client will attempt to connect to the hostname itself. PHP has a handy function called `getmxrr()` that will get the MX records. Prior to PHP 5.3 the function would only work on UNIX/Linux-based systems. As of PHP 5.3 it will also work on Windows without any messy hacks. The `getMX()` method looks like this:

```
private function getMX( $hostname ) {
  $hosts = array();
  $weights = array();
  getmxrr( $hostname, $hosts, $weights );
  $results = array();
  foreach ( $hosts as $i => $host )
    $results[ $host ] = $weights[$i];
  arsort($results, SORT_NUMERIC);
  $results[$hostname] = 0;
  return $results;
}
```

As mentioned earlier, RFC 2810 states that the domain itself is a valid location to look for an email server, so the code appends the domain to the end of the result array but gives it zero weight and adds it after the sort so that it will be lighter (lower priority) than any MX records that were returned from the DNS server.

The second method takes the MX records and tries to connect to them on port 25 (SMTP) in order until one succeeds. If it reaches the end of the list and still doesn't have a valid connection, either the host — and therefore the entire email address — is bogus or the server is down. This example assumes the server should be up and returns `false` under the case where it is unreachable.

The new method called `openSMTPSocket()` takes a host name, uses it to call `getMX()`, loops through all the hosts, and returns a valid socket pointer if it can:

```
private function openSMTPSocket( $hostname ) {
  $hosts = $this->getMX($hostname);
  foreach ( $hosts as $host => $weight ) {
    if ( $sock = @fsockopen($host, self::SMTP_PORT,
         $errno, $errstr, self::CONN_TIMEOUT) ) {
      stream_set_timeout($sock, self::READ_TIMEOUT);
      return $sock;
    }
  }
  return null;
}
```

With a valid socket pointer the example can then say "hello" to the MTA (telling it what host you are looking for in case there is more than one host on that server) and then ask if the email is valid. If it is valid it returns `true`. In all cases, it closes the socket handle when it is done with it:

```
private function validateUser( $hostname, $user ) {
  if ( $sock = $this->openSMTPSocket($hostname) ) {
    $this->smtpSend("HELO $hostname");
    $this->smtpSend("MAIL FROM: <$user@$hostname>");
    $resp = $this->smtpSend("RCPT TO: <$user@$hostname>");

    $valid = (preg_match('/250|45(1|2)\s/') == 1);
    fclose($fp);
    return $valid;
  } else {
    return false;
  }
}

private function smtpSend( $sock, $data ) {
  fwrite($sock, "$data\r\n")
  return fgets($sock, 1024);
}
```

The email address may be definitively valid (response 250) or gray-listed (responses 451 and 451) on the MTA. The method uses a regular expression to test the response and returns true in any of those cases. In a completed application it makes sense to return a confidence score instead of a Boolean. The score may be zero if the regular expression doesn't match or the MTA returns negative when asking if the user exists. It may be one if the MTA verifies the user and the user is not gray-listed, and 0.25 and 0.75 might be used for "the SMTP server is unreachable" and "the user is gray-listed," respectively. That way an application can choose to only allow a user to register if the score is 0.5 or higher.

The final piece of the puzzle is the class that holds it all together — the rest of the email address verification class looks like the code in Listing 1-55.

LISTING 1-55: EmailValidator.class.php

Available for download on Wrox.com

```
<?php

class EmailValidator {
  const CONN_TIMEOUT = 10;
  const READ_TIMEOUT = 5;
  const SMTP_PORT = 25;
  private $email;

  public function __construct( $email ) { $this->email = $email; }

  private function getParts() {
    $regex = <<<__REGEX__
/^(?P<user>
  "[\w!#$%&'*+\/=?^`{|}~.@()[\]\\;:,<>-]+" (?# quoted username )
| (?!\.)[\w!#$%&'*+\/=?^`{|}~.-]+(?<!\.)   (?# non-quoted username )
) @ (?P<host>
  (?:(?!-)[\w-]+(?<!-)\.)+(?!-)[\w-]+(?<!-) (?# host )
| \[([0-9]{1,3}\.){3}[0-9]{1,3}\] (?# host IP address )
)$/x
__REGEX__;

    return ( preg_match($regex, $this->email, $matches) ? $matches : null);
  }

  public function isValid( $lazy ) {
    static $valid = null;

    if ( $lazy ) return ( $this->getParts() != null );
    if ( $valid !== null ) return $valid;
    $valid = false;

    if ( $parts = $this->getParts() ) {
      $valid = $this->validateUser( $parts['host'], $parts['user'] );
    }
    return $valid;
  }

  private function validateUser( $hostname, $user ) { ... }
  private function openSMTPSocket( $hostname ) { ... }
  private function smtpSend( $sock, $data ) { ... }
  private function getMX( $hostname ) { ... }
};
?>
```

It is common for ISPs to block outgoing connections on port 25. This tactic forces the customer to use the ISP's mail relay and makes it easier to thwart people who are trying to use the network for spam. Unfortunately, it also means that if the example application in this section is being run on a home network it is likely that the port will be blocked and the application will always return `false` *for every email address. The only two solutions are to get the ISP to unblock the port (much more likely on hosting providers than consumer ISPs) or run the PHP from a computer living on a different ISP's network.*

Lazy validation (regular expression only) will always work regardless of the ISP's firewall settings but does not have as high a confidence factor.

The email regular expression changed slightly between the previous section and this. It now captures the hostname and user in named groups so that they can be easily referenced by PHP. It also makes the host pattern non-capturing so the matches don't end up with extra data that isn't needed. Passing a third parameter to `preg_match()` captures the matches and capture groups in an array. The output of the `$matches` array on the input `andrew@example.com` looks like this:

```
Array
(
    [0] => Array
```

```
        (
            [0] => andrew@example.com
            [user] => andrew
            [1] => andrew
            [host] => example.com
            [2] => example.com
        )

)
```

Notice how the numbered matches are still kept in the result so each named group can be referenced two different ways. It also means that changing a group from unnamed to named will not affect the ordering of the unnamed groups. The first index (index zero) always equals the entire match string. The email testing class is now complete; however, it is just one of the uses for regular expressions in PHP.

Replacing Strings

The email regular expression can also be used to replace all valid emails in a string with an HTML link to send a mail to the user. To make things more interesting the next example replaces each email address username with a mailto link and each domain with a link directly to the domain. Assume that `$emailRegex` is filled with the entire email regular expression from the previous example but with the anchors removed so it can match a partial string:

```
preg_replace( $emailRegex,
              '<a href="mailto:\1">\1</a>@<a href="http://\2">\2</a>'
              $testString );
```

This example shows a simple replacement. For more complex replacements a callback function can be used to replace the string with a computed value. Callback functions are used extensively in later chapters. Listing 1-56 is a utility class can be used to replace all email addresses in a given text with obfuscated links that can then be clicked to open the email client but won't give away the email to data miners:

LISTING 1-56: EmailLinker.php

Available for download on Wrox.com

```
<?php
class EmailLinker {

   public function getJavascript() {
     return <<<__JS__
<script type="text/javascript" language="javascript">
function mailDecode( url ) {
  var script=document.createElement('script');
  script.src = '?mail='+url;
  document.body.appendChild(script);
}
</script>
__JS__;
  }

  public function redirectIfNeeded() {
    if ( array_key_exists('mail', $_GET) ) {
      header("Location: mailto:".base64_decode($_GET['mail']));
      exit;
    }
  }

  private function emailReplaceCallback( $matches ) {
    $encoded = base64_encode($matches[0]);
    return '<a href="?mail='.urlencode($encoded).'"'.
           ' onclick="mailDecode(\''.$encoded.'\'); return false;">'.
           'email '.$matches['user'].'</a>';
  }

  public function link( $text ) {
    $emailRegex = <<<__REGEX__
/(?P<user>
  "[\w!#$%&'*+\/=?^`{|}~.@()[\]\\;:,<>-]+" (?# quoted username )
| (?!\.)[\w!#$%&'*+\/=?^`{|}~.-]+(?<!\.)   (?# non-quoted username )
) @ (?P<host>
  (?:(?!-)[\w-]+(?<!-)\.)+(?!-)[\w-]+(?<!-) (?# host )
| \[([0-9]{1,3}\.){3}[0-9]{1,3}\] (?# host IP address )
)/x
__REGEX__;

    return preg_replace_callback($emailRegex,
             array($this,'emailReplaceCallback'), $text );
  }
}
?>
```

The `preg_replace()` callback line and the callback function that is used are both highlighted. The class has corresponding JavaScript that can be retrieved using `getJavascript()` and echoed into the header of the document. The class will work without the JavaScript, but it works much better with it. It also relies on the method `redirectIfNeeded()` being called before any output. The redirect will detect if the user clicked an email link and will send the user to the properly formatted `mailto:` URL.

The resulting text does not include the email address anywhere in it but still allows users to be emailed. It is not completely secure: if spammers or malicious users went through the trouble of Base 64 decoding the string or following the link they could get the users' email addresses. But it does prevent all but the most sophisticated email data mining techniques to the point where a data miner would have to write a script specifically for this example.

PHP has been the primary focus for regular expressions up to this point. It is also possible to perform basic regular expression matches in MySQL in order to filter the data before it even gets to the PHP.

Regular Expressions in MySQL

MySQL has extremely limited support for the now familiar Perl-style regular expressions. It uses a modified POSIX format so support is limited to basic character classes, alternations, anchors, and quantifiers. Lookarounds, back-references, and capture groups are not allowed. However, regular expressions in MySQL can be useful for matching simple strings and for narrowing down a result set for later culling in PHP.

Regular expressions in MySQL are referenced using the `REGEXP` and `REGEXP BINARY` operations. The latter is case-sensitive whereas the former is not. Expression can also be negated using the `NOT` operation. For example, a links table can be queried for all links that point to example.com or its subdomains:

```
SELECT  *
  FROM  `links`
  WHERE `url` REGEXP 'https?://([a-z0-9-]*\.)*example\.com';
```

MySQL doesn't have the escaped characters that many other regular expression engines have. Instead it has special keywords that can be used in the expression to match a range of characters. Table 1-2 shows the MySQL character classes and their PHP equivalents.

TABLE 1-2: MySQL Character Classes with PHP Equivalents

MYSQL (POSIX)	PHP	LONG FORM (EITHER)
`[:alpha:]`		`[A-Za-z]`
`[:alnum:]`		`[A-Za-z0-9]`
`[:blank:]`		`[ \t\r\n]`
`[:cntrl:]`		`[\x00-\x1F\x7F]`
`[:digit:]`	`\d`	`[0-9]`
`[:graph:]`		`[\x21-\x7E]`
`[:lower:]`		`[a-z]`
`[:print:]`		`[\x20-\x7E]`
`[:punct:]`		`[!"#$%&'()*+,\-./:;<=>?@[\\\]^_`{|}~]`
`[:space:]`	`\s`	`[ \t\r\n\v\f]`
`[:upper:]`		`[A-Z]`
`[:xdigit:]`		`[A-Fa-f0-9]`

It is worth noting that `[:print:]` will match any character that can be printed to the screen and that `[:graph:]` is identical except that it will not match a space character (because space is not graphical).

The special character classes can be combined with other characters or classes. Because the subdomain of the previous regular expression consists of alphanumeric characters or hyphens it can be rewritten as:

```
SELECT  *
  FROM  `links`
  WHERE `url` REGEXP 'https?://([:alnum:-]*\.)*example\.com';
```

The word boundary shorthand (`\b`) is replaced in MySQL by two character classes. One matches the end of a word and the other matches the beginning. Like `\b` they are both zero-width. To match all messages in a forum that contain the word HTML but not XHTML, a simple regular expression could be used:

```
SELECT * FROM `forum` WHERE `body` REGEXP '[[:<:]]HTML[[:>:]]'
```

The regular expression functionality built into MySQL is sufficient under almost all circumstances. However, if a PHP program ends up doing a lot of post-filtering of the result set based on the output of a complex regular expression, it may be time to extend MySQL.

Using LIB_MYSQLUDF_PREG

The LIB_MYSQLUDF_PREG library is a set of MySQL User Defined Functions that allow Perl-compatible regular expressions (same as PHP) to be executed in a MySQL query. Besides allowing for back-references and lookarounds it also allows capture groups to be selected by the query.

The library must be installed from source. Chapter 7 on MySQL UDFs provides more details on installing from source code. If you are already familiar with the typical build process, it can be installed in three lines:

```
./configure
make
make installdb
```

It requires the libpcre headers and MySQL to be on the system. If they are installed in unusual locations there are a few extra steps. The location of either can be easily specified manually:

```
./configure --with-pcre=/path/to/libpcre --with-mysql=/path/to/mysql/config
```

Almost anything that PHP is capable of can also be done in MySQL once the library is installed.

Capturing Data

It is often useful to capture part of a complex string in a data set. One example is to query the database for a list of all domains that have registered users and return the number of users from each. There is no need for a complicated email matching expression like the one used in previous examples because the application can assume that if the email made its way into the database, it is already valid. The query looks like this:

```
SELECT
  PREG_CAPTURE('/@([^@]+)$/' , `email`, 1) AS `domain`,
  COUNT(*) AS `count`
FROM `users`
GROUP BY `domain`
```

However, if the application does this often for the same string it is a sign that a new column should be added to the table. Because a column cannot be returned as an array, the `PREG_CAPTURE` function takes a third parameter that is the group to return. If `PREG_CAPTURE` is replaced by `PREG_POSITION`, then instead of the domain it will return the index of the start of the first group. In MySQL the index is one-based so when querying for the position of the first character it is index `1`, not `0`. The default for the group parameter is `0`, which returns the entire match, `1` returns the first match, and so on, like in PHP.

String Replacement

When selecting from or updating a table, it is useful to modify an existing column. For example, the application may need to display a sample report that blanks out certain information such as revenue numbers. MySQL can replace the data at query time instead of having to loop through the entire data set in PHP when displaying it:

```
SELECT
  PREG_REPLACE('/\$[:digit:]*(\.[:digit:]+)?/',
               '[subscriber-only]', `body`)
  AS `body`
FROM `reports`;
```

String replacements are also useful when doing updates to a table. Because the library supports back-references it is easy to make complex replacements.

Filtering a Query Based on a Regular Expression

The built-in MySQL regular expression functionality is primarily useful for returning a Boolean or filtering an entire result set. LIB_MYSQLUDF_PREG can do that too.

One alias for `REGEXP` in MySQL is `RLIKE`. Similarly, the UDF includes a function `PREG_RLIKE` that returns `1` if the pattern matched and `0` if there isn't any match. The behavior is identical to the built-in MySQL functionality except that it allows for more complex Perl-compatible regular expressions. The syntax is also slightly different because the latter is a UDF. The following two queries have identical output:

```
SELECT  *
  FROM  `links`
  WHERE `url` REGEXP 'https?://([:alnum:-]*\.)*example\.com';

SELECT  *
  FROM  `links`
  WHERE PREG_RLIKE('https?://([\w-]*\.)*example\.com',`url`);
```

Regular expressions are slower than other methods of string matching because they need to compile the expression in order to match against it and each position in the string may take several passes to look for a match. For those reasons a developer should always opt to use basic string matching such as `LIKE` to filter results. However, when more complex string matching and replacements are needed Regular Expressions are the only way to go.

SUMMARY

This chapter covered both PHP and MySQL essentials for the expert developer.

It covered the object-oriented design approach now available in PHP including a number of key design patterns. It is impossible to master PHP without first having a complete understanding of class instantiation, interfaces, class methods, and constants.

This chapter also went over the foundations of MySQL. Being able to use MySQL joins is essential in a normalized relational database design where data is maintained in multiple tables. Combined with the ability to aggregate and group results, and leverage subqueries and derived tables, you can master all the power and flexibility that MySQL has to offer in retrieving your information.

Though MySQL provides options for flow control and logic within SQL, as a developer you should always determine what is best performed at the database level and what is best performed within your PHP code.

The chapter concluded with regular expressions — the cornerstone of string manipulation — and parsing in any programming language, including PHP and MySQL.