Chapter 7. Storing Information with Databases

The HTML and CSS that give your web site its pretty face reside in individual files on your web server. So does the PHP code that processes forms and performs other dynamic wizardry. There's a third kind of information necessary to a web application, though: data. And while you can store data such as user lists and product information in individual files, most people find it easier to use databases, which are the focus of this chapter.

Lots of information falls under the broad umbrella of "data":

- Who your users are, such as their names and email addresses.
- What your users do, such as message board posts and profile information.
- The "stuff" that your site is about, such as a list of record albums, a product catalog, or what's for dinner.

There are three big reasons why this kind of data belongs in a database instead of in files: convenience, simultaneous access, and security. A database program makes it much easier to search for and manipulate individual pieces of information. With a database program, you can do things such as change the email address for user `Duck29` to `ducky@ducks.example.com` in one step. If you put usernames and email addresses in a file, changing an email address would be much more complicated: read the old file, search through each line until you find the one for `Duck29`, change the line, and write the file back out. If, at same time, one request updates `Duck29`'s email address and another updates the record for user `Piggy56`, one update could be lost, or (worse) the data file corrupted. Database software manages the intricacies of simultaneous access for you.

In addition to searchability, database programs usually provide you with a different set of access control options compared to files. It is an exacting process to set things up properly so that your PHP programs can create, edit, and delete files on your web server without opening the door to malicious attackers who could abuse that setup to alter your PHP scripts and data files. A database program makes it easier to arrange the appropriate levels of access to your information. It can be configured so that your PHP programs can read and change some information, but only read other information. However the database access control is set up, it doesn't affect how files on the web server are accessed. Just because your PHP program can change values in the database doesn't give an attacker an opportunity to change your PHP programs and HTML files themselves.

The word *database* is used in a few different ways when talking about web applications. A database can be a pile of structured information, a program (such as MySQL or Oracle) that manages that structured information, or the computer on which that program runs. In this book, I use "database" to mean the pile of structured information. The software that manages the information is a *database program*, and the computer that the database program runs on is a *database server*.

Most of this chapter uses the PEAR DB database program abstraction layer. This is an add-on to PHP that simplifies communication between your PHP program and your database program. PEAR (PHP Extension and Application Repository) is a collection of useful modules and libraries for PHP. The DB module is one of the most popular PEAR modules and is bundled with recent versions of PHP. If your PHP installation doesn't have DB installed (Section 7.2, later in this chapter, shows you how to check), see Section A.3 for instructions on how to install it.

When DB isn't available, you need to rely on other PHP functions to talk to your database program. The appropriate set of functions varies with each database program. Some of the more exotic features of your database program may only be

accessible through the database-specific functions. Later in this chapter, Section 7.12 discusses shows how to work with the functions in the mysqli extension, which talks to MySQL (Versions 4.1.2 and greater).

7.1 Organizing Data in a Database

Information in your database is organized in *tables*, which have rows and columns. (Columns are also sometimes referred to as *fields*.) Each column in a table is a category of information, and each row is a set of values for each column. For example, a table holding information about dishes on a menu would have columns for each dish's ID, name, price, and spiciness. Each row in the table is the group of values for on particular dish—for example, "1," "Fried Bean Curd," "5.50," and "0" (meaning not spicy).

You can think of a table organized like a simple spreadsheet, with column names across the top, as shown in Figure 7-1.

Figure 7-1. Data organized in a grid

ID	Name	Price	Is spicy?
1	Fried Bean Curd	5.50	0
2	Braised Sea Cucumber	9.95	0
3	Walnut Bun	1.00	0
4	Eggplant with Chili Sauce	6.50	1

One important difference between a spreadsheet and a database table, however, is that the rows in a database table have no inherent order. When you want to retrieve data from a table with the rows arranged in a particular way (e.g., in alphabetic order by student name), you need to explicitly specify that order when you ask the database for the data. The *SQL Lesson:* ORDER BY and LIMIT sidebar in this chapter describes how to do this.

SQL (Structured Query Language) is a language to ask questions of and give instructions to the database program. Your PHP program sends SQL queries to a database program. If the query retrieves data in the database (for example, "Find me all spicy dishes"), then the database program responds with the set of rows that match the query. If the query changes data in the database (for example, "Add this new dish" or "Double the prices of all nonspicy dishes"), then the database program replies with whether or not the operation succeeded.

SQL is a mixed bag when it comes to case-sensitivity. SQL keywords are not case-sensitive, but in this book they are always written as uppercase to distinguish them from the other parts of the queries. Names of tables and columns in your queries generally are case-sensitive. All of the SQL examples in this book use lowercase column and table names to help you distinguish them from the SQL keywords. Any literal values that you put in queries are case-sensitive. Telling the database program that the name of a new dish is `fried bean curd` is different than telling it that the new dish is called `FRIED Bean Curd`.

Almost all of the SQL queries that you write to use in your PHP programs rely on one of four SQL commands: `INSERT`, `UPDATE`, `DELETE`, or `SELECT`. Each of these commands is described in this chapter. Section 7.3 describes the `CREATE TABLE` command, which you use to make new tables in your database.

To learn more about SQL, read *SQL in a Nutshell*, by Kevin E. Kline (O'Reilly). It provides an overview of standard SQL as well as the SQL extensions in MySQL, Oracle, PostgreSQL, and Microsoft SQL Server. For more in-depth information about working with PHP and MySQL, read *Web Database Applications with PHP & MySQL*, by Hugh E. Williams and David Lane (O'Reilly). *MySQL Cookbook*, by Paul DuBois (O'Reilly) is also an excellent source for answers to lots of SQL and MySQL questions.

7.2 Connecting to a Database Program

To use PEAR DB in a PHP program, first you have to load the DB module. Use the `require` construct, as shown in Example 7-1.

Example 7-1. Loading an external file with require

```
require 'DB.php';
```

Example 7-1 tells the PHP interpreter to execute all of the code in the file *DB.php*. *DB.php* is the main file of the PEAR DB package. It defines the functions that you use to talk to your database.

Similar to `require` is `include`. These constructs differ in how they handle errors. If you try to include or require a file that doesn't exist, `require` considers that a fatal error and your PHP program ends. The `include` construct is more forgiving and just reports a warning, allowing your program to continue running.

After the DB module is loaded, you need to establish a connection to the database with the `DB::connect( )` function. You pass `DB::connect( )` a string that describes the database you are connecting to, and it returns an *object* that you use in the rest of your program to exchange information with the database program.

An object is a new data type. It's a bundle of some data and functions that operate on that data. PEAR DB uses objects to provide you with a connection to the database. The double colons in the `DB::connect( )` function call are a way of telling the PHP interpreter that you're calling a special function based on an object.

Example 7-2 shows a call to `DB::connect( )` that connects to MySQL.

Example 7-2. Connecting with DB::connect( )

```
require 'DB.php';
$db = DB::connect('mysql://penguin:top^hat@db.example.com/restaurant');
```

The string passed to `DB::connect( )` is called a Data Source Name (DSN). Its general form is:

```
db_program://user:password@hostname/database
```

In Example 7-2, the DSN tells PEAR DB to connect to MySQL running on the database server `db.example.com` as user `penguin` with the password `top^hat`, and to access the `restaurant` database on that server.

PEAR DB supports 13 options for the `db_program` part of the DSN. These are listed in Table 7-1.

Table 7-1. PEAR DB db_program options

db_program	Database program
`dbase`	dBase
`fbsql`	FrontBase
`ibase`	InterBase
`ifx`	Informix
`msql`	Mini SQL
`mssql`	Microsoft SQL Server
`mysql`	MySQL (versions <= 4.0)
`mysqli`	MySQL (versions >= 4.1.2)
`oci8`	Oracle (Versions 7, 8, and 9)
`odbc`	ODBC
`pgsql`	PostgreSQL
`sqlite`	SQLite
`sybase`	Sybase

When your database program is running on the same computer as your web server, specify `localhost` as the *hostname* part of the DSN, as shown in Example 7-3.

Example 7-3. Connecting to localhost

```
$db = DB::connect('mysql://penguin:top^hat@localhost/restaurant');
```

If all goes well with `DB::connect( )`, it returns an object that you use to interact with the database. If there is a problem connecting, it returns a different kind of object that contains information about what went wrong. The `DB::isError( )` function checks whether the object contains error information. Use it to make sure that the connection was made before going forward in your program. Example 7-4 uses `DB::isError( )` to verify that `DB::connect( )` succeeded.

Example 7-4. Checking for connection errors

```
require 'DB.php';
$db = DB::connect('mysql://penguin:top^hat@db.example.com/restaurant');
if (DB::isError($db)) { die("Can't connect: " . $db->getMessage( )); }
```

The `DB::isError( )` function returns `true` if the object passed to it contains error information. The `die( )` function prints out a message and then causes the script to quit. In this case, the message is the string `Can't connect`: followed by the results of the `$db->getMessage( )` call. The `getMessage( )` function returns more information about the error.

Earlier, I said that an object is a bundle of data and functions that operate on that data. A `->` after an object tells the PHP interpreter that you want to call one of those functions in the object. Once you have called `DB::connect`, you use the functions in the object to interact with the database. The code `$db->getMessage( )` means "call the `getMessage( )` function inside the `$db` object." In this case, the `$db` object holds error information and the `getMessage( )` function prints out some of that information.

For example, if `top^hat` is the wrong password for user `penguin`, Example 7-4 prints:

```
Can't connect: DB Error: connect failed
```

7.3 Creating a Table

Before you can put any data into or retrieve any data from a database table, you must create the table. This is usually a one-time operation. You tell the database program to create a new table once. Your PHP program that uses the table may read from or write to that table every time it runs. But it doesn't have to re-create the table each time. If a database table is like a spreadsheet, then creating a table is like making a new spreadsheet file. After you create the file, you open it many times to read or change it.

The SQL command to create a table is `CREATE TABLE`. You provide the name of the table and the names and types of all the columns in the table. Example 7-5 shows the SQL command to create the `dishes` table pictured in Figure 7-1.

Example 7-5. Creating the dishes table

```
CREATE TABLE dishes (
    dish_id INT,
    dish_name VARCHAR(255),
    price DECIMAL(4,2),
    is_spicy INT
)
```

Example 7-5 creates a table called `dishes` with four columns. The `dishes` table looks like the one pictured in Figure 7-1. The columns in the table are `dish_id`, `dish_name`, `price`, and `is_spicy`. The `dish_id` and `is_spicy` columns are integers. The `price` column is a decimal number. The `dish_name` column is a string.

After the literal `CREATE TABLE` comes the name of the table. Then, between the parentheses, is a comma-separated list of the columns in the table. The phrase that defines each column has two parts: the column name and the column type. In Example 7-5, the column names are `dish_id`, `dish_name`, `price`, and `is_spicy`. The column types are `INT`, `VARCHAR(255)`, `DECIMAL(4,2)`, and `INT`.

Some column types include length or formatting information in the parentheses. For example, `VARCHAR(255)` means "a variable length character column that is at most 255 characters long." The type `DECIMAL(4,2)` means "a decimal number

with two digits after the decimal place and four digits total." Table 7-2 lists some common types for database table columns.

Table 7-2. Common database table column types

Column type	Description
`VARCHAR(length)`	A variable length string up to *length* characters long.
`INT`	An integer.
`BLOB`[1]	Up to 64k of string or binary data.
`DECIMAL(total_digits,decimal_places)`	A decimal number with a total of *total_digits* digits and *decimal_places* digits after the decimal point.
`DATETIME`[2]	A date and time, such as `1975-03-10 19:45:03` or `2038-01-18 22:14:07`.

[1] PostgreSQL calls this `BYTEA` instead of `BLOB`.

[2] Oracle calls this `DATE` instead of `DATETIME`.

Different database programs support different column types, although all database programs should support the types listed in Table 7-2. The maximum and minimum numbers that the database can handle in numeric columns and the maximum size of text columns varies based on what database program you are using. For example, MySQL allows `VARCHAR` columns to be up to 255 characters long, but Microsoft SQL Server allows `VARCHAR` columns to be up to 8,000 characters long. Check your database manual for the specifics that apply to you.

To actually create the table, you need to send the `CREATE TABLE` command to the database. After connecting with `DB::connect( )`, use the `query( )` function to send the command as shown in Example 7-6.

Example 7-6. Sending a CREATE TABLE command to the database program

```
require 'DB.php';
$db = DB::connect('mysql://hunter:w)mp3s@db.example.com/restaurant');
if (DB::isError($db)) { die("connection error: " . $db->getMessage( )); }
$q = $db->query("CREATE TABLE dishes (
        dish_id INT,
        dish_name VARCHAR(255),
        price DECIMAL(4,2),
        is_spicy INT
)");
```

Section 7.4, explains `query( )` in much more detail.

The opposite of `CREATE TABLE` is `DROP TABLE`. It removes a table and the data in it from a database. Example 7-7 shows the syntax of a query that removes the `dishes` table.

Example 7-7. Removing a table

```
DROP TABLE dishes
```

Once you've dropped a table, it's gone for good, so be careful with `DROP TABLE`!

7.4 Putting Data into the Database

Assuming the connection to the database succeeds, the object returned by `DB::connect( )` provides access to the data in your database. Calling that object's functions lets you send queries to the database program and access the results. To put some data into the database, pass an `INSERT` statement to the object's `query( )` function, as shown in Example 7-8.

Example 7-8. Inserting data with query( )

```
require 'DB.php';
$db = DB::connect('mysql://hunter:w)mp3s@db.example.com/restaurant');
if (DB::isError($db)) { die("connection error: " . $db->getMessage( )); }
$q = $db->query("INSERT INTO dishes (dish_name, price, is_spicy)
    VALUES ('Sesame Seed Puff', 2.50, 0)");
```

Just like with the `$db` object that `DB::connect( )` returns, the `$q` object that `query( )` returns can be tested with `DB::isError( )` to check whether the query was successful. Example 7-9 attempts an `INSERT` statement that has a bad column name in it. The `dishes` table doesn't contain a column called `dish_size`.

Example 7-9. Checking for errors from query( )

```
require 'DB.php';
$db = DB::connect('mysql://hunter:w)mp3s@db.example.com/restaurant');
if (DB::isError($db)) { die("connection error: " . $db->getMessage( )); }
$q = $db->query("INSERT INTO dishes (dish_size, dish_name, price, is_spicy)
    VALUES ('large', 'Sesame Seed Puff', 2.50, 0)");
if (DB::isError($q)) { die("query error: " . $q->getMessage( )); }
```

Example 7-9 prints:

```
query error: DB Error: syntax error
```

Instead of calling `DB::isError( )` after every query to see if it succeeded or failed, it's more convenient to use the `setErrorHandling( )` function to establish a default error-handling behavior. Pass the constant `PEAR_ERROR_DIE` to `setErrorHandling( )` to have your program automatically print an error message and exit if a query fails. Example 7-10 uses `setErrorHandling( )` and has the same incorrect query as Example 7-9.

Example 7-10. Automatic error handling with setErrorHandling( )

```
require 'DB.php';
$db = DB::connect('mysql://hunter:w)mp3s@db.example.com/restaurant');
if (DB::isError($db)) { die("Can't connect: " . $db->getMessage( )); }
```

```
// print a message and quit on future database errors
$db->setErrorHandling(PEAR_ERROR_DIE);

$q = $db->query("INSERT INTO dishes (dish_size, dish_name, price, is_spicy)
    VALUES ('large', 'Sesame Seed Puff', 2.50, 0)");
print "Query Succeeded!";
```

SQL Lesson: *INSERT*

The `INSERT` command adds a row to a database table. Example 7-11 shows the syntax of `INSERT`.

Example 7-11. Inserting data

```
INSERT INTO table (column1[, column2, column3, ...])
    VALUES (value1[, value2, value3, ...])
```

The `INSERT` query in Example 7-12 adds a new dish to the `dishes` table.

Example 7-12. Inserting a new dish

```
INSERT INTO dishes (dish_id, dish_name, price, is_spicy)
    VALUES (1, 'Braised Sea Cucumber', 6.50, 0)
```

String values such as `Braised Sea Cucumber` have to have single quotes around them when used in an SQL query. Because single quotes are used as string delimiters, you need to escape single quotes with a backslash when they appear inside of a query. Example 7-13 shows how to insert a dish named `General Tso's Chicken` into the `dishes` table.

Example 7-13. Quoting a string value

```
INSERT INTO dishes (dish_id, dish_name, price, is_spicy)
    VALUES (2, 'General Tso\'s Chicken', 6.75, 1)
```

The number of columns enumerated in the parentheses before `VALUES` must match the number of values in the parentheses after `VALUES`. To insert a row that contains values only for some columns, just specify those columns and their corresponding values, as shown in Example 7-14.

Example 7-14. Inserting without all columns

```
INSERT INTO dishes (dish_name, is_spicy)
    VALUES ('Salt Baked Scallops', 0)
```

As a shortcut, you can eliminate the column list when you're inserting values for all columns. Example 7-15 performs the same `INSERT` as Example 7-12.

Example 7-15. Inserting with values for all columns

```
INSERT INTO dishes
    VALUES (1, 'Braised Sea Cucumber', 6.50, 0)
```

Example 7-10 prints:

```
DB Error: syntax error
```

Because the program quits when it encounters the query error, the last line of Example 7-10 never runs or prints its `Query Succeeded!` message.

The `setErrorHandling( )` function belongs to the `$db` object, so you have to get a `$db` object by calling `DB::connect( )` before you can call `setErrorHandling( )`. Therefore, one call to `DB::isError( )` is still necessary in your program to see whether the connection succeeded. Once that's taken care of, however, you can call `setErrorHandling( )` and not scatter the rest of your program with `DB::isError( )` calls. Section 12.4 explains how to have `setErrorHandling( )` print out a customized message when there is a database error.

Use the `query( )` function to change data with `UPDATE` data as well. Example 7-16 shows some `UPDATE` statements.

Example 7-16. Changing data with query( )

```
require 'DB.php';
$db = DB::connect('mysql://hunter:w)mp3s@db.example.com/restaurant');
if (DB::isError($db)) { die("connection error: " . $db->getMessage( )); }
// Eggplant with Chili Sauce is spicy
$db->query("UPDATE dishes SET is_spicy = 1
            WHERE dish_name = 'Eggplant with Chili Sauce'");
// Lobster with Chili Sauce is spicy and pricy
$db->query("UPDATE dishes SET is_spicy = 1, price=price * 2
            WHERE dish_name = 'Lobster with Chili Sauce'");
```

Also use the `query( )` function to delete data with `DELETE`. Example 7-17 shows `query( )` with two `DELETE` statements.

Example 7-17. Deleting data with query( )

```
require 'DB.php';
$db = DB::connect('mysql://hunter:w)mp3s@db.example.com/restaurant');
if (DB::isError($db)) { die("connection error: " . $db->getMessage( )); }
// remove expensive dishes
if ($make_things_cheaper) {
    $db->query("DELETE FROM dishes WHERE price > 19.95");
} else {
    // or, remove all dishes
    $db->query("DELETE FROM dishes");
}
```

SQL Lesson: *UPDATE*

The `UPDATE` command changes data already in a table. Example 7-18 shows the syntax of `UPDATE`.

Example 7-18. Updating data

```
UPDATE tablename SET column1=value1[, column2=value2,
    column3=value3, ...] [WHERE where_clause]
```

The value that a column is changed to can be a string or number, as shown in Example 7-19. The lines in Example 7-19 that begin with `;` are SQL comments.

Example 7-19. Setting a column to a string or number

```
; Change price to 5.50 in all rows of the table
UPDATE dishes SET price = 5.50

; Change is_spicy to 1 in all rows of the table
UPDATE dishes SET is_spicy = 1
```

The value can also be an expression that includes column names. The query in Example 7-20 doubles the price of each dish.

Example 7-20. Using a column name in an UPDATE expression

```
UPDATE dishes SET price = price * 2
```

The `UPDATE` queries shown so far each change all rows in the `dishes` table. To just change some rows with an `UPDATE` query, add a `WHERE` clause. This is a logical expression that describes which rows you want to change. The changes in the `UPDATE` query then happen only in rows that match the `WHERE` clause. Example 7-21 contains two `UPDATE` queries, each with a `WHERE` clause.

Example 7-21. Using a WHERE clause with UPDATE

```
; Change the spicy status of Eggplant with Chili Sauce
UPDATE dishes SET is_spicy = 1
              WHERE dish_name = 'Eggplant with Chili Sauce'

; Decrease the price of General Tso's Chicken
UPDATE dishes SET price = price - 1
              WHERE dish_name = 'General Tso\'s Chicken'
```

The `WHERE` clause is explained in more detail in the sidebar *SQL Lesson:* SELECT.

The `affectedRows( )` function tells you how many rows were changed or removed by an `UPDATE` or `DELETE` statement. Call `affectedRows( )` immediately after a query to find out how many rows that query affected. Example 7-22 reports how many rows have had their prices changed by an `UPDATE` query.

Example 7-22. Finding how many rows an UPDATE or DELETE affects

```
require 'DB.php';
$db = DB::connect('mysql://hunter:w)mp3s@db.example.com/restaurant');
if (DB::isError($db)) { die("connection error: " . $db->getMessage( )); }
// Decrease the price some some dishes
$db->query("UPDATE dishes SET price=price - 5 WHERE price > 20");
print 'Changed the price of ' . $db->affectedRows( ) . 'rows.';
```

If there are five rows in the `dishes` table whose price is more than 20, then Example 7-22 prints:

```
Changed the price of 5 rows.
```

SQL Lesson: *DELETE*

The `DELETE` command removes rows from a table. Example 7-23 shows the syntax of `DELETE`.

Example 7-23. Removing rows from a table

```
DELETE FROM tablename [WHERE where_clause]
```

Without a `WHERE` clause, `DELETE` removes all the rows from the table. Example 7-24 clears out the `dishes` table.

Example 7-24. Removing all rows from a table

```
DELETE FROM dishes
```

With a `WHERE` clause, `DELETE` removes the rows that match the `WHERE` clause. Example 7-25 shows two `DELETE` queries with `WHERE` clauses.

Example 7-25. Removing some rows from a table

```
; Delete rows in which price is greater than 10.00
DELETE FROM dishes WHERE price > 10.00

; Delete rows in which dish_name is exactly "Walnut Bun"
DELETE FROM dishes WHERE dish_name = 'Walnut Bun'
```

There is no SQL `UNDELETE` command, so be careful with your `DELETE`s.

7.5 Inserting Form Data Safely

As Section 6.4.6 explained, printing unsanitized form data can leave you and your users vulnerable to a cross-site scripting attack. Using unsanitized form data in SQL queries can cause a similar problem, called an "SQL injection attack." Consider a form that lets a user suggest a new dish. The form contains a text element called `new_dish_name` into which the user can type the name of their new dish. The call to `query( )` in Example 7-26 inserts the new dish into the `dishes` table but is vulnerable to an SQL injection attack.

Example 7-26. Unsafe insertion of form data

```
$db->query("INSERT INTO dishes (dish_name)
            VALUES ('$_POST[new_dish_name]')");
```

If the submitted value for `new_dish_name` is reasonable, such as `Fried Bean Curd`, then the query succeeds. PHP's regular double-quoted string interpolation rules make the query `INSERT INTO dishes (dish_name) VALUES ('Fried Bean Curd')`, which is valid and respectable. A query with an apostrophe in it causes a problem, though. If the submitted value for `new_dish_name` is `General Tso's Chicken`, then the query becomes `INSERT INTO dishes (dish_name) VALUES ('General Tso's Chicken')`. This makes the database program confused. It thinks that the apostrophe between `Tso` and `s` ends the string, so the `s Chicken`' after the second single quote is an unwanted syntax error.

What's worse, a user that really wants to cause problems can type in specially constructed input to wreak havoc. Consider this unappetizing input:

```
x'); DELETE FROM dishes; INSERT INTO dishes (dish_name) VALUES ('y.
```

When that gets interpolated, the query becomes:

```
INSERT INTO DISHES (dish_name) VALUES ('x'); DELETE FROM dishes; INSERT INTO dishes
(dish_name) VALUES ('y')
```

Some databases let you pass multiple queries separated by semicolons in one call of `query( )`. On those databases, the `dishes` table is demolished: a dish named `x` is inserted, all dishes are deleted, and a dish named `y` is inserted.

By submitting a carefully built form input value, a malicious user is able to inject arbitrary SQL statements into your database program. To prevent this, you need to escape special characters (most importantly, the apostrophe) in SQL queries. PEAR DB provides a helpful feature called *placeholders* that makes this a snap.

PHP has an unfortunate feature called "Magic Quotes." If this is turned on, submitted form data has quotes and backslashes escaped before it is put into `$_GET` or `$_POST`. If someone submits a form with `Sauteed Pig's Stomach` typed into the a text field named `entree`, then `$_POST['entree']` is not `Sauteed Pig's Stomach`, but `Sauteed Pig\'s Stomach` instead. This is conceivably handy if all you're going to do with `$_POST['entree']` is use it in a database query, but it is very inconvenient if you want to use `$_POST['entree']` in other contexts (such as simply printing it)

where the extra backslash is not welcome.

The "Magic Quotes" feature is enabled when the PHP configuration directive `magic_quotes_gpc` is turned on. For increased efficiency and more straightforward handling of submitted form parameters, turn `magic_quotes_gpc` off and use placeholders or a quoting function when you need to prepare external input for use in a database query.

To use a placeholder in a query, put a `?` in the query in each place where you want a value to go. Then, pass `query( )` a second argument—an array of values to be substituted for the placeholders. The values are appropriately quoted before they are put into the query, protecting you from any SQL injection attacks. Example 7-27 shows the safe version of the query from Example 7-26.

Example 7-27. Safe insertion of form data

```
$db->query('INSERT INTO dishes (dish_name) VALUES (?)',
    array($_POST['new_dish_name']));
```

You don't need to put quotes around the placeholder in the query. DB takes care of that for you too. If you want to use multiple values in a query, put multiple placeholders in the query and in the value array. Example 7-28 shows a query with three placeholders.

Example 7-28. Using multiple placeholders

```
$db->query('INSERT INTO dishes (dish_name,price,is_spicy) VALUES (?,?,?)',
           array($_POST['new_dish_name'], $_POST['new_price'],
                 $_POST['is_spicy']));
```

7.6 Generating Unique IDs

As mentioned in Section 7.1, rows in a database table don't have any inherent order. In a spreadsheet, you can refer particular records such as "the first row" or "the last row" or "rows 15 to 22." A database table is different. If you want to be able to specifically identify individual records, you need to give them each a unique identifier.

To uniquely identify individual rows in a table, make a column in the table that holds an integer ID and store a different number in that column for each row. That way, even if two rows have identical values in all the other columns, you can tell them apart by using the ID column. With a `dish_id` column in the `dishes` table, you can tell apart two dishes each called "Fried Bean Curd" because the rows have different `dish_id` values.

PEAR DB helps you generate unique integer IDs with its support for *sequences*. When you ask for the next ID in a particular sequence, you get a number that you know isn't duplicated in that sequence. Even if two simultaneously executing PHP scripts ask for the next ID in a sequence at the exact same time, they each get a different ID to use.

You can have as many independent sequences as you want. To get the next value from a sequence, call the `nextID( )` function. Example 7-29 gets an ID from the `dishes` sequence and then uses it to `INSERT` a row into the `dishes` table.

Example 7-29. Getting an ID from a sequence

```
$dish_id = $db->nextID('dishes');
$db->query("INSERT INTO orders (dish_id, dish_name, price, is_spicy)
    VALUES ($dish_id, 'Fried Bean Curd', 1.50, 0)");
```

7.7 A Complete Data Insertion Form

Example 7-30 combines the database topics covered so far in this chapter with the form-handling code from Chapter 6 to build a complete program that displays a form, validates the submitted data, and then saves the data into a database table. The form displays input elements for the name of a dish, the price of a dish, and whether the dish is spicy. The information is inserted into the `dishes` table.

The code in Example 7-30 relies on the form helper functions defined in Example 6-29. Instead of repeating them in this example, the code assumes they have been saved into a file called *formhelpers.php* and then loads them with the `require 'formhelpers.php'` line at the top of the program.

Example 7-30. Form for inserting records into dishes

```
<?php
// Load PEAR DB
require 'DB.php';
// Load the form helper functions
require 'formhelpers.php';

// Connect to the database
$db = DB::connect('mysql://hunter:w)mp3s@db.example.com/restaurant');
if (DB::isError($db)) { die ("Can't connect: " . $db->getMessage( )); }
// Set up automatic error handling
$db->setErrorHandling(PEAR_ERROR_DIE);

// The main page logic:
// - If the form is submitted, validate and then process or redisplay
// - If it's not submitted, display
if ($_POST['_submit_check']) {
    // If validate_form( ) returns errors, pass them to show_form( )
    if ($form_errors = validate_form( )) {
        show_form($form_errors);
    } else {
        // The submitted data is valid, so process it
        process_form( );
    }
} else {
    // The form wasn't submitted, so display
    show_form( );
}

function show_form($errors = '') {
    // If the form is submitted, get defaults from submitted parameters
    if ($_POST['_submit_check']) {
        $defaults = $_POST;
    } else {
        // Otherwise, set our own defaults: price is $5
```

```
        $defaults = array('price' => '5.00');
    }

    // If errors were passed in, put them in $error_text (with HTML markup)
    if ($errors) {
        $error_text = '<tr><td>You need to correct the following errors:';
        $error_text .= '</td><td><ul><li>';
        $error_text .= implode('</li><li>',$errors);
        $error_text .= '</li></ul></td></tr>';
    } else {
        // No errors? Then $error_text is blank
        $error_text = '';
    }

    // Jump out of PHP mode to make displaying all the HTML tags easier
?>
<form method="POST" action="<?php print $_SERVER['PHP_SELF']; ?>">
<table>
<?php print $error_text ?>

<tr><td>Dish Name:</td>
<td><?php input_text('dish_name', $defaults); ?></td></tr>

<tr><td>Price:</td>
<td><?php input_text('price', $defaults); ?></td></tr>

<tr><td>Spicy:</td>
<td><?php input_radiocheck('checkbox','is_spicy', $defaults, 'yes'); ?>
 Yes</td></tr>

<tr><td colspan="2" align="center"><?php input_submit('save','Order'); ?>
</td></tr>

</table>
<input type="hidden" name="_submit_check" value="1"/>
</form>
<?php
      } // The end of show_form( )

function validate_form( ) {
    $errors = array( );

    // dish_name is required
    if (! strlen(trim($_POST['dish_name']))) {
        $errors[  ] = 'Please enter the name of the dish.';
    }

    // price must be a valid floating point number and
    // more than 0
    if (floatval($_POST['price']) <= 0) {
        $errors[  ] = 'Please enter a valid price.';
    }

    return $errors;
}

function process_form( ) {
```

```
    // Access the global variable $db inside this function
    global $db;

    // Get a unique ID for this dish
    $dish_id = $db->nextID('dishes');

    // Set the value of $is_spicy based on the checkbox
    if ($_POST['is_spicy'] =  = 'yes') {
        $is_spicy = 1;
    } else {
        $is_spicy = 0;
    }

    // Insert the new dish into the table
    $db->query('INSERT INTO dishes (dish_id, dish_name, price, is_spicy)
                VALUES (?,?,?,?)',
               array($dish_id, $_POST['dish_name'], $_POST['price'],
                     $is_spicy));

    // Tell the user that we added a dish.
    print 'Added ' . htmlentities($_POST['dish_name']) .
          ' to the database.';
}

?>
```

Example 7-30 has the same basic structure as the form examples from Chapter 6: functions for displaying, validating, and processing the form with some global logic that determines which function to call. The two new pieces are the global code that sets up the database connection and the database-related activities in `process_form( )`.

The database setup code comes after the `require` statements and before the `if($_POST['_submit_check'])`. The `DB::connect( )` function establishes a database connection, and the next three lines check whether the connection succeeded and turn on automatic error handling for the rest of the program.

All of the interaction with the database is in the `process_form( )` function. First, the `global $db` line lets you refer to the database connection variable inside the function as `$db` instead of the clumsier `$GLOBALS['db']`. Then, `nextId( )` gets a unique integer ID for the new dish about to be saved. The `is_spicy` column of the table holds a 1 in the rows of spicy dishes and a 0 in nonspicy dishes, so the `if( )` clause in `process_form( )` assigns the appropriate value to the local variable `$is_spicy` based on what was submitted in `$_POST['is_spicy']`.

After that comes the call to `query( )` that actually puts the new information into the database. The `INSERT` statement has four placeholders that are filled by the variables `$dish_id`, `$_POST['dish_name']`, `$_POST['price']`, and `$is_spicy`. Last, `process_form( )` prints a message telling the user that the dish was inserted. The `htmlentities( )` function protects against any HTML tags or JavaScript in the dish name.

7.8 Retrieving Data from the Database

The `query( )` function can also be used to retrieve information from the database. The syntax of `query( )` is the same, but what you do with the object that `query( )` returns is new. When it successfully completes a `SELECT` statement, `query(`

) returns an object that provides access to the retrieved rows. Each time you call the fetchRow( ) function of this object, you get the next row returned from the query. When there are no more rows left, fetchRow( ) returns a false value, making it perfect to use in a while( ) loop. This is shown in Example 7-31.

Example 7-31. Retrieving rows with query( ) and fetchRow( )

```
require 'DB.php';
$db = DB::connect('mysql://hunter:w)mp3s@db.example.com/restaurant');
$q = $db->query('SELECT dish_name, price FROM dishes');
while ($row = $q->fetchRow( )) {
    print "$row[0], $row[1] \n";
}
```

Example 7-31 prints:

```
Walnut Bun, 1.00
Cashew Nuts and White Mushrooms, 4.95
Dried Mulberries, 3.00
Eggplant with Chili Sauce, 6.50
```

The first time through the while( ) loop, fetchRow( ) returns an array containing Walnut Bun and 1.00. This array is assigned to $row. Since an array with elements in it evaluates to true, the code inside the while( ) loop executes, printing the data from the first row returned by the SELECT query. This happens three more times. On each trip through the while( ) loop, fetchRow( ) returns the next row in the set of rows returned by the SELECT query. When it has no more rows to return, fetchRow( ) returns a value that evaluates to false, and the while( ) loop is done.

To find out the number of rows returned by a SELECT query (without iterating through them all), use the numrows( ) function of the object returned by query( ). Example 7-32 reports how many rows are in the dishes table.

Example 7-32. Counting rows with numrows( )

```
require 'DB.php';
$db = DB::connect('mysql://hunter:w)mp3s@db.example.com/restaurant');
$q = $db->query('SELECT dish_name, price FROM dishes');
print 'There are ' . $q->numrows( ) . ' rows in the dishes table.';
```

With four rows in the table, Example 7-32 prints:

```
There are 5 rows in the dishes table.
```

Because sending a SELECT query to the database program and retrieving the results is such a common task, DB provides ways that collapse the call to query( ) and multiple calls to fetchRow( ) into one step. The getAll( ) function executes a SELECT query and returns an array containing all the retrieved rows. Example 7-33 uses getAll( ) to do the same thing as Example 7-31.

Example 7-33. Retrieving rows with getAll( )

```
require 'DB.php';
$db = DB::connect('mysql://hunter:w)mp3s@db.example.com/restaurant');
$rows = $db->getAll('SELECT dish_name, price FROM dishes');
foreach ($rows as $row) {
    print "$row[0], $row[1] \n";
}
```

Example 7-33 prints:

```
Walnut Bun, 1.00
Cashew Nuts and White Mushrooms, 4.95
Dried Mulberries, 3.00
Eggplant with Chili Sauce, 6.50
```

SQL Lesson: *SELECT*

The `SELECT` command retrieves data from the database. Example 7-34 shows the syntax of `SELECT`.

Example 7-34. Retrieving data

```
SELECT column1[, column2, column3, ...] FROM tablename
```

The `SELECT` query in Example 7-35 retrieves the `dish_name` and `price` columns for all the rows in the `dishes` table.

Example 7-35. Retrieving dish_name and price

```
SELECT dish_name, price FROM dishes
```

As a shortcut, you can use `*` instead of a list of columns. This retrieves all columns from the table. The `SELECT` query in Example 7-36 retrieves everything from the `dishes` table.

*Example 7-36. Using * in a SELECT query*

```
SELECT * FROM dishes
```

To restrict a `SELECT` statement so that it matches only certain rows, add a `WHERE` clause to it. Only rows that meet the tests listed in the `WHERE` clause are returned by the `SELECT` statement. The `WHERE` clause goes after the table name, as shown in Example 7-37.

Example 7-37. Restricting the rows returned by SELECT

```
SELECT column1[, column2, column3, ...] FROM tablename
       WHERE where_clause
```

The `where_clause` part of the query is a logical expression that describes which rows you want to retrieve. Example 7-38 shows some `SELECT` queries with `WHERE` clauses.

Example 7-38. Retrieving certain dishes

```
; Dishes with price greater than 5.00
SELECT dish_name, price FROM dishes WHERE price > 5.00

; Dishes whose name exactly matches "Walnut Bun"
SELECT price FROM dishes WHERE dish_name = 'Walnut Bun'

; Dishes with price more than 5.00 but less than or equal to 10.00
SELECT dish_name FROM dishes WHERE price > 5.00 AND price <= 10.00

; Dishes with price more than 5.00 but less than or equal to 10.00,
```

```
; or dishes whose name exactly matches "Walnut Bun" (at any price)
SELECT dish_name, price FROM dishes WHERE (price > 5.00 AND price <= 10.00)
       OR dish_name = 'Walnut Bun'
```

Table 7-3 lists some operators that you can use in a `WHERE` clause.

Table 7-3. SQL WHERE clause operators

Operator	Description
`=`	Equal to (like `==` in PHP)
`<>`	Not equal to (like `!=` in PHP)
`>`	Greater than
`<`	Less than
`>=`	Greater than or equal to
`<=`	Less than or equal to
`AND`	Logical AND (like `&&` in PHP)
`OR`	Logical OR (like `||` in PHP)
`( )`	Grouping

When you are only expecting one row to be returned from a query, use `getRow( )`. It executes a `SELECT` query and returns the values for just one row. Example 7-39 uses `getRow( )` to display the least expensive item in the dishes table. The `ORDER BY` and `LIMIT` parts of the query in Example 7-39 are explained in the sidebar *SQL Lesson:* ORDER BY and LIMIT.

Example 7-39. Retrieving a row with getRow( )

```
require 'DB.php';
$db = DB::connect('mysql://hunter:w)mp3s@db.example.com/restaurant');
$cheapest_dish_info = $db->getRow('SELECT dish_name, price
                                   FROM dishes ORDER BY price LIMIT 1');
print "$cheapest_dish_info[0], $cheapest_dish_info[1]";
```

Example 7-39 prints:

```
Walnut Bun, 1.00
```

When you want only one column from one row, use `getOne( )`. It executes a `SELECT` query and returns a single value: the first column from the first row returned. Example 7-40 uses `getOne( )` to find the name of the least expensive dish.

Example 7-40. Retrieving a value with getOne( )

```
require 'DB.php';
$db = DB::connect('mysql://hunter:w)mp3s@db.example.com/restaurant');
$cheapest_dish = $db->getOne('SELECT dish_name, price
                              FROM dishes ORDER BY price LIMIT 1');
print "The cheapest dish is $cheapest_dish";
```

Example 7-40 prints:

```
The cheapest dish is Walnut Bun
```

SQL Lesson: *ORDER BY and LIMIT*

As mentioned earlier in this chapter in Section 7.1, rows in a table don't have any inherent order. A database server doesn't have to return rows from a `SELECT` query in any particular pattern. To force a certain order on the returned rows, add an `ORDER BY` clause to your `SELECT`. Example 7-41 returns all the rows in the `dishes` table ordered by price, lowest to highest.

Example 7-41. Ordering rows returned from a SELECT query

```
SELECT dish_name FROM dishes ORDER BY price
```

To order from highest to lowest value, add `DESC` after the column that the results are ordered by. Example 7-42 returns all the rows in the `dishes` table ordered by price, highest to lowest.

Example 7-42. Ordering from highest to lowest

```
SELECT dish_name FROM dishes ORDER BY price DESC
```

You can specify multiple columns to order by. If two rows have the same value for the first `ORDER BY` column, they are sorted by the second. The query in Example 7-43 orders rows in `dishes` by price (highest to lowest). If multiple rows have the same price, then they are ordered alphabetically by name.

Example 7-43. Ordering by multiple columns

```
SELECT dish_name FROM dishes ORDER BY price DESC, dish_name
```

Using `ORDER BY` doesn't change the order of the rows in the table itself (remember, they don't really have any set order) but rearranges the results of the query. This affects only the answer to the query. If you hand someone a menu and ask them to read you the appetizers in alphabetical order, it doesn't affect the printed menu—just the response to your query ("Read me all the appetizers in alphabetical order").

Normally, a `SELECT` query returns all rows that match the `WHERE` clause (or all rows in a table if there is no `WHERE` clause). Sometimes it's helpful to just get a certain number of rows back. You may want to find the lowest priced dish available or just print 10 search results. To restrict the results to a specific number of rows, add a `LIMIT` clause to the end of the query. Example 7-44 returns the row from `dishes` with the lowest price.

Example 7-44. Limiting the number of rows returned by SELECT

```
SELECT * FROM dishes ORDER BY price LIMIT 1
```

Example 7-45 returns the first (sorted alphabetically by dish name) 10 rows from `dishes`.

Example 7-45. Still limiting the number of rows returned by SELECT

```
SELECT dish_name, price FROM dishes ORDER BY dish_name LIMIT 10
```

In general, you should only use `LIMIT` in a query that also has `ORDER BY`. If you leave out `ORDER BY`, the database program can return rows in any order. So, the "first" row one time a query is executed might not be the "first" row another time the same query is executed.

7.9 Changing the Format of Retrieved Rows

So far, `fetchRow( )`, `getAll( )`, and `getOne( )` have been returning rows from the database as numerically indexed arrays. This makes for concise and easy interpolation of values in double-quoted strings—but trying to remember, for example, which column from the `SELECT` query corresponds to element 6 in the result array can be difficult and error-prone. PEAR DB lets you specify that you'd prefer to have each result row delivered as either an array with string keys or as an object.

The *fetch mode* controls how result rows are formatted. The `setFetchMode( )` function changes the fetch mode. Any queries in a page after you call `setFetchMode( )` have their result rows formatted as specified by the argument to `setFetchMode( )`.

To get result rows as arrays with string keys, pass `DB_FETCHMODE_ASSOC` to `setFetchMode( )`. Note that `DB_FETCHMODE_ASSOC` is a special constant defined by PEAR DB, not a string, so you shouldn't put quotes around it. The array keys in the result row arrays correspond to column names. Example 7-46 shows how to use `fetchRow( )`, `getAll( )`, and `getRow( )` with string-keyed result rows.

Example 7-46. Retrieving rows as string-keyed arrays

```
require 'DB.php';
$db = DB::connect('mysql://hunter:w)mp3s@db.example.com/restaurant');

// Change the fetch mode to string-keyed arrays
$db->setFetchMode(DB_FETCHMODE_ASSOC);

print "With query( ) and fetchRow( ): \n";
// get each row with query( ) and fetchRow( );
$q = $db->query("SELECT dish_name, price FROM dishes");
while($row = $q->fetchRow( )) {
    print "The price of $row[dish_name] is $row[price] \n";
}

print "With getAll( ): \n";
// get all the rows with getAll( );
$dishes = $db->getAll('SELECT dish_name, price FROM dishes');
foreach ($dishes as $dish) {
    print "The price of $dish[dish_name] is $dish[price] \n";
}

print "With getRow( ): \n";
$cheap = $db->getRow('SELECT dish_name, price FROM dishes
    ORDER BY price LIMIT 1');
```

```
print "The cheapest dish is $cheap[dish_name] with price $cheap[price]";
```

Example 7-46 prints:

```
With query( ) and fetchRow( ):
The price of Walnut Bun is 1.00
The price of Cashew Nuts and White Mushrooms is 4.95
The price of Dried Mulberries is 3.00
The price of Eggplant with Chili Sauce is 6.50
With getAll( ):
The price of Walnut Bun is 1.00
The price of Cashew Nuts and White Mushrooms is 4.95
The price of Dried Mulberries is 3.00
The price of Eggplant with Chili Sauce is 6.50
With getRow( ):
The cheapest dish is Walnut Bun with price 1.00
```

In Example 7-46, `fetchRow( )`, `getAll( )`, and `getRow( )` operate almost identically as they have before: you give them an SQL query, and you get back some results. The difference is in those results. The rows that come back from these functions have string keys whose names are the names of columns in the database table.

To get result rows as objects, pass the `DB_FETCHMODE_OBJECT` constant to `setFetchMode( )`. Each result row is an object with values inside it whose names correspond to column names (such as the string array keys when the fetch mode is `DB_FETCHMODE_ASSOC`). The `DB_FETCHMODE_OBJECT` fetch mode is handy because the syntax for referring to data inside an object is a little more concise and easier to interpolate in a string compared to an string-keyed array: write the object name, then `->`, and then the name of the piece of data you want. For example, `$dish->dish_name` refers to the piece of data named `dish_name` inside the `$dish` object. Example 7-47 retrieves rows as objects.

Example 7-47. Retrieving rows as objects

```
require 'DB.php';
$db = DB::connect('mysql://hunter:w)mp3s@db.example.com/restaurant');

// Change the fetch mode to objects
$db->setFetchMode(DB_FETCHMODE_OBJECT);

print "With query( ) and fetchRow( ): \n";
// get each row with query( ) and fetchRow( );
$q = $db->query("SELECT dish_name, price FROM dishes");
while($row = $q->fetchRow( )) {
    print "The price of $row->dish_name is $row->price \n";
}

print "With getAll( ): \n";
// get all the rows with getAll( );
$dishes = $db->getAll('SELECT dish_name, price FROM dishes');
foreach ($dishes as $dish) {
    print "The price of $dish->dish_name is $dish->price \n";
}

print "With getRow( ): \n";
```

```
$cheap = $db->getRow('SELECT dish_name, price FROM dishes
    ORDER BY price LIMIT 1');
print "The cheapest dish is $cheap->dish_name with price $cheap->price";
```

Example 7-47 prints the same output as Example 7-46.

7.10 Retrieving Form Data Safely

It's possible to use placeholders with `SELECT` statements just as you do with `INSERT`, `UPDATE`, or `DELETE` statements. The `getAll( )`, `getRow( )`, and `getOne( )` functions each accept a second argument of an array of values that are substituted for placeholders in a query.

However, when you use submitted form data or other external input in the `WHERE` clause of a `SELECT`, `UPDATE`, or `DELETE` statement, you must take extra care to ensure that any SQL wildcards are appropriately escaped. Consider a search form with a text element called `dish_search` into which the user can type a name of a dish he's looking for. The call to `getAll( )` in Example 7-48 uses placeholders guard against confounding single-quotes in the submitted value.

Example 7-48. Using a placeholder in a SELECT statement

```
$matches = $db->getAll('SELECT dish_name, price FROM dishes
                        WHERE dish_name LIKE ?',
                       array($_POST['dish_search']));
```

Whether `dish_search` is `Fried Bean Curd` or `General Tso's Chicken`, the placeholder interpolates the value into the query appropriately. However, what if `dish_search` is `%chicken%`? Then, the query becomes `SELECT dish_name, price FROM dishes WHERE dish_name LIKE '%chicken%'`. This matches all rows that contain the string `chicken`, not just rows in which `dish_name` is exactly `%chicken%`.

To prevent SQL wildcards in form data from taking effect in queries, you must forgo the comfort and ease of the placeholder and rely on two other functions:

SQL Lesson: *Wildcards*

Wildcards are useful for matching text inexactly, such as finding strings that end with `.edu` or that contain `@`. SQL has two wildcards. The underscore (_) matches one character and the percent sign (`%`) matches any number of characters (including zero characters). The wildcards are active inside strings used with the `LIKE` operator in a `WHERE` clause.

Example 7-49 shows two `SELECT` queries that use `LIKE` and wildcards.

Example 7-49. Using wildcards with SELECT

```
; Retrieve all rows in which dish name begins with D
SELECT * FROM dishes WHERE dish_name LIKE 'D%'

; Retrieve rows in which dish name is Fried Cod, Fried Bod,
; Fried Nod, and so on.
SELECT * FROM dishes WHERE dish_name LIKE 'Fried _od'
```

Wildcards are active in the `WHERE` clauses of `UPDATE` and `DELETE` statements, too. The query in Example 7-50 doubles the price of all dishes that have `chili` in their names.

Example 7-50. Using wildcards with UPDATE

```
UPDATE dishes SET price = price * 2 WHERE dish_name LIKE '%chili%'
```

The query in Example 7-51 deletes all rows whose `dish_name` ends with `Shrimp`.

Example 7-51. Using wildcards with DELETE

```
DELETE FROM dishes WHERE dish_name LIKE '%Shrimp'
```

To match against a literal `%` or _ when using the `LIKE` operator, put a backslash before the `%` or _. The query in Example 7-52 finds all rows whose `dish_name` contains `50% off`.

Example 7-52. Escaping wildcards

```
SELECT * FROM dishes WHERE dish_name LIKE '%50\% off%'
```

Without the backslash, the query in Example 7-52 would match rows whose `dish_name` contains `50` and then has a space and `off` somewhere later in the name, such as `Spicy 50 shrimp with shells off salad` or `Famous 500 offer duck`.

`quoteSmart( )` function in DB and PHP's built-in `strtr( )` function. First, call `quoteSmart( )` on the submitted value.[3] This does the same quoting operation that a the placeholder does. For example, it turns `General Tso's Chicken` into '`General Tso\'s Chicken`'. The next step is to use `strtr( )` to backslash-escape the SQL wildcards `%` and `_`. The quoted and wildcard-escaped value can then be used safely in a query.

[3] The `quoteSmart( )` function was introduced in DB 1.6.0. If you are using an earlier version of DB and get an error when trying to use `quoteSmart( )`, use `quote( )` instead.

Example 7-53 shows how to use `quoteSmart( )` and `strtr( )` to make a submitted value safe for a `WHERE` clause.

Example 7-53. Not using a placeholder in a SELECT statement

```
// First, do normal quoting of the value
$dish = $db->quoteSmart($_POST['dish_search']);
// Then, put backslashes before underscores and percent signs
$dish = strtr($dish, array('_' => '\_', '%' => '\%'));
// Now, $dish is sanitized and can be interpolated right into the query
$matches = $db->getAll("SELECT dish_name, price FROM dishes
                        WHERE dish_name LIKE $dish");
```

You can't use a placeholder in this situation because the escaping of the SQL wildcards has to happen after the regular quoting. The regular quoting puts a backslash before single quotes, but also before backslashes. If `strtr( )` processes the string first, a submitted value such as `%chicken%` becomes `\%chicken\%`. Then, the quoting (whether by `quoteSmart( )` or the placeholder processing) turns `\%chicken\%` into '`\\%chicken\\%`'. This is interpreted by the database to mean a literal backslash, followed by the "match any characters" wildcard, followed by `chicken`, followed by another literal backslash, followed by another "match any characters" wildcard. However, if `quoteSmart( )` goes first, `%chicken%` is turned into '`%chicken%`'. Then, `strtr( )` turns it into '`\%chicken\%`'. This is interpreted by the database as a literal percent sign, followed by `chicken`, followed by another percent sign, which is what the user entered.

Not quoting wildcard characters has an even more drastic effect in the `WHERE` clause of an `UPDATE` or `DELETE` statement. Example 7-54 shows a query incorrectly using placeholders to allow a user-entered value to control which dishes have their prices set to $1.

Example 7-54. Incorrect use of placeholders in an UPDATE statement

```
$db->query('UPDATE dishes SET price = 1 WHERE dish_name LIKE ?',
           array($_POST['dish_name']));
```

If the submitted value for `dish_name` in Example 7-54 is `Fried Bean Curd`, then the query works as expected: the price of that dish only is set to 1. But if `$_POST['dish_name']` is `%`, then all dishes have their price set to 1! The `quoteSmart( )` and `strtr( )` technique prevents this problem. The right way to do the update is in Example 7-55.

Example 7-55. Correct use of quoteSmart( ) and strtr( ) with an UPDATE statement

```
// First, do normal quoting of the value
$dish = $db->quoteSmart($_POST['dish_name']);
// Then, put backslashes before underscores and percent signs
$dish = strtr($dish, array('_' => '\_', '%' => '\%'));
// Now, $dish is sanitized and can be interpolated right into the query
```

```
$db->query("UPDATE dishes SET price = 1 WHERE dish_name LIKE $dish");
```

7.11 A Complete Data Retrieval Form

Example 7-56 is another complete database and form program. It presents a search form and then prints an HTML table of all rows in the `dishes` table that match the search criteria. Like Example 7-30, it relies on the form helper functions being defined in a separate *formhelpers.php* file.

Example 7-56. Form for searching the dishes table

```
<?php

// Load PEAR DB
require 'DB.php';
// Load the form helper functions.
require 'formhelpers.php';

// Connect to the database
$db = DB::connect('mysql://hunter:w)mp3s@db.example.com/restaurant');
if (DB::isError($db)) { die ("Can't connect: " . $db->getMessage( )); }

// Set up automatic error handling
$db->setErrorHandling(PEAR_ERROR_DIE);

// Set up fetch mode: rows as objects
$db->setFetchMode(DB_FETCHMODE_OBJECT);

// Choices for the "spicy" menu in the form
$spicy_choices = array('no','yes','either');

// The main page logic:
// - If the form is submitted, validate and then process or redisplay
// - If it's not submitted, display
if ($_POST['_submit_check']) {
    // If validate_form( ) returns errors, pass them to show_form( )
    if ($form_errors = validate_form( )) {
        show_form($form_errors);
    } else {
        // The submitted data is valid, so process it
        process_form( );
    }
} else {
    // The form wasn't submitted, so display
    show_form( );
}

function show_form($errors = '') {
    // If the form is submitted, get defaults from submitted parameters
    if ($_POST['_submit_check']) {
        $defaults = $_POST;
    } else {
        // Otherwise, set our own defaults
        $defaults = array('min_price' => '5.00',
```

```
                        'max_price' => '25.00');
    }

    // If errors were passed in, put them in $error_text (with HTML markup)
    if ($errors) {
        $error_text = '<tr><td>You need to correct the following errors:';
        $error_text .= '</td><td><ul><li>';
        $error_text .= implode('</li><li>',$errors);
        $error_text .= '</li></ul></td></tr>';
    } else {
        // No errors? Then $error_text is blank
        $error_text = '';
    }

    // Jump out of PHP mode to make displaying all the HTML tags easier
?>
<form method="POST" action="<?php print $_SERVER['PHP_SELF']; ?>">
<table>
<?php print $error_text ?>

<tr><td>Dish Name:</td>
<td><?php input_text('dish_name', $defaults) ?></td></tr>

<tr><td>Minimum Price:</td>
<td><?php input_text('min_price', $defaults) ?></td></tr>

<tr><td>Maximum Price:</td>
<td><?php input_text('max_price', $defaults) ?></td></tr>

<tr><td>Spicy:</td>
<td><?php input_select('is_spicy', $defaults, $GLOBALS['spicy_choices']); ?>
</td></tr>

<tr><td colspan="2" align="center"><?php input_submit('search','Search'); ?>
</td></tr>

</table>
<input type="hidden" name="_submit_check" value="1"/>
</form>
<?php
      } // The end of show_form( )

function validate_form( ) {
    $errors = array( );

    // minimum price must be a valid floating point number
    if ($_POST['min_price'] != strval(floatval($_POST['min_price']))) {
        $errors[  ] = 'Please enter a valid minimum price.';
    }

    // maximum price must be a valid floating point number
    if ($_POST['max_price'] != strval(floatval($_POST['max_price']))) {
        $errors[  ] = 'Please enter a valid maximum price.';
    }

    // minimum price must be less than the maximum price
    if ($_POST['min_price'] >= $_POST['max_price']) {
```

```
        $errors[  ] = 'The minimum price must be less than the maximum price.';
    }

    if (! array_key_exists($_POST['is_spicy'], $GLOBALS['spicy_choices'])) {
        $errors[  ] = 'Please choose a valid "spicy" option.';
    }
    return $errors;
}

function process_form( ) {
    // Access the global variable $db inside this function
    global $db;

    // build up the query
    $sql = 'SELECT dish_name, price, is_spicy FROM dishes WHERE
            price >= ? AND price <= ?';

    // if a dish name was submitted, add to the WHERE clause
    // we use quoteSmart( ) and strtr( ) to prevent user-entered wildcards from working
    if (strlen(trim($_POST['dish_name']))) {
        $dish = $db->quoteSmart($_POST['dish_name']);
        $dish = strtr($dish, array('_' => '\_', '%' => '\%'));
        $sql .= " AND dish_name LIKE $dish";
    }

    // if is_spicy is "yes" or "no", add appropriate SQL
    // (if it's "either", we don't need to add is_spicy to the WHERE clause)
    $spicy_choice = $GLOBALS['spicy_choices'][ $_POST['is_spicy'] ];
    if ($spicy_choice =  = 'yes') {
        $sql .= ' AND is_spicy = 1';
    } elseif ($spicy_choice =  = 'no') {
        $sql .= ' AND is_spicy = 0';
    }

    // Send the query to the database program and get all the rows back
    $dishes = $db->getAll($sql, array($_POST['min_price'],
                                      $_POST['max_price']));

    if (count($dishes) =  = 0) {
        print 'No dishes matched.';
    } else {
        print '<table>';
        print '<tr><th>Dish Name</th><th>Price</th><th>Spicy?</th></tr>';
        foreach ($dishes as $dish) {
            if ($dish->is_spicy =  = 1) {
                $spicy = 'Yes';
            } else {
                $spicy = 'No';
            }
            printf('<tr><td>%s</td><td>$%.02f</td><td>%s</td></tr>',
                   htmlentities($dish->dish_name), $dish->price, $spicy);
        }
    }
}
?>
```

Example 7-56 is a lot like Example 7-30: the standard display/validate/process form structure with global code for database setup and database interaction inside `process_form( )`. There are a few differences, however.

Example 7-56 has an additional line in its database setup code: a call to `setFetchMode( )`. Since `process_form( )` is going to retrieve information from the database, the fetch mode is important.

The `process_form( )` function builds up a `SELECT` statement, sends it to the database with `getAll( )`, and prints the results in an HTML table. Up to four factors go into the `WHERE` clause of the `SELECT` statement. The first two are the minimum and maximum price. These are always in the query, so they get placeholders in `$sql`, the variable that holds the SQL statement.

Next comes the dish name. That's optional, but if it's submitted, it goes into the query. A placeholder isn't good enough for the `dish_name` column, though, because the submitted form data could contain SQL wildcards. Instead, `quoteSmart( )` and `strtr( )` prepare a sanitized version of the dish name, and it's added directly onto the `WHERE` clause.

The last possible column in the `WHERE` clause is `is_spicy`. If the submitted choice is `yes`, then `AND is_spicy = 1` goes into the query so that only spicy dishes are retrieved. If the submitted choice is `no`, then `AND is_spicy = 0` goes into the query so that only nonspicy dishes are found. If the submitted choice is `either`, then there's no need to have `is_spicy` in the query—rows should be picked regardless of their spiciness.

After the full query is constructed in `$sql`, it's sent to the database program with `getAll( )`. The second argument to `getAll( )` is an array containing the minimum and maximum price values so that they can be substituted for the placeholders. The array of rows that `getAll( )` returns is stored in `$dishes`.

The last step in `process_form( )` is printing some results. If there's nothing in `$dishes`, `No dishes matched` is displayed. Otherwise, a `foreach( )` loop iterates through dishes and prints out an HTML table row for each dish, using `printf( )` to format the price properly and `htmlentities( )` to encode any special characters in the dish name. An `if( )` clause turns the database-friendly `is_spicy` values of `1` or `0` to the human-friendly values of `Yes` or `No`.

7.12 MySQL Without PEAR DB

PEAR DB smooths over a lot of the rough edges of database access in a PHP program, but there are two reasons why it's not always the right choice: PEAR DB might not be available on some systems, and a program that uses the built-in PHP functions tailored to a particular database is faster than one that uses PEAR DB. Programmers who don't anticipate switching or using more than one database program often pick those built-in functions.

The basic model of database access with the built-in functions is the same as with PEAR DB. You call a function that connects to the database. It returns a variable that represents the connection. You use that connection variable with other functions to send queries to the database program and retrieve the results.

The differences are in the details. The applicable functions and how they work differ from database to database. In general, you have to retrieve results one row at a time instead of the convenience that `getAll( )` offers, and there is no unified error handling.

As an example for database access without PEAR DB, this section discusses the mysqli extension, which works with MySQL 4.1.2 or greater and with PHP 5. There are similar PHP extensions for other database programs. Table 7-4 lists the database programs that PHP supports and where in the PHP Manual you can read about the functions in the extension for each database. All of the extensions listed in Table 7-4 are not usually installed by default with the PHP interpreter, but the PHP Manual gives instructions on how to install them.

Table 7-4. Database extensions

Database program	PHP Manual URL
Adabas D	http://www.php.net/uodbc
DB2	http://www.php.net/uodbc
DB++	http://www.php.net/dbplus
Empress	http://www.php.net/uodbc
FrontBase	http://www.php.net/fbsql
Informix	http://www.php.net/ifx
InterBase	http://www.php.net/ibase
Ingres II	http://www.php.net/ingres
Microsoft SQL Server	http://www.php.net/mssql
mSQL	http://www.php.net/msql
MySQL (Version 4.1.1 and earlier)	http://www.php.net/mysql
MySQL (Version 4.1.2 and later)	http://www.php.net/mysqli
ODBC	http://www.php.net/uodbc
Oracle	http://www.php.net/oci8
Ovrimos SQL	http://www.php.net/ovrimos
PostgreSQL	http://www.php.net/pgsql
SAP DB / MaxDB	http://www.php.net/uodbc
Solid	http://www.php.net/uodbc
SQLite	http://www.php.net/sqlite
Sybase	http://www.php.net/sybase

Table 7-5 shows the rough equivalencies between PEAR DB functions and mysqli functions.

Table 7-5. Comparing PEAR DB functions and mysqli functions

PEAR DB function	mysqli function	Comments
`$db = DB::connect(` *DSN*)	`$db = mysqli_connect(`*`hostname`*, *`username`*, *`password`*, *`database`*)	
`$q = $db->query(`*SQL*)	`$q = mysqli_query($db,`*SQL*)	There is no placeholder support in `mysqli_query( )`.
`$row = $q->fetchRow( )`	`$row = mysqli_fetch_row($q)`	`mysqli_fetch_row( )` always returns numerically indexed arrays. Use `mysqli_fetch_assoc( )` for string-indexed arrays or `mysqli_fetch_object( )` for objects.
`$db->affectedRows( )`	`mysqli_affected_rows($db)`	
`$q->numRows( )`	`mysqli_num_rows($q)`	
`$db->setErrorHandling(`*ERROR_MODE*)	None	You can't set automatic error handling with mysqli, but `mysqli_connect_error( )` gives you the error message if something goes wrong connecting to the database program, and `mysqli_error($db)` gives you the error message after a query or other function call fails.

This section doesn't explore the mysqli functions in great detail but shows how to use mysqli to do some of the things you've already seen with PEAR DB. Chapter 3 of *Upgrading to PHP 5*, by Adam Trachtenberg (O'Reilly) covers the ins and outs of mysqli, including advanced features such as secure connections, parameter binding, and result buffering. Examples Example 7-57 and Example 7-58 contain the necessary changes to Example 7-56 so that it uses PHP's mysqli extension instead of PEAR DB.

The two sections of the program that need to be changed are the top-level database connection code, which is shown in Example 7-57 and the `process_form( )` function, which is shown in Example 7-58.

Example 7-57. Connecting with mysqli

```
$db = mysqli_connect('db.example.com','hunter','w)mp3s','restaurant');
if (! $db) { die("Can't connect: " . mysqli_connect_error( )); }
```

The code in Example 7-57 replaces the two lines under the `// Connect to the database` comment in Example 7-56. The `mysqli_connect( )` function establishes the database connection, and the next line checks that the connection attempt succeeds.

Example 7-58. A process_form( ) function using mysqli

```
function process_form( ) {
    // Access the global variable $db inside this function
    global $db;

    // build up the query
    $sql = 'SELECT dish_name, price, is_spicy FROM dishes WHERE ';

    // add the minimum price to the query
    $sql .= "price >= '" .
            mysqli_real_escape_string($db, $_POST['min_price']) . "' ";

    // add the maximum price to the query
    $sql .= " AND price <= '" .
            mysqli_real_escape_string($db, $_POST['max_price']) . "' ";

    // if a dish name was submitted, add to the WHERE clause
    // we use mysqli_real_escape_string( ) and strtr( ) to prevent
    // user-entered wildcards from working
    if (strlen(trim($_POST['dish_name']))) {
        $dish = mysqli_real_escape_string($db, $_POST['dish_name']);
        $dish = strtr($dish, array('_' => '\_', '%' => '\%'));
        // mysqli_real_escape_string( ) doesn't add the single quotes
        // around the value so you have to put those around $dish in
        // the query:
        $sql .= " AND dish_name LIKE '$dish'";
    }

    // if is_spicy is "yes" or "no", add appropriate SQL
    // (if it's either, we don't need to add is_spicy to the WHERE clause)
    $spicy_choice = $GLOBALS['spicy_choices'][ $_POST['is_spicy'] ];
    if ($spicy_choice =  = 'yes') {
        $sql .= ' AND is_spicy = 1';
    } elseif ($spicy_choice =  = 'no') {
        $sql .= ' AND is_spicy = 0';
    }

    // Send the query to the database program and get all the rows back
    $q = mysqli_query($db, $sql);

    if (mysqli_num_rows($q) =  = 0) {
        print 'No dishes matched.';
    } else {
        print '<table>';
        print '<tr><th>Dish Name</th><th>Price</th><th>Spicy?</th></tr>';
        while ($dish = mysqli_fetch_object($q)) {
            if ($dish->is_spicy =  = 1) {
                $spicy = 'Yes';
            } else {
                $spicy = 'No';
            }
            printf('<tr><td>%s</td><td>$%.02f</td><td>%s</td></tr>',
                   htmlentities($dish->dish_name), $dish->price, $spicy);
        }
    }
}
```

The `process_form( )` function in Example 7-58 follows the same logical flow as that in Example 7-56, but the database interaction functions are different. Since PEAR DB's placeholders aren't available, the minimum and maximum prices are put directly into the `$sql` variable holding the query. First, however, they are escaped with `mysqli_real_escape_string( )`. Similarly, `$_POST['dish_name']` is escaped with `mysqli_real_escape_string( )`. Last, the functions used to pass the query to the database and retrieve the results are different. The `mysqli_query( )` function sends the query, `mysqli_num_rows( )` reports the number of rows returned, and `mysqli_fetch_object( )` retrieves each row in the result set as an object.

7.13 Chapter Summary

Chapter 7 covers:

- Figuring out what kinds of information belong in a database.
- Understanding how data is organized in a database.
- Loading an external file with `require`.
- Establishing a database connection.
- Creating a table in the database.
- Removing a table from the database.
- Using the SQL `INSERT` command.
- Inserting data into the database with `query( )`.
- Checking for database errors with `DB::isError( )`.
- Setting up automatic error handling with `setErrorHandling( )`.
- Using the SQL `UPDATE` and `DELETE` commands.
- Changing or deleting data with `query( )`.
- Counting the number of rows affected by a query.
- Using placeholders to insert data safely.
- Generating unique ID values with sequences.
- Using the SQL `SELECT` command.
- Retrieving data from the database with `query( )` and `fetchRow( )`.
- Counting the number of rows retrieved by `query( )`.
- Retrieving data with `getAll( )`, `getRow( )`, and `getOne( )`.
- Using the SQL `ORDER BY` and `LIMIT` keywords with `SELECT`.
- Retrieving rows as string-keyed arrays or objects.
- Using the SQL wildcards with `LIKE`: `%` and _.
- Escaping SQL wildcards in `SELECT` statements.
- Saving submitted form parameters in the database.
- Using data from the database in form elements.
- Using the mysqli functions instead of PEAR DB.

7.14 Exercises

The following exercises use a database table called `dishes` with the following structure:

```
CREATE TABLE dishes (
   dish_id     INT,
   dish_name   VARCHAR(255),
   price       DECIMAL(4,2),
   is_spicy    INT
)
```

Here is some sample data to put into the `dishes` table:

```
INSERT INTO dishes VALUES (1,'Walnut Bun',1.00,0)
INSERT INTO dishes VALUES (2,'Cashew Nuts and White Mushrooms',4.95,0)
INSERT INTO dishes VALUES (3,'Dried Mulberries',3.00,0)
INSERT INTO dishes VALUES (4,'Eggplant with Chili Sauce',6.50,1)
INSERT INTO dishes VALUES (5,'Red Bean Bun',1.00,0)
INSERT INTO dishes VALUES (6,'General Tso\'s Chicken',5.50,1)
```

1. Write a program that lists all of the dishes in the table, sorted by price.
2. Write a program that displays a form asking for a price. When the form is submitted, the program should print out the names and prices of the dishes whose price is at least the submitted price. Don't retrieve from the database any rows or columns that aren't printed in the table.
3. Write a program that displays a form with a `<select>` menu of dish names. Create the dish names to display by retrieving them from the database. When the form is submitted, the program should print out all of the information in the table (ID, name, price, and spiciness) for the selected dish.
4. Create a new table that holds information about restaurant customers. The table should store the following information about each customer: customer ID, name, phone number, and the ID of the customer's favorite dish. Write a program that displays a form for putting a new customer into the table. The part of the form for entering the customer's favorite dish should be a `<select>` menu of dish names. The customer's ID should be generated by your program, not entered in the form.