About Ultimate MySQL

Written by Jeff L. Williams (Freely Distributable)
http://www.phpclasses.org/ultimatemysql

- Establish MySQL server connections easily
- Execute any SQL query
- Retrieve query results into objects or arrays
- Easily retrieve the last inserted IDs
- Manage transactions (full transaction processing)
- Retrieve the list tables of a database
- Retrieve the list fields of a table (or field comments)
- Retrieve the length or data type of a field
- Measure the time a query takes to execute
- Display query results in an HTML table
- Easy formatting for SQL parameters and values
- Generate SQL Selects, Inserts, Updates, and Deletes
- Error handling with error numbers and text
- Full PHPdoc - so most editors will show helpful tooltips
- And much more!

This documentation assumes you have a basic working knowledge and setup of PHP and MySQL .

If you run Windows and do not have PHP or MySQL installed, check out EasyPHP which is a complete package that includes an Apache server, a MySQL database, PHP 5, and easy development tools for your web site.

If you need help with PHP, you can find a simple PHP tutorial on the PHP web site or check out Google.

Setting Up Auto-Connect

The Ultimate MySQL class can automatically connect to your database when included in your code. This saves a lot of time and code and is the recommended usage for this class.

In order to turn on this functionality, you must add your connection information to the top of the class file. Open up the mysql.class.php file in an editor and look for the following lines towards the top of the file:

```
class MySQL
{
   // SET THESE VALUES TO MATCH YOUR DATA CONNECTION
   private $db_host    = "localhost";  // server name
   private $db_user    = "root";       // user name
   private $db_pass    = "";           // password
   private $db_dbname  = "";           // database name
   private $db_charset = "";           // optional character set (i.e. utf8)
   private $db_pcon    = false;        // use persistent connection?
```

This is where you'll add your connection information.

- **$db_host** - This is the name of your server. If the database is running on the same machine as your web server, "localhost" or "127.0.0.1" is the value you will want to use and points back to the machine itself.
- **$db_user** - This is the user name you use to log into your database.
- **$db_pass** - This is the password for the user.
- **$db_dbname** - This is the database you will connect to. If you want to connect to more than one database, leave this blank. You can specify the name of the database when you create the MySQL object.
- **$db_charset** - This is the character set you wish to use. The character set "utf8" is unicode and is quickly becoming the standard in the database world. This is the recommended character set to use. If you wish to use the character set defined in your database, leave this blank.
- **$db_pcon** - This is a boolean value (true/false). If you set this to true, the MySQL class will use persistant connections. The default and most common setting is false.

Once you are finished with these settings, your text should look something like this:

```
class MySQL
{
  // SET THESE VALUES TO MATCH YOUR DATA CONNECTION
   private $db_host    = "localhost";  // server name
   private $db_user    = "root";       // user name
   private $db_pass    = "password";   // password
   private $db_dbname  = "salesdb";    // database name
   private $db_charset = "";           // optional character set (i.e. utf8)
   private $db_pcon    = false;        // use persistent connection?
```

Save the file and the next time you create your MySQL object, it will automatically connect for you (unless you specify false as the first parameter):

```
   $db = new MySQL();      // Connected
   $db = new MySQL(true);  // Connected
   $db = new MySQL(false); // NOT Connected
```

Include In Your Project

To use the Ultimate MySQL class, you must first include it in your project. There are multiple ways of doing this.

The recommended way of including the class file is by using:

```
include_once "mysql.class.php"
```

This will include the class in your project but will only include it once. If an include is made in another location, the class file will still only be included a single time.

The other way you may want to include this file (but not recommended since it bypasses error handling) is by using:

```
require_once("mysql.class.php");
```

The two constructs are identical in every way except how they handle failure. include_once() produces a warning and uses the MySQL class error handling while require_once() results in a Fatal Error. In other words, use require_once() if you want a missing file to halt processing of the page.

You may also use the following if you are sure that the file will not be included more than once:

```
include("mysql.class.php");
require("mysql.class.php");
```

For advanced users, there is even a cooler way to handle includes using the autoload function. The autoload function is called whenever you attempt to create a new object. Create a file called autoloader.php and place the following code into the file:

```php
<?php
  /**
  * Automatically includes files needed if objects are created
  *
  * @param string $classname
  */
  function __autoload($classname) {
    switch ($classname) {

      case "MySQL": // The name of the class (object)
        // Include the file this class is found in
        include_once("includes/mysql.class.php");
        break;


      default:
        // The class was not found
        throw new Exception("Class does not exist: " . $classname);
        break;
    }
  }
?>
```

Add your other classes and files to the switch statement and include this file instead:

```php
// This file auto-creates our objects for us and takes care of includes
include_once("autoloader.php");
```

Whenever you create an object listed in this file, it will automatically include the file for you! This keeps your code compact and makes it easier to use any objects without worrying about including various class files.

Security and the PHP.INI include_path Directive

To secure includes from malicious hacks, it's a good idea to set the include path in your php.ini file (as long as your host allows this). All class files can be stored in a single location (or a subfolder within the include path). If you move the mysql.class.php file into your includes folder, you simply include it without a path as such:

```
include_once("mysql.class.php");
```

If you update your class file, all code using this class on your system will be updated as well. This is good practice but it takes a little setting up.

Open your **php.ini** file in a text editor. On Windows machines, it is usually located in the Windows folder. Search for `include_path=` and you will see something similar to this:

```
;;;;;;;;;;;;;;;;;;;;;;;
; Paths and Directories ;
;;;;;;;;;;;;;;;;;;;;;;;
;include_path=""
```

Create a folder you wish to store your include files in and copy these files to this location (on Linux, you may need to `chmod 644`). Then set the include_path variable in your php.ini file to this folder.

On Windows it may look like this:

```
include_path=".;c:\php\includes"
```

On Linux, it may look more like this:

```
include_path='.:/usr/share/php'
```

Now, any file in these folder may be included globally in all projects.

For more information, see the PHP web site.

Connecting to the Database

In order to connect, you must first include the class file and create the MySQL object.

```
$db = new MySQL();
```

When creating the object, you may also specify the following optional parameters:

boolean	**$connect**	Connect now?
string	**$database**	Database name
string	**$server**	Host address
string	**$username**	User name
string	**$password**	Password
string	**$charset**	Character set
boolean	**$pcon**	Persistant connection?

So we might connect using:

```
$db = new MySQL(true, "mydb", "localhost", "root", "password");
```

If you prefer not to use the auto-connect feature, connect during object creation, or you wish to connect to a different data source, use the `Open()` method. It takes almost the identical optional parameters (the initial `$connect` parameter is not needed).

So to connect to a data source, use the following syntax:

```
$success = $db->Open("mydb", "localhost", "root", "password");
```

A true is returned if we are connected and a false is returned if the connection was unsuccessful.

What if you are using the auto-connect feature but you wish to designate the database to connect to? You can specify just the database during object creation:

```
$success = $db->Open("mydb");
```

What if you wish to change databases? You can use the `SelectDatabase()` method to

change databases after you are already connected:

```
$success = $db->SelectDatabase("mydb");
```

In order for this method to work, the database must have the same user name and password as the previous connection or the user name and password must be specified in the auto-connect. If not, you will have to specify this using the `Open()` method.

You can use the `IsConnected()` method to tell if you are connected or not. If you are connected, this method returns a true. Obviously, if you are not connected, it will return false.

```
if (MySQL->IsConnected()) {
    echo "connected";
} else {
    echo "not connected");
}
```

You do not need to close the connection. Once the object is out of context (i.e. the code has finished executing for a page), the connection is automatically closed for you.

Just in case you need to close the data connection for any reason, use the `Close()` method. It takes no parameters and returns true on success and false on failure:

```
$success = $db->Close();
```

Retrieving Records

There are many different ways to retrieve records from your database.

You can use standard SQL queries (these do not have to return results):

```
$sql = "SELECT * FROM MyTable";
$results = $db->Query($sql);
```

If $results is false, then we know there was an error retrieving our data. Otherwise, the matching records (if any) are returned to us.

You can query a single row or value:

```
$sql = "SELECT * FROM MyTable LIMIT 1";
$results = $db->QuerySingleRow($sql);
```

```
$sql = "SELECT Answer FROM MyTable WHERE QuestionID = 1";
$value = $db->QuerySingleValue($sql);
```

Want to run multiple queries and cache the results? Use arrays! The QueryArray method returns all the records in a multi-dimensional array (each row is another array inside an array):

```
$sql = "SELECT * FROM MyTable";
$array = $db->QueryArray($sql);
```

We can also return a single array if we only are expecting a single row to be returned. This makes it easy to do quick lookups.

```
$sql = "SELECT * FROM MyTable LIMIT 1";
$row = $db->QuerySingleRowArray($sql);
```

You can use the built in functions that generate SQL for you in the background.

The only drawback to using the built in select methods is that you cannot use joins on multiple tables and you cannot use `GROUP BY` or `DISTINCT` queries. You will still need to use SQL for this.

You can, however, use the native MySQL functions, views, and can even alias the column names. Let's look at an example. We'll start out with something simple (and we'll deal with

displaying and working with the results in the next section).

```
$db->SelectTable("employee");
```

This will give us all the records in the employee table. But what if we want to specify certain records? What if we don't want all the records but we only want the records where the active field is set to "Y" and the employee ID is 12? We can use the `SelectRows()` method and use a filter. When creating a filter, the key is the column and the value is the value in the database.

```
$filter = array("id" => 12, "hired" => "'Y'");
```

Notice that we still need to format the filter values for SQL. Now we can plug in our filter.

```
$db->SelectRows("employee", $filter);
```

This will select all records that match our filter. What if we want only certain columns?

```
$columns = array("id", "hired");
$db->SelectRows("employee", $filter, $columns);
```

If we wanted to name the columns something else, we could specify this in our columns array:

```
$columns = array("Employee ID" => "id", "Hired?" => "hired");
$db->SelectRows("employee", $filter, $columns);
```

Lastly, we'll sort on the `hire_date` and limit our results to only 1 record:

```
$db->SelectRows("employee", $filter, $columns, "hire_date", true, 1);
```

For this example, we used "`hire_date`" as our sort but we could specify an array if we wanted to sort by more that one field. This also works with columns. If we only wanted one column, we could have specified a string instead for the column parameter.

An important note on filters. If you want to create a filter where a value does NOT equal something or you want to specify the entire comparison, leave off the key for the field name:

```
$filter = array("id" => 12, "hired <> 'Y'");
```

Here is another example:

```php
$filter = array(
    "UCASE(hired) = 'Y'",
    "hire_date BETWEEN '2007-01-01' AND '2008-01-01'");
```

This rule applies on all filters in the Ultimate MySQL class.

Working with Record Data

Now that we can select records, let's deal with the results.

```
$results = $db->Query("SELECT * FROM MyTable");
```

This returns a resource (the same resourse object returned from the native `mysql_query()` method) on success, or FALSE on error. If you need to get the last results, use the `Records()` method to retrieve them. They are cached in the object for you until the next query is executed.

Let's assume we did not have an error. The `$results` variable now contains your records but there are easier ways to deal with the results using the Ultimate MySQL class.

If we want to simply dump the results to a table, we could use the `GetHTML()` method:

```
echo GetHTML();
```

Record Count: 18

TestID	Color	Age
1	Red	7
2	Blue	3
3	Green	10

This returns our last query as a table and takes additional style formatting as optional parameters.

Want to know how many records were returned? Use the `RowCount()` method:

```
echo $db->RowCount() . " records were returned from the last query";
```

If no records are returned, the `RowCount()` method returns `0`.

Alright, let's loop through our results and show some values in code:

```
while ($row = $db->Row()) {
    echo $row->Color . " - " . $row->Age . "<br />\n";
}
```

Each row is returned through each iteration of this loop. Every time a row is grabbed from the object, the cursor is automatically set to the next row. The first time I call `Row()`, I get the first row. The second time I call it, I get the second row, and so on.

To specify the specific column name, you can use `$row->ColumnName`. That was easy enough but what if you need a counting index?

```
for ($index = 0; $index < $db->RowCount(); $index++) {
    $row = $db->Row($index);

    echo "Row " . $index . ": " .
        $row->Color . " - " . $row->Age . "<br />\n";
}
```

The `Row()` method takes a row number parameter. If you want row 1, use `Row(1)`. This will automatically seek to this row for us and return the row data.

Let's take a close look at seeking through records. We could specify a row using the `Seek()` method and it will take us to that record but instead of using `Seek(1)` or `Row(1)`, we can simply use the `MoveFirst()` method:

```
$db->MoveFirst();
```

This will take us to the beginning of our records. `MoveLast()` takes us to the end of the records. If we want to know where we are at, we can use the `SeekPosition()` method to tell use what row we are on:

```
while (! $db->EndOfSeek()) {
    $row = $db->Row();

    echo "Row " . $db->SeekPosition() . ": " .
        $row->Color . " - " . $row->Age . "<br />\n";
}
```

`EndOfSeek()` returns true when we get to the end of our results.

What if we want arrays? Let's say we need an array for a certain method call. Use `RecordsArray()` instead ot `Records()` or `RowArray()` instead of `Row()`.

```
print_r(RecordsArray());
```

Or for each row:

```php
$db->MoveFirst();
while ($row = $db->RowArray()) {
    echo $row["Color"] . " - " . $row["Age"] . "<br />\n";
}
```

Both methods take an optional result type parameter (`MYSQL_ASSOC`, `MYSQL_NUM`, `MYSQL_BOTH`) that can give you a normal array, an associative array, or both.

You do not need to close the query results. Once the object is out of context (i.e. the code has finished executing for a page) or another query is performed, the result is automatically closed for you.

Just in case you need to close the result for any reason, use the `Release()` method. It takes no parameters and returns true on success and false on failure:

```php
$db->Release();
```

A couple of useful built-in functions worth noting when working with records are the `IsDate()` and `GetBooleanValue()` static methods. You do not need to even create an object to use these. As long as you've included the class file, these are available to you.

`IsDate()` will tell you if you have a date or time that can be used or converted by PHP. This can come in handy if you want to check data.

```php
if (MySQL::IsDate("January 1, 2007")) {
    echo "This is a date";
} else {
    echo "This is NOT a date";
}
```

Another great method is `GetBooleanValue()`. It will convert any value into a boolean. `GetBooleanValue("Yes")` will return true and `GetBooleanValue("fAlSe")` will return a false. Try it. It will convert 0, 1, "Y", "YeS", "N", "no", "True", "false", "on", or "OFF" into a boolean for you (it is not case specific). This is handy since MySQL will not store a true or false in a field.

You can also specify return values:

```php
echo MySQL::SQLBooleanValue("ON", "Ya", "Nope");
echo MySQL::SQLBooleanValue(1, 'so true!', 'super-false');
echo MySQL::SQLBooleanValue($row->active, "Active", "Not Active");
```

Want your return value formatted for SQL for you? Use the SQLVALUE constants as the last optional parameter:

```
echo MySQL::SQLBooleanValue(false, "1", "0", MySQL::SQLVALUE_NUMBER);
```

Inserting New Records

Inserting records can be a real pain when putting together SQL in code. Now you can use the `InsertRow()` method to greatly simplify the task.

Let's create an array where the column name is the key and the value is the SQL value:

```
$insert["First"] = MySQL::SQLValue("Bob");
$insert["Last"]  = MySQL::SQLValue("Smith");
$insert["Phone"] = MySQL::SQLValue("555-1212");
$insert["Age"]   = 20;
```

The `SQLValue()` static method will safely format our strings for us. Now let's insert this record into the **employee** table:

```
$new_id = $db->InsertRow("employee", $insert);
```

That's it! No more SQL INSERT code. You can still execute inserts using the `Query()` method but this is much easier. It also **returned the new ID** from the newly inserted record if it was successful and a false if not.

You can also get the last inserted ID (which is the the autonumber key field in any record) by using the `GetLastInsertID()` method.

```
$id = $db->GetLastInsertID();
```

Just as a side note, you could also build the insert array in the following manner:

```
$insert = array(
    "First" => "'Bob'",
    "Last"  => "'Smith'",
    "Phone" => "'555-1212'",
    "Age"   => 20)
```

Just remember that all values have to be formatted and ready for SQL. This allows us to use MySQL function calls as parameters for inserts if we need to.

Updating Existing Records

Updating records is much easier using the UpdateRows() method.

Let's say we want to update a record whose ID is 12. Let's create an array where the column name is the key and the new value is the SQL value:

```
$update["First"] = MySQL::SQLValue("Bob");
$update["Last"]  = MySQL::SQLValue("Smith");
$update["Phone"] = MySQL::SQLValue("555-1212");
$update["Age"]   = 20;
```

The SQLValue() static method will safely format our strings for us. Now let's create our filter to tell it which record needs to be updated:

```
$filter["ID"] = 12;
```

We could specify multiple filters if we needed. Simply add another filter to the array.

If you want to use a more advanced filter, just leave off the key and add the clause(s) as a string:

```
$filter[] = "ID > 10";
$filter[] = "NOT UPPER(First) = 'BOB'";
```

Now let's update the record in the employee table:

```
$id = $db->UpdateRows("employee", $update, $filter);
```

That's it. For the rare occasion we need to update all rows in a table, simply leave off the filter.

Just remember that all values have to be formatted and ready for SQL. This allows us to use MySQL function calls as parameters for updates if we need to.

Deleting Records

Deleting records is easy using the `DeleteRows()` method.

Let's say we want to delete record 12. Let's create our filter to tell the Ultimate MySQL object which record needs to be deleted:

```
$filter["ID"] = 12;
```

We could specify multiple filters if we needed. Simple add another filter to the array.

Now let's delete the record in the **employee** table:

```
$success = $db->DeleteRows("employee", $filter);
```

If you want to create a filter where a value does NOT equal something or you want to specify the entire comparison, leave off the key for the field name:

```
$filter = array("id" => 12, "active <> 'Y'");
```

```
$filter = array("hire_date BETWEEN '2007-01-01' AND '2008-01-01'");
```

For the rare occasion we need to delete all rows in a table, simply leave off the filter. If you wish to actually truncate a table and reset the auto-number fields to 1, use the following command:

```
$db->TruncateTable("MyTable");
```

Creating and Formatting SQL

All of the SQL generation tools in the Ultimate MySQL class are static. That means that you do not have to create an object from this class in order to use them. Simply include the class and use the syntax `MySQL::Method()`.

One of the most useful SQL methods is the `SQLValue()` static function. This will format any value into a SQL ready value. For example:

```
echo MySQL::SQLValue("Bob's / | \\ \"data\"");
```

This will format this string ready for a SQL insert or update. There are many types you can format data into. They are:

MySQL::SQLVALUE_BIT	Formats boolean values into 0 or 1
MySQL::SQLVALUE_BOOLEAN	Formats boolean values into 0 or 1
MySQL::SQLVALUE_DATE	Formats a date
MySQL::SQLVALUE_DATETIME	Formats a date and time
MySQL::SQLVALUE_NUMBER	Formats any number
MySQL::SQLVALUE_T_F	Formats a boolean value into T or F
MySQL::SQLVALUE_TEXT	(Default) Formats any text, blob, or string
MySQL::SQLVALUE_TIME	Formats any time
MySQL::SQLVALUE_Y_N	Formats boolean values into Y or N

`MySQL::SQLVALUE_TEXT` is the default. If you do not specify a type, text will be used.

For example, to format a date for MySQL, use:

```
$value = MySQL::SQLValue(date("m/d/Y"), MySQL::SQLVALUE_DATE)
```

Or to format a boolean as a "Y" or "N", use:

```
$value = MySQL::SQLValue(true, MySQL::SQLVALUE_Y_N)
```

Another useful formatting function is `SQLBooleanValue()`. It will convert any value into a boolean. `GetBooleanValue("Yes")` will return true and `GetBooleanValue("fAlSe")` will return a false. Try it. It will convert 0, 1, "Y", "YeS", "N", "no", "True", "false", "on", or "OFF"

into a boolean for you (it is not case specific). This is handy since MySQL will not store a true or false in a field.

You can also specify return values:

```
echo MySQL::SQLBooleanValue("ON", "Ya", "Nope");
echo MySQL::SQLBooleanValue(1, 'so true!', 'super-false');
echo MySQL::SQLBooleanValue($row->active, "Active", "Not Active");
```

Want your return value formatted for SQL for you? Use the SQLVALUE constants as the last optional parameter:

```
echo MySQL::SQLBooleanValue(false, "1", "0", MySQL::SQLVALUE_NUMBER);
```

NOTE: SQLFix() and SQLUnfix() were designed to format SQL strings and are **depreciated** (and are only present for backwards compatibility). They leave slashes in database records which is not the ideal situation for your data!

If you wish to generate SQL, you can use the MySQL::BuildSQLSelect(), MySQL::BuildSQLInsert(), MySQL::BuildSQLUpdate(), and MySQL::BuildSQLDelete() methods. These take the exact same parameters as SelectRows(), InsertRow(), UpdateRows(), and DeleteRows() (*in fact, these methods actually use these static functions in the background to execute SQL statements*). See these commands for more information.

When creating a filter (the WHERE clause) for these methods, the key is the column and the value is the value in the database.

```
$filter = array("id" => 12, "hired" => "'Y'");
```

If you want to create a filter where a value does NOT equal something or you want to specify the entire comparison, leave off the key for the field name:

```
$filter = array("id" => 12, "hired <> 'Y'");
```

```
$filter = array(
    "UCASE(hired) = 'Y'",
    "hire_date BETWEEN '2007-01-01' AND '2008-01-01'");
```

If you would like to see the SQL generated by these methods or see the last executed queries, use the GetLastSQL() method after any query, insert, update, or delete. This method will return the last SQL executed by the Ultimate MySQL class. This could be helpful if you generate error logs.

Error Handling

Ultimate MySQL has two types of error handling built into it - default and the new exception based. Let's take a look at the default error handling.

Most methods will return a false if there was a failure.

```
$success = $db->SelectDatabase("mydb");
```

How you handle an error is up to you but there are some tools incorporated into the class that can help. `Kill()` is one of those methods.

If you want your script to stop executing and an error to show, use the `Kill()` method:

```
if (! $db->SelectDatabase("mydb")) $db->Kill();
```

This will stop any execution and dump the error to the screen. It works exactly the same as `exit(mysql_error())`. This is convenient but may not be the best idea for security. Some of the MySQL errors give information away about your database.

You can always get the error by using the `Error()` method:

```
if (! $db->SelectDatabase("mydb")) echo $db->Error();
```

You may decide to log the errors using the Error() method. It's just as easy to check for an error using this method:

```
if ($db->Error()) {
    echo "We have an error: " . $db->Error();
}
```

MySQL also hands back error numbers and they can be accessed using the ErrorNumber() method. ResetError() clears any error information.

The other (and better but more slightly more difficult) way to handle errors is by using exceptions. In order to use exceptions, you must turn this on by settings the ThrowExceptions property equal to true.

```
$db->ThrowExceptions = true;
```

Any time an error is found, an exception is thrown. You must use a try/catch block to catch exceptions.

```
try {
    $db->Query("SELECT ... ");
    $db->Query("SELECT ... ");
    $db->Query("BAD SQL QUERY THAT CREATES AN ERROR");
} catch(Exception $e) {
    // If an error occurs, do this
    echo "We caught the error: " . $e->getMessage();
}
```

You can put your database code (selects, insert, updates, and deletes) in the try block and if it fails, log or process the error in the catch block.

The try/catch block is perfect to use with transaction processing.

For more information on exceptions and try/catch blocks, see the PHP web site.

Transaction Processing

Now let's check out transactional processing. This is an excellent way to keep solid database integrity.

Let's say that you have to insert multiple records that depend on one another. If one of the insert queries fail, you want to remove the other inserts that were done before it. Transaction processing allows us to do this.

We start a transaction, execute some queries, if one fails, all we have to do it rollback. When you rollback, any query executed since you began the transaction is removed as if it never happened. If they were all successful, then you commit and all changes are saved. This can be really useful on larger databases that have parent child relationships on the tables.

Let's start out by creating a new transaction:

```
$db->TransactionBegin();
```

Now let's insert some records. We are going to skip checking for any query errors just for this part of the example:

```
$db->Query("INSERT INTO test (Color, Age) VALUES ('Blue', '3')");
$db->Query("INSERT INTO test (Color, Age) VALUES ('Green', '10')");
$db->Query("INSERT INTO test (Color, Age) VALUES ('Yellow', '1')");
```

Oops! We don't really want to save these to the database, let's rollback:

```
$db->TransactionRollback();
```

Now if you stopped right here and looked in the database, nothing has changed... not one thing was saved. It "rolled back" all these inserts.

Transaction processing also works with the InsertRow(), UpdateRows(), and DeleteRows() methods.

Let's try that again, but this time, we will commit the changes.

```
// Begin a new transaction, but this time, let's check for errors.
if (! $db->TransactionBegin()) $db->Kill();

// We'll create a flag to check for any errors
$success = true;

// Insert some records and if there are any errors, set the flag to false
$sql = "INSERT INTO test (Color, Age) VALUES ('Blue', '3')";
if (! $db->Query($sql)) $success = false;

$sql = "INSERT INTO test (Color, Age) VALUES ('Green', '10')";
if (! $db->Query($sql)) $success = false;

$sql = "INSERT INTO test (Color, Age) VALUES ('Yellow', '1')";
if (! $db->Query($sql)) $success = false;
```

Notice that you can even view what the new IDs are going to be before the records are commited. Transaction processing allows you to actually see what the final results will look like in the database.

```
echo "The new ID for the last inserted record is " .
    $db->GetLastInsertID();
echo "<br />\r";
```

Now let's check for errors:

```
// If there were no errors...
if ($success) {

    // Commit the transaction and save these records to the database
    if (! $db->TransactionEnd()) $db->Kill();

} else { // Otherwise, there were errors...

    // Rollback our transaction
    if (! $db->TransactionRollback()) $db->Kill();

}
```

Transaction processing works with all INSERT, UPDATES, and DELETE queries.

They are terrific to use in TRY/CATCH blocks for error handling.

```
// Turn on exception handling
$db->ThrowExceptions = true;

// Here's our try/catch block
try {

    // Begin our transaction
    $db->TransactionBegin();

    //Execute a query (or multiple queries)
    $db->Query($sql);

    // Commit - this line never runs if there is an error
    $db->TransactionEnd();

} catch(Exception $e) {

    // If an error occurs, rollback and show the error
    $db->TransactionRollback();
    exit($e->getMessage());

}
```

Here is a great article (sample chapter) on the basics of transaction processing courtesy of Sams Publishing.

MySQL Transactions Overview

Date: Sep 13, 2002 By Julie C. Meloni.
Sample Chapter is provided courtesy of Sams.
http://www.samspublishing.com/articles/article.asp?p=29312&rl=1

One of the greatest additions to MySQL in recent versions is its support for transactional processing. Gain an understanding of the benefits of transactions, and more importantly, how to use them in your applications.

Transactions are a new addition to MySQL but not to relational database systems in general. If you have used an enterprise database system, such as Oracle or Microsoft SQL Server, the transactional concept should seem familiar. If this is your first venture into relational databases, this hour will bring you up to speed and provide an overview of using transactions in MySQL.

In this hour, you will learn about

- **The basic properties of transactions**
- **Berkeley DB, InnoDB, and Gemini table types**

What Are Transactions?

A transaction is a sequential group of database manipulation operations, which is performed as if it were one single work unit. In other words, a transaction will never be complete unless each individual operation within the group is successful. If any operation within the transaction fails, the entire transaction will fail.

A good example would be a banking transaction, specifically a transfer of $100 between two accounts. In order to deposit money into one account, you must first take money from another account. Without using transactions, you would have to write SQL statements that do the following:

1. **Check that the balance of the first account is greater than $100.**
2. **Deduct $100 from the first account.**
3. **Add $100 to the second account.**

Additionally, you would have to write your own error-checking routines within your program, specifically to stop the sequence of events should the first account not have more than $100 or should the deduction statement fail. This all changes with transactions, for if any part of the operation fails, the entire transaction is rolled back. This means that the tables and the data inside them revert to their previous state.

Properties of Transactions

Transactions have the following four standard properties, usually referred to by the acronym ACID:

- **Atomicity ensures that all operations within the work unit are completed successfully; otherwise, the transaction is aborted at the point of failure, and previous operations are rolled back to their former state.**
- **Consistency ensures that the database properly changes states upon a successfully committed transaction.**
- **Isolation enables transactions to operate independently of and transparent to each other.**
- **Durability ensures that the result or effect of a committed transaction persists in case of a system failure.**

In MySQL, transactions begin with the statement BEGIN WORK and end with either a COMMIT or a ROLLBACK statement. The SQL commands between the beginning and ending statements form the bulk of the transaction.

COMMIT and ROLLBACK

When a successful transaction is completed, the COMMIT command should be issued so that the changes to all involved tables will take effect. If a failure occurs, a ROLLBACK command should be issued to return every table referenced in the transaction to its previous state.

NOTE

In MySQL as well as NuSphere's Enhanced MySQL, you can set the value of a session variable called AUTOCOMMIT. If AUTOCOMMIT is set to 1 (the default), then each SQL statement (within a transaction or not) is considered a complete transaction, committed by default when it finishes. When AUTOCOMMIT is set to 0, by issuing the SET AUTOCOMMIT=0 command, the subsequent series of statements acts like a transaction, and no activities are committed until an explicit COMMIT statement is issued.

If transactions were not used in application development, a large amount of programming time would be spent on intricate error checking. For example, suppose your application handles customer order information, with tables holding general order information as well as line items for that order. To insert an order into the system, the process would be something like the following:

1. **Insert a master record into the master order table.**

2. **Retrieve the ID from the master order record you just entered.**

3. **Insert records into the line items table for each item ordered.**

If you are not in a transactional environment, you will be left with some straggly data floating around your tables; if the addition of the record into the master order table succeeds, but steps 2 or 3 fail, you are left with a master order without any line items. The responsibility then falls on you to use programming logic and check that all relevant records in multiple tables have been added or go back and delete all the records that have been added and offer error messages to the user. This is extremely time-consuming, both in man-hours as well as in program-execution time.

In a transactional environment, you'd never get to the point of childless rows, as a transaction either fails completely or is completely successful.

Row-Level Locking

Transactional table types support row-level locking, which differs from the table-level locking that is enforced in MyISAM and other nontransactional table types. With tables that support row-level locking, only the row touched by an INSERT, UPDATE, or DELETE statement is inaccessible until a COMMIT is issued.

Rows affected by a SELECT query will have shared locks, unless otherwise specified by the programmer. A shared lock allows for multiple concurrent SELECT queries of the data. However, if you hold an exclusive lock on a row, you are the only one who can read or modify that row until the lock is released. Locks are released when transactions end through a COMMIT or ROLLBACK statement.

Setting an exclusive lock requires you to add the FOR UPDATE clause to your query. In the sequence below, you can see how locks are used to check available inventory in a product catalog before processing an order. This example builds on the previous example by adding more condition-checking.

NOTE

This sequence of events is independent of the programming language used; the logical path can be created in whichever language you use to create your application.

1. **Begin transaction.**

   ```
   BEGIN WORK;
   ```

2. **Check available inventory for a product with a specific ID, using a table called inventory and a field called qty.**

   ```
   SELECT qty FROM inventory WHERE id = 'ABC-001' FOR UPDATE;
   ```

3. **If the result is less than the amount ordered, rollback the transaction to release the lock.**

   ```
   ROLLBACK;
   ```

4. **If the result is greater than the amount ordered, continue issuing a statement that reserves the required amount for the order.**

   ```
   UPDATE inventory SET qty = qty - [amount ordered] WHERE id = 'ABC-001';
   ```

5. **Insert a master record into the master order table.**

6. **Retrieve the ID from the master order record you just entered.**

7. **Insert records into the line items table for each item ordered.**

8. **If steps 5 through 7 are successful, commit the transaction and release the lock.**

   ```
   COMMIT;
   ```

While the transaction remains uncommitted and the lock remains in effect, no other users can access the record in the inventory table for the product with the ID of ABC-001. If a user requests the current quantity for the item with the ID of ABC-002, that row still operates under the shared lock rules and can be read.

For the complete article, visit the Sams Publishing online at: http://www.samspublishing.com/articles/article.asp?p=29312&rl=1

Timing Queries

It is easy to time queries using Ultimate MySQL because there is a built-in timer at your disposal. To start the timer, just use:

```
$db->TimerStart();
```

Now execute your code or queries and then call:

```
$db->TimerStop();
```

To see how long the timer ran, use `TimerDuration()` to retrieve the number of microseconds that passed.

```
echo $db->TimerDuration() . " microseconds";
```

If you only wish to time a single query, use the `QueryTimed()` method.

```
$db->QueryTimed("SELECT * FROM MyTable");
echo "Query took " . $db->TimerDuration() . " microseconds";
```

This is useful if you are tracking down bottlenecks in your application.

Exploring the Database

Ultimate MySQL gives you the ability to find out about the database and query results easily. The majority of these methods will work on the last executed query if no source is specified.

`GetColumnComments()` returns the comments for fields in a table into an array:

```
$columns = $db->GetColumnComments("MyTable");
foreach ($columns as $column => $comment) {
    echo $column . " = " . $comment . "<br />\n";
}
```

`GetColumnCount()` returns the number of columns in a result set or table:

```
echo "Total Columns: " . $db->GetColumnCount("MyTable");
```

If a table name is not specified, the column count is returned from the last query.

`GetColumnDataType()` returns the data type for any specified column (name or position). If the column does not exists (or no records exist if a table is not specified), it returns false.

```
echo "Type: " . $db->GetColumnDataType("first_name", "customer");
echo "Type: " . $db->GetColumnDataType(1, "customer");
```

`GetColumnID()` returns the position of a column:

```
echo "Column Position: " .
    $db->GetColumnID("first_name", "customer");
```

`GetColumnLength()` returns the length of a text field:

```
echo "Length: " . $db->GetColumnLength("first_name", "customer");
```

If a table name is not specified, the column length is returned from the last query. This can be handy if you are building dynamic forms.

`GetColumnName()` returns the field name for a specified column number:

```
echo "Column Name: " . $db->GetColumnName(0, "MyTable"); // Evaluates a table
echo "Column Name: " . $db->GetColumnName(0); // Evaluates the current
results
```

GetColumnNames() returns the field names in a table or query in an array:

```
$columns = $db->GetColumnNames("MyTable");
foreach ($columns as $columnName) {
    echo $columnName . "<br />\n";
}
```

GetTables() returns table names from the database into an array. If the database does not contains any tables, the returned value is false:

```
$tables = $db->GetTables();
foreach ($tables as $table) {
    echo $table . "<br />\n";
}
```

Ultimate MySQL Class

Description

Description | Vars (details) | Methods (details) Constants (details)

Ultimate MySQL Wrapper Class for PHP 5.x

- Establish MySQL server connections easily
- Execute SQL queries
- Retrieve query results into objects or arrays
- Retrieve the last inserted ID
- Manage transactions (transaction processing)
- Retrieve the list tables of a database
- Retrieve the list fields of a table (or field comments)
- Retrieve the length or data type of a field
- Measure the time a query takes to execute
- Display query results in an HTML table
- Easy formatting for SQL parameters and values
- Generate SQL Selects, Inserts, Updates, and Deletes
- Error handling with error numbers and text
- **And much more!**

http://www.phpclasses.org/browse/package/3698.html

```
Feb 02, 2007 - Written by Jeff Williams (Initial Release)
Feb 11, 2007 - Contributions from Frank P. Walentynowicz
Feb 21, 2007 - Contribution from Larry Wakeman
Feb 21, 2007 - Bug Fixes and PHPDoc
Mar 09, 2007 - Contribution from Nicola Abbiuso
Mar 22, 2007 - Added array types to RecordsArray and RowArray
Jul 01, 2007 - Class name change, constructor values, static methods, fixe
Jul 16, 2007 - Bug fix, removed test, major improvements in error handling
Aug 11, 2007 - Added InsertRow() and UpdateRows() methods
Aug 19, 2007 - Added BuildSQL static functions, DeleteRows(), SelectRows(),
               IsConnected(), and ability to throw Exceptions on errors
Sep 07, 2007 - Enhancements to SQL SELECT (column aliases, sorting, limits)
Sep 09, 2007 - Updated SelectRows(), UpdateRows() and added SelectTable(),
               TruncateTable() and SQLVALUE constants for SQLValue()
```

```
Oct 23, 2007 - Added QueryArray(), QuerySingleRow(), QuerySingleRowArray(),
               QuerySingleValue(), HasRecords(), AutoInsertUpdate()
Oct 28, 2007 - Small bug fixes
Nov 28, 2007 - Contribution from Douglas Gintz
```

- **author:** Jeff L. Williams
- **version:** 2.4.1

Usage:

```
include("mysql.class.php");

$db = new MySQL();
$db = new MySQL(true, "database");
$db = new MySQL(true, "database", "localhost", "username", "password");
```

Class Constant Summary (Use with SQLValue() method)

Description | *Constants* (details) Vars (details) | | Methods (details)

SQLVALUE_BIT = *"bit"*
SQLVALUE_BOOLEAN = *"boolean"*
SQLVALUE_DATE = *"date"*
SQLVALUE_DATETIME = *"datetime"*
SQLVALUE_NUMBER = *"number"*
SQLVALUE_TEXT = *"text"*
SQLVALUE_TIME = *"time"*
SQLVALUE_T_F = *"t-f"*
SQLVALUE_Y_N = *"y-n"*

Variable Summary

Description | *Vars* (details) | Methods (details) Constants (details)

boolean **$ThrowExceptions**

Method Summary

Description | Constants (details) Vars (details) | *Methods* (details)

MySQL **__construct** ([*boolean* **$connect** = true], [*string* **$database** = ""], [*string* **$server** = ""], [*string* **$username** = ""], [*string* **$password** = ""], [*string* **$charset** = ""])

void **__destruct** ()

boolean **AutoInsertUpdate** (*string* **$tableName**, *array* **$valuesArray**, *array* **$whereArray**)

boolean **BeginningOfSeek** ()

string **BuildSQLDelete** (*string* **$tableName**, *array* **$whereArray**)

string **BuildSQLInsert** (*string* **$tableName**, *array* **$valuesArray**)

string **BuildSQLSelect** (*string* **$tableName**, [*array* **$whereArray** = null], [*array/string* **$columns** = null], [*array/string* **$sortColumns** = null], [*boolean* **$sortAscending** = true], [*integer/string* **$limit** = null])

string **BuildSQLUpdate** (*string* **$tableName**, *array* **$valuesArray**, [*array* **$whereArray** = null])

string **BuildSQLWhereClause** (*array* **$whereArray**)

object Returns **Close** ()

boolean **DeleteRows** (*string* **$tableName**, [*array* **$whereArray** = null])

boolean **EndOfSeek** ()

string **Error** ()

integer **ErrorNumber** ()

boolean **GetBooleanValue** (*any* **$value**)

array **GetColumnComments** (*string* **$table**)

integer **GetColumnCount** ([*string* **$table** = ""])

string **GetColumnDataType** (*string* **$column**, [*string* **$table** = ""])

integer **GetColumnID** (*string* **$column**, [*string* **$table** = ""])

integer **GetColumnLength** (*string* **$column**, [*string* **$table** = ""])

integer **GetColumnName** (*string* **$columnID**, [*string* **$table** = ""])

array **GetColumnNames** ([*string* **$table** = ""])

string **GetHTML** ([*boolean* **$showCount** = true], [*string* **$styleTable** = null], [*string* **$styleHeader** = null], [*string* **$styleData** = null])

integer **GetLastInsertID** ()

string **GetLastSQL** ()

array **GetTables** ()

boolean **HasRecords** ([*string* **$sql** = ""])

integer **InsertRow** (*string* **$tableName**, *array* **$valuesArray**)

boolean **IsConnected** ()

boolean **IsDate** (*date/string* **$value**)

void **Kill** ([*mixed* **$message** = ''])

boolean **MoveFirst** ()

boolean **MoveLast** ()

boolean **Open** ([*string* **$database** = ""], [*string* **$server** = ""], [*string* **$username** = ""], [*string* **$password** = ""], [*string* **$charset** = ""], [*boolean* **$pcon** = false])

object PHP **Query** (*string* **$sql**)

array **QueryArray** (*string* **$sql**, [*integer* **$resultType** = MYSQL_BOTH])

object PHP **QuerySingleRow** (*string* **$sql**)

array **QuerySingleRowArray** (*string* **$sql**, [*integer* **$resultType** = MYSQL_BOTH])

mixed **QuerySingleValue** (*string* **$sql**)

object PHP **QueryTimed** (*string* **$sql**)

object PHP **Records** ()

Records **RecordsArray** ([*integer* **$resultType** = MYSQL_BOTH])

boolean **Release** ()

object PHP **Row** ([*integer* **$optional_row_number** = null])

array **RowArray** ([*integer* **$optional_row_number** = null], [*integer* **$resultType** = MYSQL_BOTH])

integer **RowCount** ()

object Fetched **Seek** (*integer* **$row_number**)

integer **SeekPosition** ()

boolean **SelectDatabase** (*string* **$database**, [*string* **$charset** = ""])

boolean **SelectRows** (*string* **$tableName**, [*array* **$whereArray** = null], [*array/string* **$columns** = null], [*array/string* **$sortColumns** = null], [*boolean* **$sortAscending** = true], [*integer/string* **$limit** = null])

boolean **SelectTable** (*string* **$tableName**)

string **SQLBooleanValue** (*any* **$value**, *any* **$trueValue**, *any* **$falseValue**, [*string* **$datatype** = self::SQLVALUE_TEXT])

string **SQLFix** (*string* **$value**)

string **SQLUnfix** (*string* **$value**)

string **SQLValue** (*any* **$value**, [*string* **$datatype** = self::SQLVALUE_TEXT])

Float **TimerDuration** ([*integer* **$decimals** = 4])

void **TimerStart** ()

void **TimerStop** ()

boolean **TransactionBegin** ()

boolean **TransactionEnd** ()

boolean **TransactionRollback** ()

boolean **TruncateTable** (*string* **$tableName**)

boolean **UpdateRows** (*string* **$tableName**, *array* **$valuesArray**, [*array* **$whereArray** = null])

Variables

Description | Vars (*details*) Constants (details) | Methods (details)

boolean **$ThrowExceptions** = false (line 47)

Determines if an error throws an exception

- **var:** Set to true to throw error exceptions
- **access:** public

Methods

Constructor __construct

Constructor: Opens the connection to the database

- **access:** public

MySQL **__construct** ([*boolean* **$connect** = true], [*string* **$database** = ""], [*string* **$server** = ""], [*string* **$username** = ""], [*string* **$password** = ""], [*string* **$charset** = ""])

- *boolean* ***$connect****: (Optional) Auto-connect when object is created*
- *string* ***$database****: (Optional) Database name*
- *string* ***$server****: (Optional) Host address*
- *string* ***$username****: (Optional) User name*
- *string* ***$password****: (Optional) Password*
- *string* ***$charset****: (Optional) Character set (i.e. 'utf8')*

Example:

```
$db = new MySQL();
$db = new MySQL(true, "database");
$db = new MySQL(true, "database", "localhost", "username", "password");
```

Destructor __destruct

Destructor: Closes the connection to the database

- **access:** public

void **__destruct** ()

AutoInsertUpdate (line 113)

Automatically does an INSERT or UPDATE depending if an existing record exists in a table

- **return:** Returns TRUE on success or FALSE on error

boolean **AutoInsertUpdate** (*string* **$tableName**, *array* **$valuesArray**, *array* **$whereArray**)

- *string **$tableName**: The name of the table*
- *array **$valuesArray**: An associative array containing the column names as keys and values as data. The values must be SQL ready (i.e. quotes around strings, formatted dates, ect)*
- *array **$whereArray**: An associative array containing the column names as keys and values as data. The values must be SQL ready (i.e. quotes around strings, formatted dates, ect).*

BeginningOfSeek

Returns true if the internal pointer is at the beginning of the records

- **return:** TRUE if at the first row or FALSE if not
- **access:** public

boolean **BeginningOfSeek** ()

Example:

```
if ($db->BeginningOfSeek()) {
   echo "We are at the beggining of the record set";
}
```

BuildSQLDelete

[STATIC] Builds a SQL DELETE statement

- **return:** Returns the SQL DELETE statement
- **access:** public
- **static**

string **BuildSQLDelete** (*string* **$tableName**, [*array* **$whereArray** = null])

- *string **$tableName**: The name of the table*

- *array **$whereArray**: (Optional) An associative array containing the column names as keys and values as data. The values must be SQL ready (i.e. quotes around strings, formatted dates, ect). If not specified then all values in the table are deleted.*

```
// Let's create an array for the example
// $arrayVariable["column name"] = formatted SQL value
$filter["ID"] = MySQL::SQLValue(7, MySQL::SQLVALUE_NUMBER);
// Echo out the SQL statement
echo MySQL::BuildSQLDelete("MyTable", $filter);
```

BuildSQLInsert

[STATIC] Builds a SQL INSERT statement

- **return:** Returns a SQL INSERT statement
- **access:** public
- **static**

string **BuildSQLInsert** (*string* **$tableName**, *array* **$valuesArray**)

- *string **$tableName**: The name of the table*
- *array **$valuesArray**: An associative array containing the column names as keys and values as data. The values must be SQL ready (i.e. quotes around strings, formatted dates, ect)*

```
// Let's create an array for the example
// $arrayVariable["column name"] = formatted SQL value
$values["Name"] = MySQL::SQLValue("Violet");
$values["Age"] = MySQL::SQLValue(777, MySQL::SQLVALUE_NUMBER);
// Echo out the SQL statement
echo MySQL::BuildSQLInsert("MyTable", $values);
```

BuildSQLSelect

Builds a simple SQL SELECT statement

- **return:** Returns a SQL SELECT statement
- **access:** public
- **static**

string **BuildSQLSelect** (*string* **$tableName**, [*array* **$whereArray** = null], [*array/string* **$columns** = null], [*array/string* **$sortColumns** = null], [*boolean* **$sortAscending** = true], [*integer/string* **$limit** = null])

- *string* ***$tableName****: The name of the table*
- *array* ***$whereArray****: (Optional) An associative array containing the column names as keys and values as data. The values must be SQL ready (i.e. quotes around strings, formatted dates, ect)*
- *array/string* ***$columns****: (Optional) The column or list of columns to select*
- *array/string* ***$sortColumns****: (Optional) Column or list of columns to sort by*
- *boolean* ***$sortAscending****: (Optional) TRUE for ascending; FALSE for descending This only works if $sortColumns are specified*
- *integer/string* ***$limit****: (Optional) The limit of rows to return*

```
// Let's create an array for the example
// $arrayVariable["column name"] = formatted SQL value
$values["Name"] = MySQL::SQLValue("Violet");
$values["Age"] = MySQL::SQLValue(777, MySQL::SQLVALUE_NUMBER);
// Echo out the SQL statement
echo MySQL::BuildSQLSelect("MyTable", $values);
```

BuildSQLUpdate

[STATIC] Builds a SQL UPDATE statement

- **return:** Returns a SQL UPDATE statement
- **access:** public
- **static**

string **BuildSQLUpdate** (*string* **$tableName**, *array* **$valuesArray**, [*array* **$whereArray** = null])

- *string* ***$tableName****: The name of the table*
- *array* ***$valuesArray****: An associative array containing the column names as keys and values as data. The values must be SQL ready (i.e. quotes around strings, formatted dates, ect)*
- *array* ***$whereArray****: (Optional) An associative array containing the column names as keys and values as data. The values must be SQL ready (i.e. quotes around strings, formatted dates, ect). If not specified then all values in the table are updated.*

```
// Let's create two arrays for the example
```

```
// $arrayVariable["column name"] = formatted SQL value
$values["Name"] = MySQL::SQLValue("Violet");
$values["Age"] = MySQL::SQLValue(777, MySQL::SQLVALUE_NUMBER);
$filter["ID"] = MySQL::SQLValue(10, MySQL::SQLVALUE_NUMBER);
// Echo out some SQL statements
echo MySQL::BuildSQLUpdate("Test", $values, $filter)";
```

BuildSQLWhereClause

[STATIC] Builds a SQL WHERE clause from an array. If a key is specified, the key is used at the field name and the value as a comparison. If a key is not used, the value is used as the clause.

- **return:** Returns a string containing the SQL WHERE clause
- **access:** public
- **static**

string **BuildSQLWhereClause** (*array* **$whereArray**)

- *array* ***$whereArray****: An associative array containing the column names as keys and values as data. The values must be SQL ready (i.e. quotes around strings, formatted dates, ect)*

Close

Close current MySQL connection

- **return:** TRUE on success or FALSE on error
- **access:** public

object Returns **Close** ()

Example:

```
$db->Close();
```

DeleteRows

Deletes rows in a table based on a WHERE filter (can be just one or many rows based on the filter)

- **return:** Returns TRUE on success or FALSE on error
- **access:** public

boolean **DeleteRows** (*string* **$tableName**, [*array* **$whereArray** = null])

- *string **$tableName**: The name of the table*
- *array **$whereArray**: (Optional) An associative array containing the column names as keys and values as data. The values must be SQL ready (i.e. quotes around strings, formatted dates, ect). If not specified then all values in the table are deleted.*

Example

```
// $arrayVariable["column name"] = formatted SQL value
$filter["ID"] = 7;

// Execute the delete
$result = $db->DeleteRows("MyTable", $filter);

// If we have an error
if (! $result) {

   // Show the error and kill the script
   $db->Kill();

}
```

EndOfSeek

Returns true if the internal pointer is at the end of the records

- **return:** TRUE if at the last row or FALSE if not
- **access:** public

boolean **EndOfSeek** ()

Example:

```
if ($db->EndOfSeek()) {
   echo "We are at the end of the record set";
}
```

Error

Returns the last MySQL error as text

- **return:** Error text from last known error
- **access:** public

string **Error** ()

Example:

```
if (! $db->Query("SELECT * FROM Table")) {
   echo $db->Error(); //Shows the error
}

if ($db->Error()) $db->Kill();
```

ErrorNumber

Returns the last MySQL error as a number

- **return:** Error number from last known error
- **access:** public

integer **ErrorNumber** ()

Example:

```
if ($db->ErrorNumber() <> 0) {
   $db->Kill(); //show the error message
}
```

GetBooleanValue

[STATIC] Converts any value of any datatype into boolean (true or false)

- **return:** Returns TRUE or FALSE
- **access:** public
- **static**

boolean **GetBooleanValue** (*any* **$value**)

- *any **$value**: Value to analyze for TRUE or FALSE*

Example:

```
echo (MySQL::GetBooleanValue("Y") ? "True" : "False");

echo (MySQL::GetBooleanValue("no") ? "True" : "False");

echo (MySQL::GetBooleanValue("TRUE") ? "True" : "False");

echo (MySQL::GetBooleanValue(1) ? "True" : "False");
```

GetColumnComments

Returns the comments for fields in a table into an array or NULL if the table has not got any fields

- **return:** An array that contains the column comments
- **access:** public

array **GetColumnComments** (*string* **$table**)

- *string **$table**: Table name*

Example:

```
$columns = $db->GetColumnComments("MyTable");
foreach ($columns as $column => $comment) {
```

```
    echo $column . " = " . $comment . "<br />\n";
}
```

GetColumnCount

This function returns the number of columns or returns FALSE on error

- **return:** The total count of columns
- **access:** public

integer **GetColumnCount** ([*string* **$table** = ""])

- *string **$table**: (Optional) If a table name is not specified, the column count is returned from the last query*

Example:

```
echo "Total Columns: " . $db->GetColumnCount("MyTable");
```

GetColumnDataType

This function returns the data type for a specified column. If the column does not exists or no records exist, it returns FALSE

- **return:** MySQL data (field) type
- **access:** public

string **GetColumnDataType** (*string* **$column**, [*string* **$table** = ""])

- *string **$column**: Column name or number (first column is 0)*
- *string **$table**: (Optional) If a table name is not specified, the last returned records are used*

Example:

```
echo "Type: " . $db->GetColumnDataType("FirstName", "Customer");
```

GetColumnID

This function returns the position of a column

- **return:** Column ID
- **access:** public

integer **GetColumnID** (*string* **$column**, [*string* **$table** = ""])

- *string **$column**: Column name*
- *string **$table**: (Optional) If a table name is not specified, the last returned records are used.*

Example:

```
echo "Column Position: " . $db->GetColumnID("FirstName", "Customer");
```

GetColumnLength

This function returns the field length or returns FALSE on error

- **return:** Field length
- **access:** public

integer **GetColumnLength** (*string* **$column**, [*string* **$table** = ""])

- *string **$column**: Column name*
- *string **$table**: (Optional) If a table name is not specified, the last returned records are used.*

Example:

```
echo "Length: " . $db->GetColumnLength("FirstName", "Customer");
```

GetColumnName

This function returns the name for a specified column number. If the index does not exists or no records exist, it returns FALSE

- **return:** Field Length
- **access:** public

integer **GetColumnName** (*string* **$columnID**, [*string* **$table** = ""])

- *string **$columnID**: Column position (0 is the first column)*
- *string **$table**: (Optional) If a table name is not specified, the last returned records are used.*

Example:

```
echo "Column Name: " . $db->GetColumnName(0);
```

GetColumnNames

Returns the field names in a table in an array or NULL if the table has no fields

- **return:** An array that contains the column names
- **access:** public

array **GetColumnNames** ([*string* **$table** = ""])

- *string **$table**: (Optional) If a table name is not specified, the last returned records are used*

Example:

```
$columns = $db->GetColumnNames("MyTable");
foreach ($columns as $columnName) {
   echo $columnName . "<br />\n";
}
```

GetHTML

This function returns the last query as an HTML table

- **return:** HTML containing a table with all records listed
- **access:** public

string **GetHTML** ([*boolean* **$showCount** = true], [*string* **$styleTable** = null], [*string* **$styleHeader** = null], [*string* **$styleData** = null])

- *boolean **$showCount**: (Optional) TRUE if you want to show the row count, FALSE if you do not want to show the count*
- *string **$styleTable**: (Optional) Style information for the table*
- *string **$styleHeader**: (Optional) Style information for the header row*
- *string **$styleData**: (Optional) Style information for the cells*

Example:

```
$db->Query("SELECT * FROM Customer");
echo $db->GetHTML();
```

GetLastInsertID

Returns the last autonumber ID field from a previous INSERT query

- **return:** ID number from previous INSERT query
- **access:** public

integer **GetLastInsertID** ()

Example:

```
$sql = "INSERT INTO Employee (Name) Values ('Bob')";
if (! $db->Query($sql)) {
   $db->Kill();
}
echo "Last ID inserted was: " . $db->GetLastInsertID();
```

GetLastSQL

Returns the last SQL statement executed

- **return:** Current SQL query string
- **access:** public

string **GetLastSQL** ()

Example:

```
$sql = "INSERT INTO Employee (Name) Values ('Bob')";
if (! $db->Query($sql)) {$db->Kill();}
echo $db->GetLastSQL();
```

GetTables

This function returns table names from the database into an array. If the database does not contains any tables, the returned value is FALSE

- **return:** An array that contains the table names

array **GetTables** ()

Example:

```
$tables = $db->GetTables();
foreach ($tables as $table) {
   echo $table . "<br />\n";
}
```

HasRecords (line 801)

Determines if a query contains any rows

- **return:** TRUE if records exist, FALSE if not or query error
- **access:** public

boolean **HasRecords** ([*string* **$sql** = ""])

- *string **$sql**: [Optional] If specified, the query is first executed Otherwise, the last query is used for comparison*

InsertRow

Inserts a row into a table in the connected database

- **return:** Returns last insert ID on success or FALSE on failure
- **access:** public

integer **InsertRow** (*string* **$tableName**, *array* **$valuesArray**)

- *string* ***$tableName****: The name of the table*
- *array* ***$valuesArray****: An associative array containing the column names as keys and values as data. The values must be SQL ready (i.e. quotes around strings, formatted dates, ect)*

Example

```
// $arrayVariable["column name"] = formatted SQL value
$values["Name"] = MySQL::SQLValue("Violet");
$values["Age"]  = MySQL::SQLValue(777, MySQL::SQLVALUE_NUMBER);

// Execute the insert
$result = $db->InsertRow("MyTable", $values);

// If we have an error
if (! $result) {

   // Show the error and kill the script
   $db->Kill();

} else {

   // No error, show the new record's ID
   echo "The new record's ID is: " . $result;

}
```

IsDate

[STATIC] Determines if a value of any data type is a date PHP can convert

- **return:** Returns TRUE if value is date or FALSE if not date
- **access:** public
- **static**

boolean **IsDate** (*date/string* **$value**)

- *date/string* ***$value***

Example

```
if (MySQL::IsDate("January 1, 2000")) {
   echo "valid date";
}
```

Kill

Stop executing (die/exit) and show the last MySQL error message

- **access:** public

void **Kill** ([*mixed* **$message** = ''])

Example:

```
//Stop executing the script and show the last error
$db->Kill();
```

MoveFirst

Seeks to the beginning of the records

- **return:** Returns TRUE on success or FALSE on error
- **access:** public

boolean **MoveFirst** ()

Example:

```
$db->MoveFirst();
while (! $db->EndOfSeek()) {
   $row = $db->Row();
   echo $row->ColumnName1 . " " . $row->ColumnName2 . "\n";
}
```

MoveLast

Seeks to the end of the records

- **return:** Returns TRUE on success or FALSE on error
- **access:** public

boolean **MoveLast** ()

Example:

```
$db->MoveLast();
```

Open

Connect to specified MySQL server

- **return:** Returns TRUE on success or FALSE on error
- **access:** public

boolean **Open** ([*string* **$database** = ""], [*string* **$server** = ""], [*string* **$username** = ""], [*string* **$password** = ""], [*string* **$charset** = ""], [*boolean* **$pcon** = false])

- *string* ***$database****: (Optional) Database name*
- *string* ***$server****: (Optional) Host address*
- *string* ***$username****: (Optional) User name*
- *string* ***$password****: (Optional) Password*
- *string* ***$charset****: (Optional) Character set*
- *boolean* ***$pcon****: (Optional) Persistant connection*

Example

```
if (! $db->Open("MyDatabase", "localhost", "user", "password")) {
   $db->Kill();
}
```

Query

Executes the given SQL query and returns the records

- **return:** 'mysql result' resource object containing the records on SELECT, SHOW, DESCRIBE or EXPLAIN queries and returns; TRUE or FALSE for all others i.e. UPDATE, DELETE, DROP AND FALSE on all errors (setting the local Error message)
- **access:** public

object PHP **Query** (*string* **$sql**)

- *string **$sql**: The query string should not end with a semicolon*

Example:

```
if (! $db->Query("SELECT * FROM Table")) echo $db->Kill();
```

QueryArray (line 1017)

Executes the given SQL query and returns a multi-dimensional array

- **return:** A multi-dimensional array containing all the data returned from the query or FALSE on all errors
- **access:** public

array **QueryArray** (*string* **$sql**, [*integer* **$resultType** = MYSQL_BOTH])

- *string **$sql**: The query string should not end with a semicolon*
- *integer **$resultType**: (Optional) The type of array Values can be: MYSQL_ASSOC, MYSQL_NUM, MYSQL_BOTH*

QuerySingleRow (line 1033)

Executes the given SQL query and returns only one (the first) row

- **return:** resource object containing the first row or FALSE if no row is returned from the query
- **access:** public

object PHP **QuerySingleRow** (*string* **$sql**)

- *string **$sql**: The query string should not end with a semicolon*

QuerySingleRowArray (line 1051)

Executes the given SQL query and returns the first row as an array

- **return:** An array containing the first row or FALSE if no row is returned from the query
- **access:** public

array **QuerySingleRowArray** (*string* **$sql**, [*integer* **$resultType** = MYSQL_BOTH])

- *string **$sql**: The query string should not end with a semicolon*
- *integer **$resultType**: (Optional) The type of array Values can be: MYSQL_ASSOC, MYSQL_NUM, MYSQL_BOTH*

QuerySingleValue (line 1067)

Executes a query and returns a single value. If more than one row is returned, only the first value in the first column is returned.

- **return:** The value returned or FALSE if no value
- **access:** public

mixed **QuerySingleValue** (*string* **$sql**)

- *string **$sql**: The query string should not end with a semicolon*

QueryTimed

Executes the given SQL query, measures it, and saves the total duration in

microseconds

- **return:** 'mysql result' resource object containing the records on SELECT, SHOW, DESCRIBE or EXPLAIN queries and returns TRUE or FALSE for all others i.e. UPDATE, DELETE, DROP
- **access:** public

object PHP **QueryTimed** (*string* **$sql**)

- *string **$sql**: The query string should not end with a semicolon*

Example

```
$db->QueryTimed("SELECT * FROM MyTable");
echo "Query took " . $db->TimerDuration() . " microseconds";
```

Records

Returns the records from the last query

- **return:** 'mysql result' resource object containing the records for the last query executed
- **access:** public

object PHP **Records** ()

Example:

```
$records = $db->Records();
```

RecordsArray

Returns all records from last query and returns contents as array or FALSE on error

- **return:** in array form
- **access:** public

Records **RecordsArray** ([*integer* **$resultType** = MYSQL_BOTH])

- *integer **$resultType**: (Optional) The type of array Values can be: MYSQL_ASSOC,*

MYSQL_NUM, MYSQL_BOTH

Example

```
$myArray = $db->RecordsArray(MYSQL_ASSOC);
```

Release

Frees memory used by the query results and returns the function result

- **return:** Returns TRUE on success or FALSE on failure
- **access:** public

boolean **Release** ()

Example:

```
$db->Release();
```

Row

Reads the current row and returns contents as a PHP object or returns false on error

- **return:** object or FALSE on error
- **access:** public

object PHP **Row** ([*integer* **$optional_row_number** = null])

- *integer* ***$optional_row_number****: (Optional) Use to specify a row*

Example:

```
$db->MoveFirst();
while (! $db->EndOfSeek()) {
   $row = $db->Row();
   echo $row->ColumnName1 . " " . $row->ColumnName2 . "\n";
}
```

RowArray

Reads the current row and returns contents as an array or returns false on error

- **return:** Array that corresponds to fetched row or FALSE if no rows
- **access:** public

array **RowArray** ([*integer* **$optional_row_number** = null], [*integer* **$resultType** = MYSQL_BOTH])

- *integer* ***$optional_row_number****: (Optional) Use to specify a row*
- *integer* ***$resultType****: (Optional) The type of array Values can be: MYSQL_ASSOC, MYSQL_NUM, MYSQL_BOTH*

Example:

```
for ($index = 0; $index < $db->RowCount(); $index++) {
   $val = $db->RowArray($index);
}
```

RowCount

Returns the last query row count

- **return:** Row count or FALSE on error
- **access:** public

integer **RowCount** ()

Example:

```
$db->Query("SELECT * FROM Customer");
echo "Row Count: " . $db->RowCount();
```

Seek

Sets the internal database pointer to the specified row number and returns the result

- **return:** row as PHP object
- **access:** public

object Fetched **Seek** (*integer* **$row_number**)

- *integer* ***$row_number****: Row number*

Example:

```
$db->Seek(0); //Move to the first record
```

SeekPosition

Returns the current cursor row location

- **return:** Current row number
- **access:** public

integer **SeekPosition** ()

Example:

```
echo "Current Row Cursor : " . $db->GetSeekPosition();
```

SelectDatabase

Selects a different database and character set

- **return:** Returns TRUE on success or FALSE on error
- **access:** public

boolean **SelectDatabase** (*string* **$database**, [*string* **$charset** = ""])

- *string* ***$database****: Database name*
- *string* ***$charset****: (Optional) Character set (i.e. 'utf8')*

Example:

```
$db->SelectDatabase("DatabaseName");
```

SelectRows

Gets rows in a table based on a WHERE filter

- **return:** Returns records on success or FALSE on error
- **access:** public

boolean **SelectRows** (*string* **$tableName**, [*array* **$whereArray** = null], [*array/string* **$columns** = null], [*array/string* **$sortColumns** = null], [*boolean* **$sortAscending** = true], [*integer/string* **$limit** = null])

- *string **$tableName**: The name of the table*
- *array **$whereArray**: (Optional) An associative array containing the column names as keys and values as data. The values must be SQL ready (i.e. quotes around strings, formatted dates, ect)*
- *array/string **$columns**: (Optional) The column or list of columns to select*
- *array/string **$sortColumns**: (Optional) Column or list of columns to sort by*
- *boolean **$sortAscending**: (Optional) TRUE for ascending; FALSE for descending This only works if $sortColumns are specified*
- *integer/string **$limit**: (Optional) The limit of rows to return*

Example

```
// $arrayVariable["column name"] = formatted SQL value
$filter["Color"] = MySQL::SQLValue("Violet");
$filter["Age"]   = MySQL::SQLValue(777, MySQL::SQLVALUE_NUMBER);

// Execute the select
$result = $db->SelectRows("MyTable", $filter);

// If we have an error
if (! $result) {

   // Show the error and kill the script
   $db->Kill();

}
```

SelectTable (line 1229)

Retrieves all rows in a specified table

- **return:** Returns records on success or FALSE on error
- **access:** public

boolean **SelectTable** (*string* **$tableName**)

- *string **$tableName**: The name of the table*

SQLBooleanValue

[STATIC] Converts a boolean into a formatted TRUE or FALSE value of choice

- **return:** SQL formatted value of the specified data type
- **access:** public
- **static**

string **SQLBooleanValue** (*any* **$value**, *any* **$trueValue**, *any* **$falseValue**, [*string* **$datatype** = 'string'])

- *any **$value**: value to analyze for TRUE or FALSE*
- *any **$trueValue**: value to use if TRUE*
- *any **$falseValue**: value to use if FALSE*
- *string **$datatype**: Use SQLVALUE constants or the strings: string, text, varchar, char, boolean, bool, Y-N, T-F, bit, date, datetime, time, integer, int, number, double, float*

Example:

```
echo MySQL::SQLBooleanValue(false, "1", "0", MySQL::SQLVALUE_NUMBER);
echo MySQL::SQLBooleanValue($test, "Jan 1, 2007 ", "2007/06/01", MySQL::
SQLVALUE_DATE);
echo MySQL::SQLBooleanValue("ON", "Ya", "Nope");
echo MySQL::SQLBooleanValue(1, '+', '-');
```

SQLFix

[STATIC] Returns string suitable for SQL
Depreciated - instead use SQLValue($value, "text")

- **return:** SQL formatted value
- **access:** public
- **static**

string **SQLFix** (*string* **$value**)

- *string* ***$value***

Example:

```
$value = MySQL::SQLFix("\hello\ /world/");
echo $value . "\n" . MySQL::SQLUnfix($value);
```

SQLUnfix

[STATIC] Returns MySQL string as normal string
Depreciated

- **access:** public
- **static**

string **SQLUnfix** (*string* **$value**)

- *string* ***$value***

Example:

```
$value = MySQL::SQLFix("\hello\ /world/");
echo $value . "\n" . MySQL::SQLUnfix($value);
```

SQLValue

[STATIC] Formats any value into a string suitable for SQL statements (NOTE: Also supports data types returned from the gettype function)

- **access:** public
- **static**

string **SQLValue** (*any* **$value**, [*string* **$datatype** = SQLVALUE_TEXT])

- *any **$value**: Any value of any type to be formatted to SQL*
- *string **$datatype**: Use SQLVALUE constants or the strings: string, text, varchar, char, boolean, bool, Y-N, T-F, bit, date, datetime, time, integer, int, number, double, float*

Example:

```
echo MySQL::SQLValue("it's a string", "text");
$sql = "SELECT * FROM Table WHERE Field1 = " . MySQL::SQLValue("123", MySQL::SQLVALUE_NUMBER);
$sql = "UPDATE Table SET Field1 = " . MySQL::SQLValue("July 4, 2007", MySQL::SQLVALUE_DATE);
```

TimerDuration

Returns last measured duration (time between TimerStart and TimerStop)

- **return:** Microseconds elapsed
- **access:** public

Float **TimerDuration** ([*integer* **$decimals** = 4])

- *integer **$decimals**: (Optional) The number of decimal places to show*

Example:

```
$db->TimerStart();
// Do something or run some queries
$db->TimerStop();
echo $db->TimerDuration(2) . " microseconds";
```

TimerStart

Starts time measurement (in microseconds)

- **access:** public

void **TimerStart** ()

Example:

```
$db->TimerStart();
// Do something or run some queries
$db->TimerStop();
echo $db->TimerDuration() . " microseconds";
```

TimerStop

Stops time measurement (in microseconds)

- **access:** public

void **TimerStop** ()

Example:

```
$db->TimerStart();
// Do something or run some queries
$db->TimerStop();
echo $db->TimerDuration() . " microseconds";
```

TransactionBegin

Starts a transaction

- **return:** Returns TRUE on success or FALSE on error
- **access:** public

boolean **TransactionBegin** ()

Example:

```
$sql = "INSERT INTO MyTable (Field1, Field2) Values ('abc', 123)";
$db->TransactionBegin();
if ($db->Query($sql)) {
   $db->TransactionEnd();
   echo "Last ID inserted was: " . $db->GetLastInsertID();
} else {
   $db->TransactionRollback();
   echo "Query Failed";
}
```

TransactionEnd

Ends a transaction and commits the queries

- **return:** Returns TRUE on success or FALSE on error
- **access:** public

boolean **TransactionEnd** ()

Example:

```
$sql = "INSERT INTO MyTable (Field1, Field2) Values ('abc', 123)";
$db->TransactionBegin();
if ($db->Query($sql)) {
   $db->TransactionEnd();
   echo "Last ID inserted was: " . $db->GetLastInsertID();
} else {
   $db->TransactionRollback();
   echo "Query Failed";
}
```

TransactionRollback

Rolls the transaction back

- **return:** Returns TRUE on success or FALSE on failure
- **access:** public

boolean **TransactionRollback** ()

Example:

```
$sql = "INSERT INTO MyTable (Field1, Field2) Values ('abc', 123)";
$db->TransactionBegin();
if ($db->Query($sql)) {
   $db->TransactionEnd();
   echo "Last ID inserted was: " . $db->GetLastInsertID();
} else {
   $db->TransactionRollback();
   echo "Query Failed";
}
```

TruncateTable (line 1503)

Truncates a table removing all data

- **return:** Returns TRUE on success or FALSE on error

boolean **TruncateTable** (*string* **$tableName**)

- *string* ***$tableName****: The name of the table*

UpdateRows

Updates a row in a table from the connected database

- **return:** Returns TRUE on success or FALSE on error
- **access:** public

boolean **UpdateRows** (*string* **$tableName**, *array* **$valuesArray**, [*array* **$whereArray** = null])

- *string* ***$tableName****: The name of the table*

- *array **$valuesArray**: An associative array containing the column names as keys and values as data. The values must be SQL ready (i.e. quotes around strings, formatted dates, ect)*
- *array **$whereArray**: (Optional) An associative array containing the column names as keys and values as data. The values must be SQL ready (i.e. quotes around strings, formatted dates, ect). If not specified then all values in the table are updated.*

```
// Create an array that holds the update information
// $arrayVariable["column name"] = formatted SQL value
$update["Name"] = MySQL::SQLValue("Bob");
$update["Age"]  = MySQL::SQLValue(25, MySQL::SQLVALUE_NUMBER);

// Execute the update where the ID is 1
if (! $db->UpdateRows("test", $values, array("id" => 1))) $db->Kill
```

Class Constants (Use with **SQLValue()** method)

Description | Constants (*details*) Vars (details) | Methods (details)

SQLVALUE_BIT = "bit"

SQLVALUE_BOOLEAN = "boolean"

SQLVALUE_DATE = "date"

SQLVALUE_DATETIME = "datetime"

SQLVALUE_NUMBER = "number"

SQLVALUE_TEXT = "text"

SQLVALUE_TIME = "time"

SQLVALUE_T_F = "t-f"

SQLVALUE_Y_N = "y-n"