



Moving XML Data using Object-oriented PHP

Learn what object-oriented PHP can do for you—from code reuse, to flexible applications that move XML data in and out of Oracle database

By Yuli Vasiliev

This eBook was designed and published by [Free PDF Ebooks](http://www.freepdfbooks.com).

For more free eBooks visit our Web site at <http://www.acrobatplanet.com>.

Introduction

As a programming language, PHP has many advantages, such as ease of use, quick development time, and very high performance. The question that really matters is “can you use PHP to write object-oriented applications?”

With PHP 5’s brand new object model, PHP 5 has become a language that leverages the power and flexibility of object-oriented programming in a number of useful ways. Like most object-oriented languages, PHP 5 allows the developer to take advantage of interfaces, abstract classes, private/public/protected access modifiers, static members and methods, exception handling, and other features that were not available in prior releases of PHP. Perhaps the most important thing to note about the object-oriented improvements of PHP 5 is that objects are now referenced by handle and not by value.

Object-oriented PHP thus gives us the ability to build more scalable applications, make effective reuse of code, and handle exceptions elegantly. Welcome news to many a PHP-and-Java developer is that PHP can integrate with Java—you can activate Java objects in your PHP code and then use those objects as if they were regular PHP objects.

With this background, we start off by discussing some of the object-oriented features available in PHP5. Moving on, the article then shows how you can use PHP to simplify the creation of XML documents, generate XML from SQL data, and finally, how to take advantage of Oracle XML DB functionality.

Table of Content

<u>Introduction.....</u>	<u>2</u>
<u>Table of Content.....</u>	<u>3</u>
<u>Building Blocks of Applications.....</u>	<u>4</u>
<u>On Functionality and Implementation.....</u>	<u>8</u>
<u>On Reusing Code.....</u>	<u>9</u>
<u>On Handling Exceptions.....</u>	<u>9</u>
<u>On Extending classes.....</u>	<u>14</u>
<u>On Interaction Between Objects.....</u>	<u>18</u>
<u>Taking Advantage of Oracle XML DB Functionality.....</u>	<u>23</u>

Building Blocks of Applications

As you probably already know, the fundamental building block in any object-oriented language is a structure called a *class*—a blueprint from which objects are created. As an analogy to help you think object-oriented is of a building made up of blocks where classes represent the blocks. It is important to note that all blocks in this case are exchangeable. In practice, this means that if you are not satisfied with the implementation of the class any longer, you can instead use a relevant class that has the same Application Programming Interface (API) but a different implementation.

Let's step through a simple example of swapping one class implementation for another. First, however, note that this example (and all other examples throughout the article) uses the tables in the SCOTT schema (password TIGER), so before trying any of the scripts in this article, you must create a PHP file that contains all the required authentication information for the connection. The samples throughout use a file called `ScottCred.php`, as shown here (note that your actual connection string will likely be different from mine):

```
<?php
//File: ScottCred.php
$user="scott";
$pswd="tiger";
$conn="(DESCRIPTION=
      (ADDRESS_LIST=
        (ADDRESS=(PROTOCOL=TCP) (HOST=localhost) (PORT=1521))
      )
      (CONNECT_DATA=(SID=orcl) (SERVER=DEDICATED))
)";
?>
```

[runin subhead] Defining the class

With that the connection string information available, let's turn to an example that illustrates how you can rewrite the implementation of a class without changing existing applications (clients) that employ this class. Consider the class (`dbConn4.php`) defined in Listing 1, which defines a simple object that can connect to and query a database

Listing 1: A simple connection [`dbConn4.php`]

```
<?php
//File: dbConn4.php
class dbConn4 {
    var $user;
    var $pswd;
    var $db;
    var $conn;
    var $query;
    function dbConn4($user, $pswd, $db)
    {
        $this->user = $user;
```

```

        $this->pswd = $pswd;
        $this->db = $db;
        $this->ConnToDb();
    }
    function ConnToDb()
    {
        if(!$this->conn = @OCILogon($this->user, $this->pswd, $this->db))
        {
            $err = OCIError();
            trigger_error('Could not establish a connection: ' . $err['message']);
        }
    }
    function query($sql)
    {
        if(!$this->query = @OCIParse($this->conn, $sql)) {
            $err = OCIError($this->conn);
            trigger_error('Failed to parse SQL query: ' . $err['message']);
            return false;
        }
        else if(!@OCIExecute($this->query)) {
            $err = OCIError($this->query);
            trigger_error('Failed to execute SQL query: ' . $err['message']);
            return false;
        }
        return true;
    }
    function fetch()
    {
        if(!@OCIFetch($this->query)){
            return false;
        }
        return $this->query;
    }
}
?>

```

Once written, the `dbConn4` class can be used in applications to establish a connection to an Oracle database and then issue queries against it as needed.

[run-in subhead] Testing the connect class

To see the class defined in Listing 1 in action, you might use a very simple PHP script much like the `select.php` script shown here:

```

<?php

//File: select.php
require_once 'dbConn4.php';
require_once 'ScottCred.php';

```

```

$db = new dbConn4($user, $pswd, $conn);
$sql="SELECT ENAME FROM EMP";
if($db->query($sql)){
    print 'Employee Names: ' . '<br />';

    while ($row = $db->fetch()) {

        print oci_result($row,'ENAME') . '<br />';
    }
}
?>

```

When you execute the `select.php` script (assuming you've setup both `ScottCred.php` and `dbConn4.php`), the script outputs the names of employees from the `employee` table under the `SCOTT` demonstration schema.

Looking through the `select.php` script, you'll notice that using classes has resulted in compact, readable code. The `select.php` script makes use of the same PHP commands that you use to include the contents of one script in another; in the case of `select.php`, we're making the script aware of the class that we want to use. These commands are:

- [bullet]`include`—includes the script or class file; if the script or class file isn't found, a warning is issued, but the calling script continues to execute.

- [bullet]`require`—requires that the script or class file be included; if not found, the calling script terminates with a fatal error. Use `require` if the script or classfile is crucial to your application.

The suffix `_once` can be added to either of these to indicate that a single instance only of the script or object should be included or required. This suffix comes in handy, especially when you are developing complex applications that employ scripts, which in turn may include other scripts.

Let's look at how we've made use of these commands in `select.php` script, and highlight some of the processing that occurs:

First of all, to make `select.php` aware of the `dbConn4` class, we include a `require_once` statement in the file, as in:

```
require_once 'dbConn4.php';
```

Similarly, you require the credentials that are needed to establish a database connection:

```
require_once 'ScottCred.php';
```

Next, you instantiate an object of `dbConn4` class, passing in the values from `ScottCred.php` as parameters to the constructor to initialize, as follows:

```
$db = new dbConn4($user, $pswd, $conn);
```

Once an instance of the class is created, you can call the methods of that instance as needed:

```
$db->query($sql)
```

[run-in subhead] Putting PHP5 to work

Turning back to the dbConn4 class, you may notice that it was written for PHP 4. Of course, it still can be used in new applications written for PHP 5. However, to take advantage of the new object-oriented features available in PHP 5, you might want to rewrite this class as shown in Listing 2:

Listing 2: A simple connection class [dbConn5.php]

```
<?php
//File: dbConn5.php
class dbConn5 {
    private $user;
    private $pswd;
    private $db;
    private $conn;
    private $query;
    public function __construct($user, $pswd, $db)
    {
        $this->user = $user;
        $this->pswd = $pswd;
        $this->db = $db;
        $this->ConnToDb();
    }
    private function ConnToDb()
    {
        if(!$this->conn = oci_connect($this->user, $this->pswd, $this->db))
        {
            $err = oci_error();
            trigger_error('Could not establish a connection: ' . $err['message']);
        }
    }
    public function query($sql)
    {
        if(!$this->query = oci_parse($this->conn, $sql)) {
            $err = oci_error($this->conn);
            trigger_error('Failed to execute SQL query: ' . $err['message']);
            return false;
        }
        else if(!oci_execute($this->query)) {
            $err = oci_error($this->query);
            trigger_error('Failed to execute SQL query: ' . $err['message']);
            return false;
        }
        return true;
    }
    public function fetch()
    {
        if(oci_fetch($this->query)){
            return $this->query;
        }
    }
}
```

```

    }
    else {
        return false;
    }
}
}
?>

```

As you can see, the implementation of the class has been improved to conform to the new standards of PHP 5. For example, the `dbConn5.php` class (Listing 2) takes advantage of PHP 5's support for encapsulation, which it achieves (as with most other object-oriented languages) by means of access modifiers—`public`, `protected`, and `private`. The idea behind encapsulation is to enable the developer to design classes that reveal only the members and methods that are available for external use, while hiding internals and implementation details.

For example, the `ConnToDb()` method in the `dbConn5` class is declared with the `private` modifier because this method should be called only from inside the constructor, when a new instance of the class is initialized; therefore, there is no need to allow client code to access this method directly.

Since the implementation of the newly created `dbConn5` class is different from the one used in `dbConn4`, you may be asking yourself “Does that mean we need to rewrite the client code that uses the `dbConn4` class as well?” The answer is obvious: no. You don't need to rewrite client code that uses the `dbConn4` class since you have changed neither the interface of the class nor, more importantly, its functionality. To make the `select.php` script work with `dbConn5`, you need only replace `dbConn4` with `dbConn5` throughout the script.

On Functionality and Implementation

As you learned from the above example, you can easily replace one class with another in a calling script if both classes deliver the same API and the same functionality. In practical terms, this means that you can improve upon any class implementation used in existing applications—until the improvements change the API of the class or its functionality. (Of course, you can also add functionality by extending the API of a class with new public methods, in which case, you must ensure that the resulting new class still works with old clients, however.)

Sometimes, it's easy to rewrite a class implementation without changing its functionality (as we've just seen); at other times, it's not. For example, say you want to use the `oci_new_connect()` function instead of `oci_connect()`, in the `ConnToDb()` method of the `dbConn5` class (see Listing 2), so that multiple connections to the database each has its own connection object, enabling you to apply commits and rollbacks to the specified connection only. This functionality is markedly different than that provided by `oci_connect()`, which re-uses existing connections that have the same parameters, and assumes that commits and rollbacks are applied to all open transactions in the

page. A call to `oci_connect()` doesn't return a new connection object (as does `oci_new_connect()`), rather it returns the identifier of the already opened connection.

Therefore, if you make such a replacement in the `dbConn5` class—`oci_new_connect()` in place of `oci_connect()`—you will no longer be able to use the `dbConn5` class in any existing client of `dbConn4` that was designed to work with transactions, assuming that connections are shared at the page level.

As you can see, two classes that implement the same API don't necessarily automatically guarantee identical functionality. Thus, you must be careful whenever you replace one class with another in existing applications—be sure the functionality is what you want.

On Reusing Code

Once a class has been written and debugged, you can then reuse it over and over again in new projects. In "Building Blocks of Applications" [LINK], you learned how to use the `dbConn5` class in a script that selects data from the database. In fact, you are not limited to a `SELECT` operation and can use the `query()` method of the `dbConn5` class to perform any SQL statement against the database. For instance, you might use the following script to insert a row into the `dept` table under the `SCOTT` database schema:

```
<?php
//File: insert.php
require_once 'dbConn5.php';
require_once 'ScottCred.php';
$db = new dbConn5($user, $pswd, $conn);
$sql="INSERT INTO DEPT VALUES(50, 'Computing', 'Los Angeles)";
if($db->query($sql)){
    print 'data have been submitted';
}
else {
    print 'failed to submit data';
}
?>
```

Similarly, you might use the `query()` method to perform DDL operations against the database. It is important to note that the preceding example shows the simplest way in which you can take advantage of code reuse. It demonstrates how to reuse an existing class to solve a similar problem in another script. Later in this article, you will learn about other ways to achieve code reuse in object oriented PHP applications.

On Handling Exceptions

For handling errors, PHP 5 offers a new mechanism that is completely different from that in PHP 4. In PHP 5, you can create and throw an instance of the built-in `Exception` class in response to an error that has occurred in your object code. Throwing an exception terminates method execution and

makes the appropriate Exception instance available to the client code. Listing 3 illustrates a dbConn5e class that supports the PHP 5's new exception model:

Listing 3: A connection class that throws exceptions

```
<?php
//File: dbConn5e.php
class dbConn5e {
    private $user;
    private $pswd;
    private $db;
    private $conn;
    private $query;
    public function __construct($user, $pswd, $db)
    {
        $this->user = $user;
        $this->pswd = $pswd;
        $this->db = $db;
        $this->ConnToDb();
    }
    private function ConnToDb()
    {
        if(!$this->conn = oci_connect($this->user, $this->pswd, $this->db))
        {
            $err = oci_error();
            throw new Exception('Could not establish a connection: ' .
                $err['message']);
        }
    }
    public function query($sql)
    {
        if(!$this->query = oci_parse($this->conn, $sql)) {
            $err = oci_error($this->conn);
            throw new Exception('Failed to execute SQL query: ' .
                $err['message']);
        }
        else if(!oci_execute($this->query)) {
            $err = oci_error($this->query);
            throw new Exception('Failed to execute SQL query: ' .
                $err['message']);
        }
        return true;
    }
    public function fetch()
    {
        if(oci_fetch($this->query)){
            return $this->query;
        }
        else {
            return false;
        }
    }
}
```

```
}  
?>
```

As shown in Listing 3, the `dbConn5e` class doesn't use the `trigger_error()` function used in the `dbConn5` class (Listing 2). Instead, when something goes wrong, the `dbConn5e` class throws an exception using the `throw new Exception()` syntax.

As previously mentioned, when an exception is thrown, it becomes available to the client context. On the client side, you should place any code that might throw an exception inside of the `try` block. When you have defined the `try` block, you must then define at least one `catch` block, which will be used to handle thrown exceptions. So, client code no longer needs to check the return value of every method to see whether an error has occurred—the code within a `catch` block takes care of the thrown exception.

To test the `dbConn5e` class (Listing 3), here's a new client, the `select_e.php` script:

```
<?php  
//File: select_e.php  
require_once 'dbConn5e.php';  
require_once 'ScottCred.php';  
  
try {  
    $db = new dbConn5e($user, $pswd, $conn);  
    $sql="SELECT ENAME FROM EMP";  
    $db->query($sql);  
    print 'Employee Name: ' . '<br />';  
  
    while ($row = $db->fetch()) {  
        print oci_result($row,'ENAME') . '<br />';  
    }  
} catch (Exception $e) {  
    print $e->getMessage();  
    exit();  
}  
  
?>
```

As you can see, rather than checking `dbConn5`'s `query()` method for a return value of `true`, the `select_e.php` test script shown above wraps the code that might throw an exception in the `try` block, and the `catch` block defines how to handle thrown exceptions—in this case, simply printing the appropriate error message.

The `catch` block shown in the `select_e.php` script is fine for test applications; however, in a real-world application, you might want your application to distinguish between different error types in

the catch block. One way to do this is to pass the user-defined exception flag to the `Exception` constructor, as in:

```
        throw new Exception('Could not establish a connection: ' .
    $err['message'], self::CONNECTION_ERROR);
```

Listing 4 (`dbConn5e2.php`) illustrates how you can modify the `dbConn5e` class (Listing 3) so that the new `dbConn5e2` class can throw the exceptions that will differ by error types using these flags. Listing 4: A connection class that throws error [more robust error handling]

```
<?php
//File: dbConn5e2.php
class dbConn5e2 {
    private $user;
    private $pswd;
    private $db;
    private $conn;
    private $query;
    const CONNECTION_ERROR = 1;
    const SQLEXECUTION_ERROR = 2;
    function __construct($user, $pswd, $db)
    {
        $this->user = $user;
        $this->pswd = $pswd;
        $this->db = $db;
        $this->ConnToDb();
    }
    function ConnToDb()
    {
        if(!$this->conn = oci_connect($this->user, $this->pswd, $this->db))
        {
            $err = oci_error();
            throw new Exception('Could not establish a connection: ' .
    $err['message'], self::CONNECTION_ERROR);
        }
    }
    function query($sql)
    {
        if(!$this->query = oci_parse($this->conn, $sql)) {
            $err = oci_error($this->conn);
            throw new Exception('Failed to execute SQL query: ' .
    $err['message'], self::SQLEXECUTION_ERROR);
        }
        else if(!oci_execute($this->query)) {
            $err = oci_error($this->query);
            throw new Exception('Failed to execute SQL query: ' .
    $err['message'], self::SQLEXECUTION_ERROR);
        }
        return true;
    }
    function fetch()
```

```

    {
        if(oci_fetch($this->query)){
            return $this->query;
        }
        else {
            return false;
        }
    }
}
?>

```

Now, you can improve the `catch` block in the client code so that it recognizes different error categories. In the `select_e2.php` script, failure to connect to the database results in a fatal error, which means the script will terminate. On the other hand, failure to perform a query against the database leads to generating the appropriate warning message only—execution continues after the `catch` block.

```

<?php
//File: select_e2.php
require_once 'dbConn5e2.php';
require_once 'ScottCred.php';

try {
$db = new dbConn5e2($user, $pswd, $conn);
$sql="SELECT ENAME FROM EMP";
$db->query($sql);
print 'Employee Name: ' . '<br />';

while ($row = $db->fetch()) {

    print oci_result($row,'ENAME') . '<br />';
}
}
catch (Exception $e) {
    if ($e->getCode() == dbConn5e2::CONNECTION_ERROR) {
        die($e->getMessage());
    }
    else if ($e->getCode() == dbConn5e2::SQLEXECUTION_ERROR) {
        print $e->getMessage();
    }
}
//Continue execution
?>

```

Remember, just as in Java and other programming languages, exceptions are not necessarily always errors. Whether or not an exception represents an error is determined by the application that employs the class in which the exception occurred. In other words, if an exception has been thrown, it is up to the client code to decide what to do.

For example, an application might require a user to log in using a logon page. How would you like such an application to behave if a user mistypes when entering his credentials? Obviously, it would be a bad idea in such a case to throw an exception that will terminate the execution. Instead, you most probably might want the application to generate a warning message and then allow the user to enter his credentials again.

On Extending classes

As discussed earlier, a well-written piece of object oriented code can be reused in more than one application. In "On Reusing Code" [LINK], you saw how to reuse an existing class in another application. You can also gain the benefits of reusability by extending an existing class with additional functionality, specific to the application. Using this approach, you don't use the base class directly; instead, you create a new class that is an extension of the base class. A class can function as the base class for many specific classes unless it has been defined as `final`. In the object-oriented world, the ability to create new classes based on an existing class is called inheritance.

Despite the fact that the `multiple inheritance` is not supported in PHP 5, you can find extensibility very useful when developing complex object oriented applications since this allows you to use well-designed classes over and over as the basis for new classes.

In PHP 5, a class can inherit the functionality of an existing base class by using the `extends` keyword. Just as in Java and other programming languages, you can extend both user-defined and predefined PHP classes.

The following example illustrates how you can extend the predefined `DomDocument` class with a public method that is intended to simplify the process of creating XML documents.

```
<?php
//File: MyDomDocument.php
class MyDomDocument extends DomDocument{

    public function __construct($ver = "1.0", $encode = "UTF-8")
    {
        parent::__construct($ver, $encode);
    }
    public function addElement($node, $elemName, $elemVal="")
    {
        if(!$child = $this->createElement($elemName, $elemVal))
            throw new Exception('Could not create a new node ');
        if(!$child = $node->appendChild($child))
            throw new Exception('Could not append a new node ');
        return $child;
    }
}
?>
```

In the `MyDomDocument.php` code, you explicitly call the parent constructor within the overriding constructor. This is because, unlike most object-oriented languages, PHP doesn't implicitly invoke the parent constructor when executing an overriding constructor in the child class. So, if you define a constructor in a child class, you must explicitly call the constructor of the parent class within the new child constructor. Another important thing to note here is that all the parent members and methods defined as either protected or public are accessible within a child class.

Regardless of whether or not a child class overrides a public or protected parent method or member, you can still access it within that child class by using the `parent::` prefix. To client code, any method or member defined as public in the parent class will be accessible with an instance of the child class by default, unless this child class has overridden that parent method or member. (The ability of a class to override inherited members or methods is called `polymorphism`.) Methods and members defined as final in the parent class cannot be overridden in child classes.

Now let's put the `MyDomDocument` class to use:

```
<?php
//File: dom.php
require_once 'MyDomDocument.php';
try {
    //first, you create an instance of MyDomDocument
    $dom = new MyDomDocument();
    //now, you are ready to create the root of the XML document
    $root = $dom->addElement($dom, 'EMPLOYEE');
    //you can always add an attribute to the node if necessary
    $root->setAttribute('id', '1');
    //once the root element is created, you can add nested nodes as needed
    $emplno = $dom->addElement($root, 'EMPNO', '212');
    $ename = $dom->addElement($root, 'ENAME', 'John Jamison');
    $title = $dom->addElement($root, 'TITLE', 'Programmer');
    //finally, we output the XML document
    echo $dom->saveXML();
}
catch(Exception $e) {
    print $e->getMessage();
}
?>
```

As you can see, the `dom.php` script is straightforward—it creates an instance of `MyDomDocument` and populates an XML DOM structure. Notice that using the `MyDomDocument` class reduces the process of adding a new node to an XML document to a single line, thereby making the code involved in creating an XML document more readable. Running `dom.php` results in the following output:

```
<?xml version="1.0" encoding="UTF-8" ?>
<EMPLOYEE id="1">
  <EMPNO>212</EMPNO>
```

```

    <ENAME>John Jamison</ENAME>

    <TITLE>Programmer</TITLE>
</EMPLOYEE>

```

Unlike this simple example, with its hard-coded data in the test script, a real-world script most probably would retrieve data from a file or database, as shown in the `domXML.php`:

```
<?php
```

```

//File: domXML.php
require_once 'MyDomDocument.php';
require_once 'dbConn5e.php';
require_once 'ScottCred.php';
$i = 0;
try {
    $dom = new MyDomDocument();
    $root = $dom->addElement($dom, 'EMPLOYEES');
    $db = new dbConn5e($user, $pswd, $conn);
    $sql="SELECT EMPNO, ENAME, JOB FROM EMP";
    $db->query($sql);
    while ($row = $db->fetch()) {
        $empl = $dom->addElement($root, 'EMPLOYEE');
        $empl->setAttribute('id', ++$i);
        $empno = $dom->addElement($empl, 'EMPNO', oci_result($row, 'EMPNO'));
        $ename = $dom->addElement($empl, 'ENAME', oci_result($row, 'ENAME'));
        $title = $dom->addElement($empl, 'TITLE', oci_result($row, 'JOB'));
    }
    print $dom->saveXML();
}
catch(Exception $e) {
    print $e->getMessage();
}
?>

```

The `domXML.php` script uses the `MyDomDocument` class after first selecting relational data (using the `dbConn5e` class shown in Listing 3) from the database and then converting the retrieved SQL data to XML format. Assuming we're connected to the same database as in the earlier examples, the output generated from `domXML.php` is as follows:

```

<?xml version="1.0" encoding="UTF-8"?>
<EMPLOYEES>
  <EMPLOYEE id="1">
    <EMPNO>7369</EMPNO>
    <ENAME>SMITH</ENAME>
    <TITLE>CLERK</TITLE>
  </EMPLOYEE>

  ...
</EMPLOYEES>

```

[run-in subhead] Retrieving XML Tagged Data from Oracle

As you can see, the `xmlDOM.php` script simply generates XML from SQL data using methods of the `MyDomDocument` class. Alternatively, you can use SQL XML functions to accomplish the same general result, in which case you would want to use the `domTaggedXML.php` script to generate the XML data:

```
<?php
//File: domTaggedXML.php
require_once 'MyDomDocument.php';
require_once 'dbConn5e.php';
require_once 'ScottCred.php';
try {
    $dom = new MyDomDocument();
    $db = new dbConn5e($user, $pswd, $conn);
    $sql="SELECT (XMLELEMENT(\"EMPLOYEE\", XMLATTRIBUTES(rownum AS \"id\"),
XMLELEMENT(\"EMPNO\", empno), XMLELEMENT(\"ENAME\", ename),
XMLELEMENT(\"TITLE\", job))).getClobVal() as MYXML FROM EMP";
    $db->query($sql);
    $rows=$db->fetchAll();
    foreach ($rows['MYXML'] as $row) {
        $rslt=$rslt.$row;
    }
    $dom->loadXML("<EMPLOYEES>$rslt</EMPLOYEES>");
    print $dom->saveXML();
}
catch(Exception $e) {
    print $e->getMessage();
}
?>
```

The `domTaggedXML.php` script produces the same output as the `xmlDOM.php` script.

However, before you can run the `domTaggedXML.php` script, you have to add the `fetchAll()` public method to the `dbConn5e` class. This method should fetch all the rows of the result data and then return them as an array to the calling script. The `fetchAll()` method is implemented as follows:

```
public function fetchAll()
{
    if(oci_fetch_all($this->query, $results)){
        return $results;
    }
    else {
        return false;
    }
}
```

This another example of code reuse. All it needed was the simple addition of a method to the `dbConn5e` class to use that class in a script that demonstrates another way to generate XML from SQL data.

On Interaction Between Objects

Proceeding with the example, you can go one step further and save the XML content stored in the internal XML tree of the `MyDomDocument` instance to an Oracle database. You might want to extract the `employee` XML elements from the XML tree and save them as separate `employee` XML documents in the database. To do this, you first need to design a database schema to store the `employee` XML documents. The simplest way of doing this is to design the appropriate annotated XML schema for the `employee` XML documents and then register this XML schema with your Oracle database. As a result, Oracle automatically will create the required database schema for you.

To perform all the database operations presented in this section, you can use `SQL*Plus` or similar tool. To start with, you need to create a database user. To do this, logon as `SYSDBA` and then create a user, say `USR`. The next step is to grant the `RESOURCE` and `CONNECT` roles to the newly created user. Next, connect as `USR` and then execute the following PL/SQL code in order to register the annotated XML schema that is intended to validate `employee` XML documents.

Listing 5: PL/SQL code for XML DB setup

```
BEGIN

DBMS_XMLSCHEMA.registerSchema(
'http://localhost:8080/public/emp.xsd',
xmltype('<?xml version="1.0" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
          xmlns:xdb="http://xmlns.oracle.com/xdb"
          version="1.0" >
  <xs:element name="EMPLOYEE" type="EmpType"
             xdb:defaultTable="EMPLOYEE"/>
  <xs:complexType name="EmpType" xdb:SQLType="EMP_T">
    <xs:sequence>
      <xs:element name="EMPNO" type="EmpNoType" minOccurs="1"
                 xdb:SQLName="EMPNO"/>
      <xs:element name="ENAME" type="EnameType" xdb:SQLName="ENAME"/>
      <xs:element name="TITLE" type="TitleType"
                 xdb:SQLName="TITLE"/>
    </xs:sequence>
  </xs:complexType>
  <xs:simpleType name="EmpNoType">
    <xs:restriction base="xs:integer"/>
  </xs:simpleType>
  <xs:simpleType name="EnameType">
    <xs:restriction base="xs:string">
      <xs:minLength value="2"/>
      <xs:maxLength value="30"/>
    </xs:restriction>
  </xs:simpleType>
</xs:schema>

```

```

        </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="TitleType">
        <xs:restriction base="xs:string">
            <xs:minLength value="2"/>
            <xs:maxLength value="30"/>
        </xs:restriction>
    </xs:simpleType>
</xs:schema>'),
TRUE, TRUE, FALSE, TRUE

);

END;
/
PL/SQL procedure successfully completed.

```

As a result, Oracle automatically creates the `employee` table of `XMLType` that will be used to store employee XML documents conforming to the above XML schema. Before proceeding, you might want to define the `PRIMARY KEY` constraint on the `employee XMLType` table in order to prevent duplicate employee records. This can be done as follows:

```

ALTER TABLE EMPLOYEE

ADD constraint EMPNO_IS_PRIMARYKEY
PRIMARY KEY (xmldata."EMPNO");
Table altered.

```

Now that you have defined the `XMLType` storage for employee XML documents, turn back to the `MyDomDocument` class and add the following public method to that class:

```

public function insertSchemaBasedXML($tagName, $db, $table, $schema)
{
    try {
        $items = parent::getElementsByTagName($tagName);
        for ($i = 0; $i < $items->length; $i++) {
            $nodeXML = parent::saveXML($items->item($i)) . "\n";
            $db->query("INSERT INTO " . $table . " VALUES( XMLType('" . $nodeXML .
            "' ).createSchemaBasedXML('" . "$schema" . "')");
        }
    }
    catch (Exception $e) {
        throw $e;
    }
    print "Data have been submitted";
}

```

As this code shows, you use the `parent` keyword to call parent class's methods. Alternatively, you might specify the name of the parent class, in our case `DomDocument` instead of using the `parent` keyword. Perhaps the most interesting thing to note about the above code is that it shows a way of

how objects can interact with each other. In particular, this demonstrates how you can dynamically associate two objects by passing one object (actually you are passing only a reference to that object) as a parameter to a method of another object. The `insertSchemaBasedXML()` method takes an instance of `dbConn5` as a parameter and then invokes the `query()` method to perform the INSERT operation against the database.

With the `insertSchemaBasedXML()` method in place, you no longer need to worry about composing INSERT statements in order to insert schema based XML data into an Oracle database: `insertSchemaBasedXML()` will do it for you. When calling the `insertSchemaBasedXML()` method, you have to specify the following parameters:

- A tag name that will be used to search for appropriate elements within the internal XML tree of the `MyDomDocument` instance. Each of these elements will be used as an independent XML document.
- An instance of the `dbConn5e` class that will be used to interact with the database.
- The name of an XMLType table into which you are going to insert XML content.
- A registered XML schema that will be used to validate the inserted XML documents.

As a result, the work of inserting the generated `employee` XML documents into the database is reduced to a single line. This may be demonstrated as follows:

```
<?php
//File: domXML_db.php
require_once 'MyDomDocument.php';
require_once 'dbConn5e.php';
require_once 'ScottCred.php';
try {
    $dom = new MyDomDocument();
    $root = $dom->addElement($dom, 'EMPLOYEES');
    $db = new dbConn5e($user, $pswd, $conn);
    $sql="SELECT EMPNO, ENAME, JOB FROM EMP";
    $db->query($sql);
    while ($row = $db->fetch()) {
        $empl = $dom->addElement($root, 'EMPLOYEE');
        $emplno = $dom->addElement($empl, 'EMPNO', oci_result($row,'EMPNO'));
        $ename = $dom->addElement($empl, 'ENAME', oci_result($row,'ENAME'));
        $title = $dom->addElement($empl, 'TITLE', oci_result($row,'JOB'));
    }
    $db2 = new dbConn5e('usr', 'pswd', $conn);
    $dom->insertSchemaBasedXML('EMPLOYEE', $db2,'employee', 'http://localhost:
8080/public/emp.xsd');
}
catch(Exception $e) {
    print $e->getMessage();
}
}
```

?>

Now, to make sure everything goes as planned, you can issue the following query from SQL*Plus:

```
SELECT COUNT(*) FROM employee;  
COUNT(*)
```

14

The result demonstrates that 14 rows have been successfully moved from the `emp` table under the SCOTT demonstration schema into the `employee XMLType` table in the `USR` schema. Please note that if you try to run the above PHP script again, you'll get the following error message:

```
Failed to execute SQL query: ORA-00001: unique constraint  
(USR.EMPNO_IS_PRIMARYKEY) violated
```

Looking through the `insertSchemaBasedXML()` method, you may notice that it employs a `try-catch` statement to detect and respond to the errors that occur when inserting XML content into an XMLType table in the database. This may seem confusing at first, since all the examples discussed earlier in this article used a `try-catch` statement only in client code. In this example, however, using a `try-catch` statement within a class method makes sense because the code within the `try` block calls another class method, namely the `query()` method of the `dbConn5e` class, that itself can throw an exception.

Specifically, the `query()` method throws an exception when failing to perform the query against the database. So, if you omit the `try-catch` statement in the `insertSchemaBasedXML()` method, then an exception thrown in the `query()` method will be caught only in the calling script. This means that the `insertSchemaBasedXML()` method will not terminate when `query()` fails to insert a row (an XML document) into the specified XMLType table. Instead, it will try to insert the following row until it reaches the end of the list of the generated XML documents.

To prevent this behavior, you simply wrap the code calling the `query()` method in the `try` block and define a `catch` block that, in response to an exception, will throw a new exception, thereby terminating `insertSchemaBasedXML` execution when the first failure to insert a row occurs.

However, if you want the `insertSchemaBasedXML()` method to continue execution when an insert failure has occurred, thereby trying to insert the following row, then you can simply replace the line of code that throws an exception in the `catch` block with a line that just prints a warning message. In this case, the `catch` block in the `insertSchemaBasedXML()` method would look as follows:

```
catch (Exception $e) {
```

```
        print 'Could not insert a row into the ' . $table . ' table: ' . $e-  
>getMessage();  
    }
```

Taking Advantage of Oracle XML DB Functionality

The previous section demonstrated how you can move XML data in an Oracle database with PHP by means of SQL. In fact, Oracle XML DB provides you with different strategies to move XML content in and out of the Oracle database.

For example, as an alternative to SQL you could use industry-standard Internet protocols such as FTP, HTTP, or WebDAV, to insert XML content into the database. In this case, you simply put the files that contain the schema based XML content into a folder in Oracle XML DB repository. Oracle then implicitly inserts the appropriate rows into the table specified as the default table in the XML schema. Turning back to the `MyDomDocument` class, you might want to add a new `insertXMLByFTP()` public method to that class:

```
public function insertXMLByFTP($tagName, $table, $user, $pswd, $ftp_dir,
$host, $port=2100, $timeout=30)
{
    try {
        $items = parent::getElementsByTagName($tagName);
        $con_id = ftp_connect($host, $port, $timeout);
        $login = ftp_login($con_id, $user, $pswd);
        ftp_chdir($con_id, $ftp_dir);
        for ($i = 0; $i < $items->length; $i++) {
            $nodeXML = parent::saveXML($items->item($i)) . "\n";
            $dest_file = $tagName . $i . '.xml';
            $temp = tmpfile();
            fwrite($temp, $nodeXML);
            fseek($temp, 0);
            if(!ftp_fput($con_id, $dest_file, $temp, FTP_ASCII))
                throw new Exception("Cannot put $dest_file");
            fclose($temp);
        }
        catch (Exception $e) {
            throw $e;
        }
        print "Data have been submitted";
    }
}
```

To see the `insertXMLByFTP()` method in action, you might use the following script. Before you run the script shown below, make sure to delete the rows inserted into the `employee` table in the previous example in order to avoid integrity constraint violations.

```
<?php
//File: domXMLFTP_db.php
require_once 'MyDomDocument.php';
require_once 'dbConn5e.php';
require_once 'ScottCred.php';
try {
    $dom = new MyDomDocument();
```

```

    $root = $dom->addElement($dom, 'EMPLOYEES');
    $db = new dbConn5e($user, $pswd, $conn);
    $sql="SELECT EMPNO, ENAME, JOB FROM EMP";
    $db->query($sql);
    while ($row = $db->fetch()) {
        $empl = $dom->addElement($root, 'EMPLOYEE');
        $empl->setAttribute('xmlns:xsi', 'http://www.w3.org/2001/XMLSchema-
instance');
        $empl->setAttribute('xsi:noNamespaceSchemaLocation', 'http://localhost:
8080/public/emp.xsd');
        $emplno = $dom->addElement($empl, 'EMPNO', oci_result($row,'EMPNO'));
        $ename = $dom->addElement($empl, 'ENAME', oci_result($row,'ENAME'));
        $title = $dom->addElement($empl, 'TITLE', oci_result($row,'JOB'));
    }
    $dom->insertXMLByFTP('EMPLOYEE', 'employee', 'usr', 'pswd', '/public',
'localhost');
}

catch(Exception $e) {

    print $e->getMessage();
}
?>

```

The `domXMLFTP_db.php` script will create the appropriate schema-based `employee` XML document for each row stored in the `emp` table under `SCOTT/TIGER` demonstration schema and then put that XML document as a file into the `/public` directory of Oracle XML DB repository. Oracle in turn will automatically insert the appropriate row into the `employee` table of `XMLType` in the `USR` database schema. To be sure it has done so, you can issue the following query after you have run the above script:

```

SELECT COUNT(*) FROM employee;

COUNT(*)
-----
        14

```

Note that despite the fact that you have the same results regardless of the way you choose to insert XML content into an `XMLType` table, there are a few differences in the way XML data is moved in the database. For instance, using the SQL-based approach allows you to explicitly control transactions from your PHP code—the thing that cannot be accomplished by means of FTP or another Internet protocol mentioned earlier.

However, you should remember that all the operations you perform against the database by using the `oci_execute()` function are not transactional by default. This is because the default execution mode is `OCI_COMMIT_ON_SUCCESS`. To change the default behavior, you must specify the `OCI_DEFAULT` execution mode as the second parameter in the `oci_execute()` call. This will allow you to explicitly control the transaction from your PHP script. Turning back to the

dbConn5e class, discussed earlier in this article, you might want to edit its query() public method as follows:

```
public function query($sql, $mode = OCI_COMMIT_ON_SUCCESS)
{
    if(!$this->query = oci_parse($this->conn, $sql)) {
        $err = oci_error($this->conn);
        throw new Exception('Failed to execute SQL query: ' .
        $err['message']);
    }
    else if(!oci_execute($this->query, $mode)) {
        $err = oci_error($this->query);
        throw new Exception('Failed to execute SQL query: ' .
        $err['message']);
    }
    return true;
}
```

Next, add the following public methods to the dbConn5e class:

```
public function commit()
{
    if(!oci_commit($this->conn)) {
        return false;
    }
    return true;
}
public function rollback()
{
    if(!oci_rollback($this->conn)) {
        return false;
    }
    return true;
}
```

With the above dbConn5e's methods in place, you might want to add a new insertSchemaBasedXML_trans() public method to the MyDomDocument class discussed earlier in this article:

```
public function insertSchemaBasedXML_trans($tagName, $db, $table, $schema)
{
    try {
        $items = parent::getElementsByTagName($tagName);
        for ($i = 0; $i < $items->length; $i++) {
            $nodeXML = parent::saveXML($items->item($i)) . "\n";
            $db->query("INSERT INTO " . $table . " VALUES( XMLType('" . $nodeXML .
            "') .createSchemaBasedXML('" . "$schema" . "'))", OCI_DEFAULT);
        }
        if(!$db->commit())
            print "Failed to commit the transaction";
    }
}
```

```

catch (Exception $e) {
    if($db->rollback())

        print "Failed to rollback the transaction";

    throw $e;
}
print "Data have been submitted";
}

```

Now you might want to replace the `insertSchemaBasedXML()` method with `insertSchemaBasedXML_trans()` in the `domXML_db.php` script discussed earlier in the article in order to guarantee that all the generated XML content will be inserted into the `employee` table. If it fails to insert at least one row then the whole transaction will be rolled back.

Another important feature of Oracle XML DB is its access control list (ACL) based security mechanism that is designed to protect resources stored in Oracle XML DB. In the `domXMLFTP_db.php` script discussed earlier in this section, you store the `employee` XML documents in the `/public` repository folder. However, in a real situation, using the `/public` folder is not always a good idea because any database user has the `READ` and `WRITE` privileges on that folder by default. Thus, you might want to create a new protected folder to store your data. To do this, you could perform the following steps.

First, you create a new ACL that grants all privileges to the database user whose account you are going to use to connect to the database. For instance, in the following example you grant all privileges to the `USR` database user. You can run the following PL/SQL code from the `SQL*Plus` being connected as a user with the `XDBADMIN` or `DBA` role:

```

DECLARE
    rslt BOOLEAN;
BEGIN
    rslt := DBMS_XDB.createResource('/sys/acls/all_usr_acl.xml',
    '<acl description="usr_acl"
    xmlns="http://xmlns.oracle.com/xdb/acl.xsd"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.oracle.com/xdb/acl.xsd
    http://xmlns.oracle.com/xdb/acl.xsd">
    <ace>
        <principal>USR</principal>

        <grant>true</grant>

        <privilege>
            <all/>
        </privilege>
    </ace>
    </acl>');
    COMMIT;
END;

```

```

/
PL/SQL procedure successfully completed.
The following PL/SQL code will create a new folder in XML repository and protect it by the
all_usr_acl ACL created above:
DECLARE
    rslt BOOLEAN;
BEGIN
    rslt:=DBMS_XDB.createFolder('/home/usr');
    DBMS_XDB.setAcl('/home/usr', '/sys/acls/all_usr_acl.xml');
    COMMIT;
END;
/
PL/SQL procedure successfully completed.

```

Note that only the USR database user and the users granted the DBA role will have all privileges on the /home/usr folder. For example, you could change now the directory specified in the insertXMLByFTP() call in the domXMLFTP_db.php script as follows:

```

    $dom->insertXMLByFTP('EMPLOYEE', 'employee', 'usr', 'pswd', '/home/usr',
    'localhost');

```

This will work, since you still connect to the database as USR.

Conclusion

As you no doubt have realized, taking advantage of object orientation in PHP lets you write scalable, maintainable, and reusable code. Once a class has been written and debugged, you can then reuse it in a number of different ways. This allows you to reuse well-designed pieces of object-oriented code over and over, reducing or eliminating redundant code in your applications. Another important point to recognize is that PHP is evolving, becoming object-oriented in nature. With a wide range of predefined classes—that you can reuse or extend as needed—available in PHP 5, you can design even complex applications with a minimum of effort.

Yuli Vasiliev (jvyul@yahoo.com) is a software developer, freelance author, and consultant who focuses mainly on Oracle Objects and Oracle XML Technology.

References

[PHP 5, Oracle, and the Future](#)

[XML in PHP 5 – What’s New?](#)

[Zend Engine II - PHP's OO Evolution](#)

[Changes in PHP 5/Zend Engine II](#)

[Exceptional Code](#)

The Hitchhiker's Guide to PHP

Open Source Developer Center

PHP Manual