PHP API Reference

This appendix describes the application programming interface for writing PHP scripts that use the PHP Data Objects (PDO) database-access extension to interface with MySQL. The API consists of a set of classes and methods for communicating with MySQL servers and accessing databases.

PDO works with PHP 5.0 and up, but this appendix assumes a minimum of PHP 5.1 because that is when PDO was first bundled with PHP. See `http://www.php.net/pdo` for more information. If you need to install PHP, see Appendix A, "Software Required to Use This Book."

The examples in this appendix are only brief code fragments. For complete scripts and instructions for writing them, see Chapter 9, "Writing MySQL Programs Using PHP." The manual for PHP itself is available at the PHP Web site, `http://www.php.net`.

I.1 Writing PHP Scripts

PHP scripts are plain text files that contain a mixture of PHP code and non-PHP content such as HTML. PHP interprets the script to produce a Web page to be sent as output to the client. The non-PHP content is copied to the output without interpretation. PHP code is interpreted and replaced by whatever output the code produces.

PHP begins interpreting a file in text copy mode. To switch into and out of PHP code mode, use special tags that signify the beginning and end of PHP code. PHP understands four types of tags, although some of them must be explicitly enabled if you want to use them. One way to do this is by turning them on in the PHP initialization file, `php.ini`. The location of this file is system dependent; on many Unix systems, it's found in `/etc` or `/usr/local/lib`. On Windows, look in the PHP installation directory.

PHP understands the following tag styles:

- The default style uses `<?php` and `?>` tags:

```
<?php print ("Hello, world."); ?>
```

- Short-open-tag style uses `<?` and `?>` tags:

  ```
  <? print ("Hello, world."); ?>
  ```

  This style also supports `<?=` and `?>` tags as a shortcut for displaying the result of an expression without using a print statement:

  ```
  <?= "Hello, world." ?>
  ```

  Short tags can be enabled with a directive in the PHP initialization file:

  ```
  short_open_tag = On;
  ```

- Active Server Page-compatible style uses `<%` and `%>` tags:

  ```
  <% print ("Hello, world."); %>
  ```

  This style also supports `<%=` and `%>` tags as a shortcut for displaying the result of an expression without using a print statement:

  ```
  <%= "Hello, world." %>
  ```

  ASP-style tags can be enabled with a directive in the PHP initialization file:

  ```
  asp_tags = On;
  ```

- If you use an HTML editor that doesn't understand the other tags, you can use `<script>` and `</script>` tags:

  ```
  <script language="php"> print ("Hello, world."); </script>
  ```

Short tags and ASP-style tages are not portable; you cannot assume that a particular PHP installation will have them enabled.

I.2 PDO Classes

This appendix discusses the following classes from the PDO extension:

- `PDO` is the primary class. The class constructor is used for connecting to the database server. It returns a database-handle object that has methods for further interaction with the server.
- `PDOStatement` is the statement-handle class, returned by the `query()` and `prepare()` methods of `PDO` objects. A statement handle provides access to a statement result, such as statement metadata and result set contents.
- `PDOException` is the PDO error class. Objects of this class support methods for obtaining diagnostic information when an exception is raised due to occurrence of a PDO error.

I.3 PDO Methods

The following descriptions discuss available PDO methods, organized by the class with which they are associated. Certain object names recur throughout the method descriptions in this appendix and have the following conventional meanings:

- Database handle methods are called using a `$dbh` object, which is obtained by calling the `PDO` class constructor, `new PDO()`.
- Statement handle methods are called using a `$sth` object, which is returned by `$dbh->prepare()` or `$dbh->query()`.
- Exception objects are denoted by `$e`.

The method descriptions indicate data types for return values and parameters. A type of `mixed` indicates that a value might have different data types depending on how the value is used.

Many methods return a value that indicates success or failure. This value is relevant if PDO exceptions are not enabled, and should be tested to determine method outcome. If PDO exceptions are enabled, method errors cause PDO to raise a `PDOException`, which can be caught by using a `try/catch` construct. (See Section I.3.3, "`PDOException` Object Methods.")

Square brackets (`[]`) in syntax descriptions indicate optional parameters. When an optional parameter is followed by `= value`, it indicates that if the parameter is omitted from a method call, *`value`* is its default value.

The examples print messages and query results as plain text for the most part. This is done to make the code easier to read. However, for scripts intended for execution in a Web environment, you generally should encode output with `htmlspecialchars()` if it may contain characters that are special in HTML, such as '`<`', '`>`', or '`&`'.

In the descriptions that follow, the term "`SELECT` statement" should be taken to mean a `SELECT` statement or any other statement that returns rows, such as `DESCRIBE`, `EXPLAIN`, or `SHOW`.

I.3.1 `PDO` Class Methods

The `PDO` class includes methods for operations such as connecting to the database server, preparing and executing SQL statements, and setting or getting connection attributes.

- ```
  PDO
  __construct (string $dsn
               [, string $username
               [, string $password
               [, array $options]]])
  ```
  ```

This is the PDO constructor, which is executed when you invoke `new PDO()`. The constructor attempts to connect to a database server and returns an object representing a database handle if the attempt is successful. PHP raises a `PDOException` if an error occurs:

```
try
{
  $dbh = new PDO("mysql:host=localhost;dbname=sampdb", "sampadm", "secret");
}
catch (PDOException $e)
{
  die ($e->getMessage () . "\n");
}
```

To close the connection, set the database handle to `NULL`:

```
$dbh = NULL;
```

The `$dsn` argument represents the data source name (DSN). The DSN can take several forms:

- A driver DSN begins with a driver name and a colon, followed by optional driver-specific parameters. For MySQL, a driver DSN looks like this:

  `mysql:host=`*`host_name`*`;dbname=`*`db_name`*

  The `host` and `dbname` parameters indicate the host where the MySQL server is running and the database to select as the default database. The default `host` value is `localhost`. No default database is selected if `dbname` is omitted. Other possible parameters are `port` to specify the TCP/IP port number, `unix_socket` to specify the Unix socket file pathname, and (as of PHP 5.3.6) `charset` to specify the connection character set. If you use `unix_socket`, do not use `host` or `port`.

- A URI DSN begins with `uri:` followed by a URI that specifies the location of a file that contains a driver DSN. The URI can be local or remote. A local URI looks like this:

  `uri:file:///usr/local/lib/my-dsn-file`

- An alias DSN is a name *`XXX`* that associates with a configuration parameter of `pdo.dsn.`*`XXX`* in the `php.ini` file. For example, an alias of `sampdb` associates with a configuration parameter of `pdo.dsn.sampdb`, and the value of that parameter in `php.ini` should be a driver DSN.

The `$username` and `$password` arguments, if given, are the username and password of the MySQL account to use.

The `$options` array, if given, provides additional connection options that are not specified in the other arguments. Some of the options shown here are specific to the MySQL driver. Others are generic and may be supported by other drivers. For integer-valued options that turn behaviors on or off, pass 1 or 0 to enable or disable them.

- `PDO::ATTR_AUTOCOMMIT` (integer value; default enabled)

  Enable or disable autocommit mode.

- `PDO::ATTR_PERSISTENT` (integer value; default disabled)

  Enable or disable use of a persistent connection.

- `PDO::ATTR_TIMEOUT` (integer value; default 300)

  For MySQL, the connection timeout in seconds. For other database systems, this attribute may have a different meaning.

- `PDO::MYSQL_ATTR_COMPRESS` (integer value; default 0)

  Requests use of the compressed client/server communication protocol if the client and server both support it.

- `PDO::MYSQL_ATTR_DIRECT_QUERY`, `PDO::ATTR_EMULATE_PREPARES` (integer value; default enabled)

  Enable or disable use of direct statements. With direct statements, placeholders are emulated on the client side before sending queries to the server.

- `PDO::MYSQL_ATTR_FOUND_ROWS` (integer value; default 0)

  The type of row count to return for `UPDATE` statements. By default, the server returns the number of rows changed. Setting this attribute to 1 causes the server to return the number of rows matched.

- `PDO::MYSQL_ATTR_INIT_COMMAND` (string value)

  A statement to execute after connecting to the MySQL server, and after any automatic reconnect.

- `PDO::MYSQL_ATTR_LOCAL_INFILE` (integer value; default disabled)

  Enable or disable `LOAD DATA LOCAL`. Note that the MySQL server might not support `LOCAL`, or PHP safe mode might be in effect. In either case, attempts to enable `LOCAL` will be ineffective.

- `PDO::MYSQL_ATTR_MAX_BUFFER_SIZE` (integer value; default 1MB)

  The maximum size in bytes for column values returned by PDO. Truncation occurs for longer values.

- `PDO::MYSQL_ATTR_READ_DEFAULT_FILE` (string value)

  An option file from which to read options rather than the default file or files.

- `PDO::MYSQL_ATTR_READ_DEFAULT_GROUP` (string value)

  The group for which to read options from any option files that are read.

- `PDO::MYSQL_ATTR_USE_BUFFERED_QUERY` (integer value; default enabled)

  Enable or disable buffering of query result sets on the client side. When disabled, rows are retrieved from the server one at a time.

- bool
  beginTransaction (void)

  Disables autocommit mode and starts a transaction. Returns TRUE for success or FALSE for failure. To end the transaction, call commit() to commit any changes or rollback() to cancel any changes.

```
try
{
  $dbh->beginTransaction ();  # start transaction
  $dbh->exec ($stmt1);        # execute statements
  $dbh->exec ($stmt2);
  $dbh->commit ();            # commit if successful
}
catch (PDOException $e)
{
  # roll back if unsuccessful, but use empty
  # exception handler to catch rollback failure
  print ($e->getMessage () . "\n");
  try
  {
    $dbh->rollback ();
  }
  catch (PDOException $e) { }
}
```

- bool
  commit (void)

  Commits the current transaction and restores the autocommit mode. Returns TRUE for success, FALSE for failure, or raises an exception if no transaction is active.

  For an example, see the description of beginTransaction().

- string
  errorCode (void)

  Returns a string containing the five-character SQLSTATE value for the most recent operation on the database handle. A return value equal to PDO::ERR_NONE ("00000") means "no error."

```
if (!($sth = $dbh->query ($stmt)))
{
  print ("The statement failed.\n");
  print ("errorCode: " . $dbh->errorCode () . "\n");
  print ("errorInfo: " . join (", ", $dbh->errorInfo ()) . "\n");
}
```

- array
  errorInfo (void)

  Returns a three-element array containing error information for the most recent operation on the database handle. The array values are the SQLSTATE value (the same value returned by errorCode()) and driver-specific error code and error message values. For MySQL, the driver-specific values are a numeric code and message string.

  If the handle operation succeeds, the return value may be a single-element array containing the SQLSTATE value PDO::ERR_NONE ("00000").

  For an example, see the description of errorCode().

- int
  exec (string $stmt)

  Executes the SQL statement passed in the argument and returns the number of affected rows. Returns FALSE or the empty string if an error occurs.

  ```
  $count = $dbh->exec ("DELETE FROM member WHERE member_id = 149");
  printf ("Number of rows deleted: %d\n", $count);
  ```

  Use exec() for statements such as INSERT or DELETE that modify database contents. For statements such as SELECT that produce a result set, use query() instead.

- mixed
  getAttribute (int $attr)

  Returns the value of the specified database-handle attribute, or raises an exception for failure.

  Section I.3.4, "PDO Constants," lists some of the available attributes that can be retrieved with getAttribute().

  ```
  printf ("Driver name: %s\n",
          $dbh->getAttribute (PDO::ATTR_DRIVER_NAME));
  printf ("Server info: %s\n",
          $dbh->getAttribute (PDO::ATTR_SERVER_INFO));
  printf ("Server version: %s\n",
          $dbh->getAttribute (PDO::ATTR_SERVER_VERSION));
  ```

- array
  getAvailableDrivers (void)

  Returns an array containing the names of the available PDO drivers.

  ```
  $drivers = $dbh->getAvailableDrivers ();
  printf ("Number of drivers available: %d\n", count ($drivers));
  print ("Driver names: " . join (" ", $drivers) . "\n");
  ```

`getAvailableDrivers()` can also be called as a static method without obtaining a database handle first:

```
$drivers = PDO::getAvailableDrivers ();
```

- `string`
  `inTransaction` `(void)`

  Returns `TRUE` if there is a transaction active, `FALSE` if not.

- `string`
  `lastInsertId` `([string $name])`

  Returns the most recently generated sequence number for the connection. The behavior is driver-specific. For MySQL, the value is that returned by the `mysql_insert_id()` C API function. For some drivers (not MySQL), the `$name` argument must be given to specify the name of the sequence object.

```
$dbh->exec ("INSERT INTO grade_event (date, category)
             VALUES('2012-11-01','T')");
printf ("New grade_event ID: %d\n", $dbh->lastInsertId ());
```

- `PDOStatement`
  `prepare` `(string $stmt`
  `[, array $options])`

  Prepares the SQL statement passed in the first argument and returns a `PDOStatement` statement handle to use for further operations on the statement, or `FALSE` if statement preparation fails. To execute the statement, invoke the statement handle's `execute()` method.

```
$sth = $dbh->prepare ("INSERT INTO absence (student_id, date)
                       VALUES (?, ?)");
$sth->execute (array (7, "2012-10-01"));
$sth->execute (array (18, "2012-10-03"));
```

  The statement may contain placeholders in either positional or named format. Data values should be bound to the placeholders before invoking `execute()`, or else passed as parameters to `execute()`. For additional examples, see the descriptions of `bindParam()` and `bindValue()` in Section I.3.2, "`PDOStatement` Object Methods."

  The `$options` array, if given, specifies key/value pairs for setting attributes of the statement handle produced by `prepare()`.

- `PDOStatement`
  `query` `(string $stmt`
  `[, fetch_mode_option] ...)`

  Executes the SQL statement passed in the first argument and returns a `PDOStatement` statement handle to use for accessing the result set, or `FALSE` if an error occurs.

```
$sth = $dbh->query ("SELECT last_name, first_name FROM president");
while ($row = $sth->fetch ())
  printf ("%s %s\n", $row[1], $row[0]);
```

Use `query()` for statements such as `SELECT` that produce a result set. For statements such as `INSERT` or `DELETE` that modify database contents, use `exec()` instead.

Any arguments following the first are treated as arguments to pass to `setFetchMode()` for the statement handle returned by `query()`. See the description of `setFetchMode()` for the permitted arguments. Alternatively, specify the fetch mode by calling `setFetchMode()` directly after `query()` returns, or by passing a mode to `fetch()`. The fetch mode determines the type of object returned by `fetch()`.

It is also possible to use the `PDOStatement` object as an iterator without calling `fetch()`:

```
foreach ($sth as $row)
  printf ("%s %s\n", $row[1], $row[0]);
```

- ```
  string
  quote (string $str
         [, int $param_type])
  ```

  Escapes any special characters in the string passed as the first argument (using the conventions required by the current driver), adds surrounding quotes, and returns the resulting string. Returns `FALSE` if the driver does not support this method.

  ```
  $quoted_val1 = $dbh->quote (13);
  $quoted_val2 = $dbh->quote ("it's a string");
  ```

  The second argument may be specified to indicate the data type of the first argument. The default is `PDO::PARAM_STR`. See Section I.3.4, "PDO Constants," for a list of parameter type values.

  `quote()` doesn't correctly handle `NULL` values; it returns a quoted empty string rather than an unquoted word `NULL`. If your data values might be `NULL`, you're probably better off to take the approach of using placeholders and binding data values to them. Then PDO properly handles any required special processing.

- ```
  bool
  rollback (void)
  ```

  Rolls back the current transaction and restores the autocommit mode. Returns `TRUE` for success, `FALSE` for failure, or raises an exception if no transaction is active.

  For an example, see the description of `beginTransaction()`.

- ```
  bool
  setAttribute (int $attr,
                mixed $value)
  ```

  Sets an attribute for the database handle. The first argument names the attribute and the second provides its value. Returns `TRUE` for success or `FALSE` for failure.
  ```

Section I.3.4, "PDO Constants," lists some of the attributes that can be set with `setAttribute()`.

```
$dbh->setAttribute (PDO::ATTR_ERRMODE, PDO::ERRMODE_WARNING);
$dbh->setAttribute (PDO::ATTR_AUTOCOMMIT, true);
```

I.3.2 `PDOStatement` Object Methods

A `PDOStatement` object represents a statement handle returned by the `query()` or `prepare()` database-handle methods. Statement handles have methods for operations such as executing statements, accessing statement metadata and result set contents, binding data values to prepared statements, and binding variables to result sets.

- ```
  bool
  bindColumn (mixed $column,
              mixed $var
              [, int $type
              [, int $len
              [, mixed $options]]])
  ```

  Binds a column of a result set to a PHP variable, so that fetching a row sets the variable to the column value for the row. (Fetch the rows using a fetch mode of `PDO::FETCH_BOUND`.) Returns `TRUE` for success or `FALSE` for failure.

  The `$column` value can be given as a column number (beginning with 1) or column name (in the lettercase returned by the driver). `$var` is the PHP variable to which column values should be bound for each row fetch operation.

  `$type` specifies the data type of the column. The default is `PDO::PARAM_STR`. See Section I.3.4, "PDO Constants," for a list of parameter type values.

  The `$len` and `$options` values are specified the same way as for `bindParam()`.

  ```
  $sth = $dbh->query ("SELECT last_name, first_name FROM president");
  $sth->bindColumn ("last_name", $l_name); # specify column by name
  $sth->bindColumn (2, $f_name);           # specify column by position
  while ($sth->fetch (PDO::FETCH_BOUND))
    printf ("%s %s\n", $f_name, $l_name);
  ```

- ```
  bool
  bindParam (mixed $column,
             mixed $var
             [, int $type
             [, int $len
             [, mixed $options]]])
  ```

  Binds a PHP variable to a placeholder in a prepared statement. Returns `TRUE` for success or `FALSE` for failure. To provide a value for the placeholder, assign it to the variable before calling `execute()`.

The `$column` value can be given as a placeholder number (beginning with 1) or a placeholder name in the statement string (a name preceded by a colon). `$var` is the PHP variable to be bound to the placeholder.

`$type` specifies the data type of the column. The default is `PDO::PARAM_STR`. See Section I.3.4, "PDO Constants," for a list of parameter type values. If a placeholder is associated with an `INOUT` stored procedure parameter, perform an OR operation on the type with `PDO::INPUT_OUTPUT` (for example, `PDO::PARAM_INT|PDO::INPUT_OUTPUT`).

`$len` indicates the length of the data type. If a placeholder is associated with an `OUT` stored procedure parameter, you should provide an explicit length.

`$options` provides data for the driver.

```
$sth = $dbh->prepare ("INSERT INTO absence (student_id, date)
                       VALUES (:id, :date)");
$sth->bindParam (":id", $student_id);
$sth->bindParam (":date", $date);
$student_id = 7;
$date = "2012-10-01";
$sth->execute ();
$student_id = 18;
$date = "2012-10-03";
$sth->execute ();
```

- ```
  bool
  bindValue (mixed $column,
             mixed $value
             [, int $type])
  ```

  Binds a value to a placeholder in a prepared statement. Returns `TRUE` for success or `FALSE` for failure. The value is used for the next call to `execute()`.

  The `$column` and `$type` values are specified the same way as for `bindParam()`.

  ```
  $sth = $dbh->prepare ("INSERT INTO absence (student_id, date)
                         VALUES (?, ?)");
  $sth->bindValue (1, 7);
  $sth->bindValue (2, "2012-10-01");
  $sth->execute ();
  $sth->bindValue (1, 18);
  $sth->bindValue (2, "2012-10-03");
  $sth->execute ();
  ```

- ```
  bool
  closeCursor (void)
  ```

  Releases resources associated with the statement. Returns `TRUE` for success or `FALSE` for failure. This method can be used if you want to execute a statement again but have not

fetched the entire result set currently associated with the statement handle. (For MySQL, this should not be necessary because the driver retrieves any unfetched part of the result set as necessary, but that might not be true for other drivers.)

- `int`
  `columnCount` `(void)`

  Returns the number of columns in the result set produced by executing a statement. This value is 0 if the statement has not been executed or did not produce a result set.

```
$sth = $dbh->query ("SELECT * FROM president");
printf ("Number of columns in result set: %d\n", $sth->columnCount ());
```

- `string`
  `errorCode` `(void)`

  This is similar to `errorCode()` for `PDO` objects but applies to operations on `PDOStatement` objects.

```
if (!$sth->execute ())
{
  print ("Could not execute statement.\n");
  print ("errorCode: " . $sth->errorCode () . "\n");
  print ("errorInfo: " . join (", ", $sth->errorInfo ()) . "\n");
}
```

- `array`
  `errorInfo` `(void)`

  This is similar to `errorInfo()` for `PDO` objects but applies to operations on `PDOStatement` objects.

  For an example, see the description of `errorCode()`.

- `bool`
  `execute` `([array $params])`

  Executes a prepared statement and returns `TRUE` for success or `FALSE` for failure.

  If the statement contains placeholders, either bind data values to them before invoking `execute()`, or else pass the data values as parameters to `execute()`. For examples, see the descriptions of `prepare()`, `bindParam()`, and `bindValue()`.

- `mixed`
  `fetch` `([int $fetch_mode`
  `[, int $cursor_orientation`
  `[, int $cursor_offset]]])`

  Returns the next row of the result set, or `FALSE` if there are no more rows. The row has the format determined by the statement handle's fetch mode, or by the `$fetch_mode` argument if present. The default fetch mode is `PDO::FETCH_BOTH` unless it has been

changed by calling `setFetchMode()` or the statement handle was obtained by a call to `$dbh->query()` for which a fetch mode was passed. Section I.3.4, "PDO Constants," lists some of the permitted fetch modes.

```
$sth = $dbh->query ("SELECT last_name, first_name FROM president");
while ($row = $sth->fetch ())
  printf ("%s %s\n", $row[1], $row[0]);
```

The `$cursor_orientation` and `$cursor_offset` arguments are used to control scrollable cursors. These two arguments do not apply to MySQL, which does not support scrollable cursors.

- ```
  array
  fetchAll ([int $fetch_mode
              [, int $col_num = 0
              [, array $constructor_args]]])
  ```

  Returns any remaining rows of the result set as an array of rows. The fetch mode for the rows is determined the same way as for `fetch()`.

  ```
  $sth = $dbh->query ("SELECT last_name, first_name FROM president");
  $rows = $sth->fetchAll ();
  foreach ($rows as $row)
    printf ("%s %s\n", $row[1], $row[0]);
  ```

  If the fetch mode is `PDO::FETCH_COLUMN`, `fetchAll()` returns an array containing the values from the column of the result set specified by `$col_num`. Column numbers begin with 0.

  ```
  $sth = $dbh->query ("SELECT last_name, first_name FROM president");
  $first_names = $sth->fetchAll (PDO::FETCH_COLUMN, 1);
  print (join (", ", $first_names) . "\n");
  ```

  The `$constructor_args` argument is used for a custom class constructor. See the PHP manual for details.

- ```
  string
  fetchColumn ([int $col_num = 0])
  ```

  Returns one column from the next row of the result set, or `FALSE` if there are no more rows. `$col_num` specifies which column to return. Column numbers begin with 0. If you need to fetch multiple columns from each row, do not use this method.

  ```
  $sth = $dbh->query ("SELECT COUNT(*) FROM member");
  printf ("Number of members: %d\n", $sth->fetchColumn (0));
  ```

- ```
  mixed
  fetchObject ([string $class_name
                [, array $constructor_args]])
  ```
  ```

Returns the next row of the result set as a class instance, or FALSE if there are no more rows or an error occurs.

```
$sth = $dbh->query ("SELECT last_name, first_name FROM president");
while ($row = $sth->fetchObject ())
  printf ("%s %s\n", $row->first_name, $row->last_name);
```

$class_name is the name of the resulting class (stdClass if none is given). The $constructor_args argument is used for a custom class constructor. See the PHP manual for details.

- mixed
  getAttribute (int $attr)

  Returns the value of the specified statement-handle attribute, or raises an exception for failure. There are no MySQL-specific statement attributes, so the MySQL driver does not support getAttribute() as a statement method.

- mixed
  getColumnMeta (int $col_num)

  Returns an associative array containing metadata for the specified column of the result set, or FALSE if no such column exists. $col_num specifies which column to return. Column numbers begin with 0.

```
$sth = $dbh->query ("SELECT last_name, first_name FROM president");
var_dump ($sth->getColumnMeta (0));
var_dump ($sth->getColumnMeta (1));
```

The information returned by this method is driver dependent. At the time of writing, the array returned by the MySQL driver contains the values shown in the following table.

Name	Value
native_type	The PHP native type for the column value
flags	Flags describing the column attributes
table	The table containing the column (empty string for expressions)
name	The column name
len	The column length
precision	The column precision
pdo_type	The column type (corresponds to a PDO::PARAM_*XXX* value)

- bool
  nextRowset (void)

  Advances to the next rowset for a statement handle that has multiple rowsets. Returns TRUE for success or FALSE for failure.

Multiple rowsets can be produced by calling a stored procedure that produces multiple result sets, or by executing a statement string that contains multiple statements separated by semicolons. This is similar to processing multiple result sets using the C API (see Section 7.7, "Using Multiple-Statement Execution").

```
$sth = $dbh->query ("SELECT last_name, first_name FROM president LIMIT 5;
                     SELECT 1, 2, 3;
                     SHOW TABLES");
do
{
  $rowset = $sth->fetchAll (PDO::FETCH_NUM);
  if ($rowset)
  {
    $count = 0;
    foreach ($rowset as $row)
    {
      for ($i = 0; $i < sizeof ($row); $i++)
        print ($row[$i] . ($i < sizeof ($row) - 1 ? "," : "\n"));
      $count++;
    }
    printf ("Number of rows returned: %d\n\n", $count);
  }
} while ($sth->nextRowset ());
```

- `int`
  `rowCount` `(void)`

  Returns the rows-affected count for the statement. Use this only with statements such as `INSERT` or `DELETE` that modify rows. To get a row count for statements such as `SELECT` that produce a result set, fetch the rows and count them because `rowCount()` is not guaranteed to be meaningful.

- `bool`
  `setAttribute` `(int $attr,`
  `mixed $value)`

  Sets an attribute for the statement handle. The first argument names the attribute and the second provides its value. Returns `TRUE` for success or `FALSE` for failure. There are no MySQL-specific statement attributes, so the MySQL driver does not support `setAttribute()` as a statement method.

- `bool`
  `setFetchMode` `(int $fetch_mode`
  `[, fetch_mode_option] ...)`

  Sets the row-fetching mode for the statement. Returns `TRUE` for success or `FALSE` for failure. The fetch mode affects how methods such as `fetch()` and `fetchAll()` return rows when invoked with no explicit fetch-mode argument.

```
$sth = $dbh->query ("SELECT last_name, first_name FROM president");
$sth->setFetchMode (PDO::FETCH_OBJ);
while ($row = $sth->fetch ())
  printf ("%s %s\n", $row->last_name, $row->first_name);
```

Section I.3.4, "PDO Constants," describes several of the fetch modes that may be passed for the `$fetch_mode` argument.

For some values of `$fetch_mode`, additional arguments may be passed to `setFetchMode()` to affect how row-fetching methods work:

- `setFetchMode (PDO::FETCH_COLUMN, int $col_num)`

  Return a single column from rows of the result set. See the description for `fetchColumn()`.

- `setFetchMode (PDO::FETCH_CLASS, string $class_name, array $constructor_args)`

  Return rows of the result set as a new class instance. See the description for `fetchObject()`.

- `setFetchMode (PDO::FETCH_INTO, object $object)`

  Return rows of the result set into an existing class instance, mapping result set columns onto properties of the object's class.

I.3.3 `PDOException` Object Methods

By default, PDO raises an exception only for the `PDO` constructor (that is, when you call `new PDO()` to connect to a database server), and other PDO methods indicate failure by their return value. If you enable PDO exceptions after connecting, the PDO extension instead raises exceptions when its methods fail. `PDOException` objects contain the error information provided as a result of such exceptions.

To enable PDO exceptions, use the database handle:

```
$dbh->setAttribute (PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
```

PDO supports three error modes:

- `PDO::ERRMODE_SILENT`: PDO does nothing other than set the error information. This is the default error mode.
- `PDO::ERRMODE_WARNING`: This is similar to silent mode, but PDO emits a warning message in addition to setting the error information.
- `PDO::ERRMODE_EXCEPTION`: PDO raises an exception after setting the error information.

If exceptions are enabled, information about errors becomes available that you can get using the `getCode()` and `getMessage()` methods of the exception object.

Exceptions terminate your script by default. To handle them yourself, use `try` and `catch`. In the `catch` block, you can access the exception's methods that return error information:

```
try
{
  $sth = $dbh->query ("SELECT * FROM no_such_table");
}
catch (PDOException $e)
{
  print ("getCode value: " . $e->getCode() . "\n");
  print ("getMessage value: " . $e->getMessage() . "\n");
}
```

- `integer`
  `getCode` `(void)`

  Returns a five-character SQLSTATE value containing the error code. A return value equal to `PDO::ERR_NONE` (`"00000"`) means "no error."

- `string`
  `getMessage` `(void)`

  Returns a string containing the error message.

I.3.4 PDO Constants

This section describes some of the constants that can be used with PDO methods, such as the `getAttribute()` and `setAttribute()` methods for database handles. The values shown are representative only. For a complete list, see the PDO section of the PHP manual.

General database-handle attributes:

- `PDO::ATTR_AUTOCOMMIT`

  The current autocommit mode.

- `PDO::ATTR_CLIENT_VERSION`

  A string describing the client library version.

- `PDO::ATTR_CONNECTION_STATUS`

  For MySQL, this indicates how the connection was made.

- `PDO::ATTR_DEFAULT_FETCH_MODE`

  The row-fetching mode. (Available as a database-handle attribute as of PHP 5.2.4.)

- `PDO::ATTR_DRIVER_NAME`

  The PDO driver name.

- `PDO::ATTR_ERRMODE`

  The error-handling mode. For descriptions of the permitted values, see Section I.3.3, "`PDOException` Object Methods."

- `PDO::ATTR_SERVER_INFO`

  A string providing some server activity information.

- `PDO::ATTR_SERVER_VERSION`

  A string describing the server version.

Fetch-mode values that control the form in which result set rows are fetched:

- `PDO::FETCH_ASSOC`

  Return an array with elements accessed by associative index.

- `PDO::FETCH_BOTH`

  Return an array with elements accessed by associative or numeric index.

- `PDO::FETCH_BOUND`

  Return row elements bound to PHP variables by preceding `bindColumn()` calls.

- `PDO::FETCH_CLASS`

  Return row elements into properties of a new class instance.

- `PDO::FETCH_INTO`

  Return row elements into properties of an existing class instance.

- `PDO::FETCH_NUM`

  Return an array with elements accessed by numeric index.

- `PDO::FETCH_OBJ`

  Return an object with elements accessed as properties.

Parameter-type values:

- `PDO::PARAM_BOOL`

  A boolean parameter.

- `PDO::PARAM_INT`

  An integer parameter.

- `PDO::PARAM_NULL`

  Indicates a SQL `NULL` value.

- `PDO::PARAM_STR`

  A string parameter.