

## Session 8 – April 11, 2005

- fundamentals of database design
- referential integrity
- JOINing multiple tables
- MyIsam vs InnoDB (transactional vs non-transactional)
- mysql's OO interface
- prepare/execute
- using PEAR
- PEAR::DB
- PHP 5 Exception mechanism
- sample application: PHP University
- assignment

### Lightning Intro to Database Design

Imagine you were setting up a web application for a hypothetical PHP University. You would have students, courses, and instructors.

Each of those things is an *entity* and they have *attributes*. A student has attributes such as firstname and lastname. An instructor also has firstname, lastname, perhaps a bio for marketing hype purposes. A course – we're going to call them workshops, one-day affairs so we can avoid the additional complexity of dates and times of class meetings – has a name and a description. It also has an instructor, and a date and time when it takes place.

In your data model, the entities become tables and the attributes become columns of those tables.

When you model the relationships between the entities you need to consider which side of the relationship is "one" and which side is "many." An instructor can have more than one workshop. (A workshop could also have more than one instructor if you do team-teaching, but for simplicity's sake let's pretend we don't.) A student can take more than one workshop.

The relationship between instructor and workshop in our model is one-to-many, or 1-M. It would be a poor idea to have columns like "instructor\_lastname" in our workshop table, because lastname is properly an attribute of an instructor, not a workshop. If an instructor's name were misspelled or otherwise had to be changed, it would be awkward to manage if the name were scattered all over the database. Indeed, one of the principles of database *normalization* is that one fact should only exist in one place at one time. If there's an instructor whose last name is Foo, that fact belongs in the instructor table.

The instructor entity itself is an attribute of the workshop; and the last name of the instructor, an attribute of the instructor.

So our workshop table might look like this:

```
mysql> describe workshop;
```

Field	Type	Null	Key	Default	Extra
id	smallint(5) unsigned		PRI	NULL	auto_increment
name	varchar(60)		MUL		
description	text				
instructor_id	smallint(6)			0	
date_time	datetime			0000-00-00 00:00:00	
lastmod	timestamp(14)	YES		NULL	

6 rows in set (0.04 sec)

And the instructor table will look something like this:

```
mysql> describe instructor;
```

Field	Type	Null	Key	Default	Extra
id	smallint(5) unsigned		PRI	NULL	auto_increment
lastname	varchar(30)		MUL		
firstname	varchar(30)				
middlename	varchar(30)				
email	varchar(40)				
bio	text				

6 rows in set (0.00 sec)

The instructor can have many workshops; the workshops will have only one instructor identified by the instructor\_id, which points to an instructor record having that id. Now, if you should happen to need to modify the instructor's data, you do it once and in just one place.

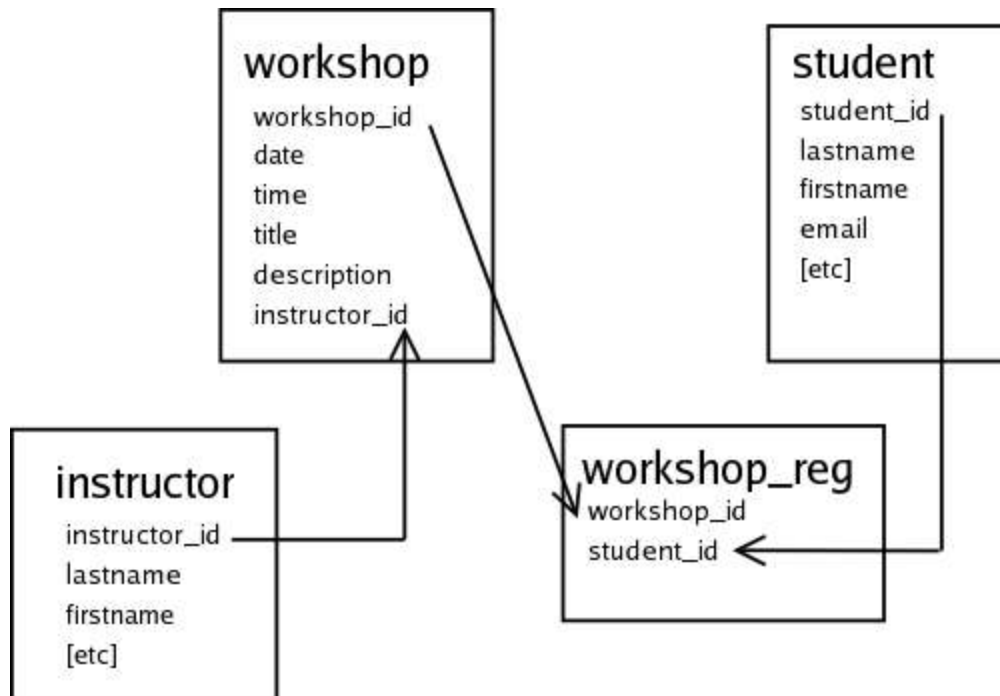
What about the students and the workshops? A student can take many workshops, and a workshop certainly can have many students in it. How do you model that many-to-many relationship and keep it from getting messy? As it happens, the workshop registration itself can be thought of as an entity. There is a 1-M relationship between student and registration, and a 1-M relationship between workshop and registration. Solution: create a workshop\_registration table. Our table ends up looking rather abstract, but also quite simple:

```
mysql> describe workshop_registration
```

Field	Type	Null	Key	Default	Extra
student_id	smallint(5) unsigned		PRI	0	
workshop_id	smallint(5) unsigned		PRI	0	

2 rows in set (0.00 sec)

This type of table, which resolves the M-M relationship quite nicely, is sometimes called a *junction table*. Our database schema can be represented graphically as shown below. The arrowed ends of the lines are the “many” side of the one-to-many (1-M) relationship.



## Joining multiple tables with JOIN

One consequence of having an integer representing the instructor in the workshop table is that you have to do a little more work to SELECT data that includes, for example, the workshop names and dates as well as a human-friendly representation of the instructor, as opposed to a number. For that we have JOIN, of which an example follows on the next page. In a typical JOIN query, you essentially create a new in-memory table consisting of columns from different tables based on the equality of two columns in different tables. In a well-designed database you will probably find yourself JOINing tables often. See <http://dev.mysql.com/doc/mysql/en/JOIN.html>

## Referential Integrity

Having an elegant data model is of little value if you have orphaned records all over the place. Imagine, for example, that you delete a student record but leave behind the related student workshop registration rows: the `student_id` becomes mysterious, having no corresponding parent record in the student table. So your application needs to enforce *referential integrity* by deleting all the student's related records in other tables as well as the student record itself in the student table.

Note that MySQL's InnoDB table type supports transactions as well as *foreign keys*. You can stipulate at table creation time (or later by means of ALTER TABLE) that related rows should be deleted from child tables when a parent record is delete (with an "ON DELETE CASCADE" clause).

You can also set up the key relationship so that it is an error to try to insert an orphan into a child table, e.g., MySQL won't let you insert a value for `instructor_id` in the workshop table if there's

no instructor record in the parent instructor table that has that id. You would do this is by specifying the index in the child table – e.g., instructor\_id – as NOT NULL. Please see [http://dev.mysql.com/doc/mysql/en/ANSI\\_diff\\_Foreign\\_Keys.html](http://dev.mysql.com/doc/mysql/en/ANSI_diff_Foreign_Keys.html) for details.

You can also enforce referential integrity at the application rather than at the database level, and that's the way our sample application "PHP University" has elected to do it. It uses MyIsam tables (the default), which do not support either transactions or foreign keys, as opposed to InnoDB tables, which do, but which are not as fast as MyIsam.

## Data Validation and Business Rules

Data validation can get a little more complicated when you're dealing with an underlying database, as opposed to performing a more simple-minded task like emailing form input to somebody. What fields are required? What kinds of values are permitted where? One gotcha about MySQL is that in certain circumstances, if the input you are supplying in an UPDATE or INSERT statement is not valid for the column type, MySQL will not complain or throw any sort of error, but instead will silently change the value to some default. For example, a malformed date/time input value gets turned into "0000-00-00 00:00:00."

In the case of the interface we've built for the workshop table, we assume a business rule that says an administrative user *can* insert a workshop without specifying the instructor. In such cases, the instructor id is simply 0. So we have to use a LEFT JOIN to read the data for all the workshops, whether there's an instructor appointed or not.

```
SELECT
    date_format( workshop.date_time, '%a %M %e, %Y' ) 'date',
    lower( date_format( workshop.date_time, '%l:%i %p' ) )
'time',
    workshop.name,
    workshop.id,
    IF (
        instructor.lastname IS NOT NULL ,instructor.lastname,'TBA'
    ) 'instructor'
FROM workshop
LEFT JOIN instructor ON workshop.instructor_id = instructor.id
ORDER BY workshop.date_time")
```

date	time	name	id	instructor
Tue December 7, 2004	4:00 pm	Secure Web Applications	2	TBA
Sat January 15, 2005	2:00 pm	Web Services	1	Sklar
Tue February 8, 2005	2:00 pm	Tuning High End Mysql Apps	3	Zawodny
Thu March 24, 2005	10:00 am	Grokking PHP 5	6	Trachtenberg

The resulting data set will contain either “TBA” for those workshops records that have no instructor, or else a lastname for those that do. Working out the SQL is sometime the trickiest part – it's a good idea to test your query at the command line before doing it through PHP. Once you get them, looping through the results is cake.

Incidentally, if ever you do need to purge orphans from a table, this same LEFT JOIN technique can be used. Suppose someone, somehow (despite your careful coding), deletes a workshop but leaves the related workshop registration records intact, orphaned. You could find them with

```
mysql> SELECT workshop_registration.workshop_id FROM
workshop_registration LEFT JOIN workshop ON workshop.id =
workshop_registration.workshop_id WHERE workshop.id IS NULL;
+-----+
| workshop_id |
+-----+
|          60 |
+-----+
1 row in set (0.00 sec)
```

You could also locate and delete orphans in one query, because MySQL supports multi-table delete statements:

```
mysql> DELETE workshop_registration FROM workshop_registration LEFT JOIN
workshop on workshop_registration.workshop_id = workshop.id WHERE
workshop.id IS NULL;
Query OK, 1 row affected (0.00 sec)
```

Another issue to consider is under what circumstances certain records should be deleted at all. For example, if a user wants to delete an instructor whose related workshop records are all in the future, that's one thing. But suppose the instructor has related workshop records that took place in the past. Do your users really want to erase history? It might be better to have a column in the instructor table that serves as a flag, perhaps called something like 'inactive', and turn it on in those instructor records you might otherwise be tempted to delete outright. Or you could move them to some different table for instructors who have quit, retired, or been fired (-:

## Side note: Transactions

InnoDB is a table type that supports a database feature known as transactions. A transaction is a sequence of database queries that works as one atomic unit. Either the whole thing succeeds, or the whole thing fails. If something fails, you can roll back the database to the state it was in before the transaction began. A classic example is a financial application in which the customer transfers money from one account to another. Suppose the first query decrements the source account by  $n$  dollars and second increments the destination account by  $n$  dollars. If the first succeeds but the second does not, you have an unhappy customer.

To use transactions in MySQL, you first specify the table type as InnoDB at creation time.

To wrap a set of queries in a transaction, first set autocommit to false.

Do your transactions, then commit when all is well. Otherwise, rollback.

```
mysql> set autocommit = false;
Query OK, 0 rows affected (0.00 sec)
mysql> select lastname from transactional;
```

```

+-----+
| lastname |
+-----+
| Smith    |
| Gacker   |
| Bludgeon |
+-----+
3 rows in set (0.00 sec)

mysql> update transactional set lastname = "D'Oh!!!!!" ;
Query OK, 3 rows affected (0.00 sec)
Rows matched: 3  Changed: 3  Warnings: 0

mysql> select lastname from transactional;
+-----+
| lastname |
+-----+
| D'Oh!!!! |
| D'Oh!!!! |
| D'Oh!!!! |
+-----+
3 rows in set (0.00 sec)

mysql> rollback;
Query OK, 0 rows affected (0.00 sec)

mysql> select lastname from transactional;
+-----+
| lastname |
+-----+
| Smith    |
| Gacker   |
| Bludgeon |
+-----+
3 rows in set (0.00 sec)

```

Doing transactions programmatically (as opposed to via the command line client) is straightforward. The `mysqli` API provides `autocommit(boolean mode)`, `commit()` and `rollback()` methods to call on your database connection object, and these have procedural counterparts as well. See the docs for details.

## Concurrency

Any RDBMS worth the name is very good about preventing data corruption resulting from different clients writing simultaneously. But the application developer who isn't careful can still be susceptible to what is known as the *lost update* problem. Suppose I open John Customer's contact data record via a web form and change his home phone number but I don't touch his office number. Meanwhile, someone else opens John Customer's record and changes the office but not the home number. Unless the application takes measures to prevent it, whichever user saves last will clobber the other's update.

A good technique is to have a timestamp column in the table --in our student table we call it `lastmod` -- and retrieve it when you `SELECT` -- you can put it in an HTML form as a hidden field. Remember that a MySQL timestamp column type automatically saves the date/time of the last modification to the record. Upon form submission, use `LOCK TABLES` to keep another client

from sneaking in an UPDATE behind your back; SELECT the timestamp again and compare it to the submitted timestamp value; abort the operation if the two are not equal; otherwise, do the UPDATE; then unlock the tables (unconditionally) when you're done. (See [http://dev.mysql.com/doc/en/LOCK\\_TABLES.html](http://dev.mysql.com/doc/en/LOCK_TABLES.html). Here is pseudo-code; we will see a more concrete example later in the discussion.

```
function update() {
    lock table;
    SELECT lastmod;
    compare submitted lastmod to lastmod value just fetched;
    if ( their_lastmod != database_lastmod) {
        unlock tables;
        abort;
    } else {
        continue with UPDATE;
        unlock tables;
    }
}
```

## Other Error Handling

Besides checking for concurrent access problems, you also check the return value of any `mysql_query` that does an UPDATE or INSERT to make sure the query succeeded, because it can fail even if the query is syntactically valid. For example, if `student.email` is defined as a unique index, then by definition each email must be unique in that table. If  `david.mintz@phpuniversity.edu` is already in David Mintz' email field, you can't change somebody else's email to  `david.mintz@phpuniversity.edu`; that would violate a unique index constraint and MySQL will give you a *duplicate entry* error if you try.

Your application should handle this type of error gracefully, as a sort of second level of input validation, rather than treating it as a serious database error and quitting.

## The OO interface to mysqli

In Session 6 we dealt with the `mysqli` extension for accessing MySQL 4.1+ databases in PHP 5; we introduced PHP classes and objects in Session 7. If you looked at the `mysqli` documentation at `php.net`, you noticed `mysqli` has two interfaces: one procedural and one object oriented. We are now in a position to appreciate the latter.

Step one is the same as with the procedural interface: get a connection object.

```
$db = mysqli_connect('localhost', 'php', 'phpis2kool', 'php_university');
```

Again, `mysqli_connect()` returns a database connection object if it succeeds; on failure, it returns false, and PHP will throw an error of level `E_WARNING` with an error message such as

**Warning: mysqli\_connect(): Unknown MySQL server host 'local host' (1) in /usr/local/apache/htdocs/php/spring\_2005/examples/8/mysqli\_connect.php on line 3**

With a connection in hand, as before, you do your database queries through this object; but with the OO interface you call methods on \$conn instead of calling functions called mysqli\_xxxx() and passing the connection object as the first argument. You may well find this style less verbose and more natural than the procedural interface:

```
$result = $db->query("select lastname, firstname from instructor order by
lastname,firstname");
while ($instructor = $result->fetch_assoc()) {
    echo "$instructor[lastname], $instructor[firstname]<br />\n" ;
}
$result->close();
$db->close();
```

The query() method return as mysqli\_result object; and on this \$result object, in turn, you call fetch\_xxx() methods to get the data. All the procedural mysqli functions have object oriented counterparts. Enjoy. <http://php.net/manual/en/ref.mysqli.php>

## Prepare/Execute with OO mysqli

In the Session 6 examples we used mysqli\_escape\_string() to make sure that external variables used in our queries would not cause a SQL error, or worse. Another technique involves a *prepared statement* with placeholders for the variable data which you then *execute* against the database.

You send a sort of query template to the server first, with question marks representing the paramters; then you *bind* the input parameters to your PHP variables; then execute the query. As you might expect, you use the prepare(), bind\_param() and execute() methods to do this:

```
$lastname = "O'Reilly" ; $firstname = 'Tim' ;
$stmt = $db->prepare("INSERT INTO person (lastname,firstname) VALUES (?,?)")
    or die($db->error());
$stmt->bind_param('ss', $lastname, $firstname);
$stmt->execute()
    or die($db->error());
$stmt->close();
```

Note the first argument to bind\_param(): this is a *datatype specification* string consisting of one character for each of the respective query parameters. The types are i, d, s and b for integer, double, string and blob (big data type), respectively. In the above example we have two strings parameters -- \$lastname and \$firstname -- so the first argument to bind\_param() is 'ss'.

The advantages of getting in the habit of using prepare/execute are (1) you do not have to call any escape\_string method on your input; the database driver does this automatically for you; (2) if you are running the same query multiple times with different input data, you gain considerable efficiency – prepare once, execute repeatedly; (3) some other database APIs in other programming languages require you to prepare/execute in most if not all cases, so if you plan ever to use something like Java JDBC or Perl DBI, it's not a bad idea to get used to prepared statements now.

You can also bind PHP variables to output results and gain an increase in speed, particularly for large datasets, by using the bind\_result() method.

```

$db = mysqli_connect('localhost','php','phpis2kool','php_university')
    or die('ack!');

$search_term = "M%";

// prepare statement
$stmt = $db->prepare(
    "SELECT id, lastname, firstname
    FROM student WHERE lastname like ?
    ORDER BY lastname, firstname");

// bind input parameter
$stmt->bind_param('s' ,$search_term);

// execute statement
$stmt->execute() or die($stmt->error);

// buffer result set in statement handle so we can get count
// before we iterate through it
$stmt->store_result();
echo "<p>Number of rows found: $stmt->num_rows</p>";

// bind output (result) to variables
$stmt->bind_result($id,$lastname,$firstname);

// iterate through result. Note how $id, $lastname, $firstname
// automatically become the row values
while($stmt->fetch()) {
    echo "$id: $lastname, $firstname<br />" ;
}

// clean up
$stmt->close();
$db->close();

```

## Introducing PEAR

The PHP Application and Extension Repository includes dozens of (free) packages for a range of web programming tasks, including authentication, caching, benchmarking, filesystem access, http/web automation, date and time manipulation, image manipulation, logging, configuration file management, email, HTML form creation and validation, XML, and more. The basic PEAR library now comes bundled with PHP, and the default `include_path` in `php.ini` usually includes the PEAR installation directory. In other words, your development environment is probably ready now for you to use PEAR *packages* (or *modules*). See <http://pear.php.net/> to see what is available.

The PEAR package manager that makes tasks like upgrading, installing, and removing packages ridiculously easy, and this too is probably already installed on your system. On \*nix systems there is a shell script called `pear`; on Windows, a batch file called `pear.bat`. `pear[.bat]` provides a command line interface to package management. If you are on a shared host where you do not have sufficient privileges to install things in the central system library directories, you can configure `pear` to use your own personal directory. Run `pear` with no arguments to get a help screen.

If you find that there's no PEAR installation at all on your system, go to <http://pear.php.net/manual/en/installation.getting.php> for help.

Once you're set up, you can install a pear package simply by typing

```
pear install Package_Name
```

and then sit back and watch it go. If there are any failed dependencies, the installer will let you know so you can resolve them. In fact you can ask the installer to download and install any dependencies automatically by giving the `-a` or `--alldeps` option to the `install` command.

## PEAR DB

Perhaps the most popular PEAR package is DB, which provides unified API for accessing SQL databases and myriad convenience methods to make your database access easier. If you think you might eventually need to port your project to another database backend, using PEAR/DB from day one should make it a lot easier than it would be if you are using a database-specific native PHP API. If your SQL itself is tailored your particular db's idiosyncrasies, then some code will have to be re-written. But if the SQL is sufficiently generic, an application's database can be changed by changing just one connection string.

DB plays nicely with certain other PEAR packages -- for example, the HTML\_QuickForm package includes a class that can use DB::Result objects to populate html SELECT menus automatically.

If you think the preceding mysqli code example was verbose, you will probably be grateful for the shortcuts and convenience that PEAR DB provides – numerous wrapper methods that automate much of the parameter binding, result-freeing, and so forth.

### Connecting

Step One is load DB.php (e.g., with a `require` or `include`). It should be in the top level of your PEAR installation directory, which should be in your include path.

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

Step Two is get a connection by calling DB's static `connect()` method. The required argument is a connection string, formally known as a *Data Source Name* or DSN. `connect()` supports a fairly broad range of syntactical variations -- please see <http://pear.php.net/manual/en/package.database.db.intro-dsn.php> for complete documentation -- but a typical pattern is

```
database_program://user:password@host/database_name
```

In our PHP University application, we could connect by saying

```
$db = DB::connect('mysqli://php:phpis2kool@localhost/php_university');
```

This method will return one of two things: a database connection object on success, or a `DB_Error` object on failure. `DB_Error` contains information about what went wrong; you should check your return value before proceeding.

```
<?php
require_once 'DB.php';
$dsn = 'mysql://php:phpis2kool@localhost/php_university';

$db =& DB::connect($dsn);
if (DB::isError($db)) {
    die($db->getMessage());
}
?>
```

There are additional options that you can pass to `DB::connect()`. See <http://pear.php.net/manual/en/package.database.db.intro-connect.php> for information.

## PEAR (and DB) Error Handling

PEAR provides flexible error handling that makes it easy to adjust your error handling for development and debugging versus production environments and modify your error handling at runtime. The following discussion applies to the DB package as well as most other PEAR packages. See <http://pear.php.net/manual/en/core.pear.pear.intro.php>. You can call the `setErrorHandling()` method statically on the PEAR base class itself to configure your global PEAR error handling, or call it on a subclass of PEAR, such as the database connection object you will get from [DB::connect\(\)](#), to set the error handling for that object. You pass it one of these pre-defined integer constants, possibly followed by `$options`:

- `PEAR_ERROR_RETURN` If an error occurs, a `PEAR_Error` is returned from the error-generation method (normally `raiseError()`.) This is the default.
- `PEAR_ERROR_PRINT` Like `PEAR_ERROR_RETURN`, but an error message will be printed additionally.
- `PEAR_ERROR_TRIGGER` Like `PEAR_ERROR_RETURN`, but the PHP builtin-function `trigger_error()` will be called in `PEAR_Error`'s constructor with the error message.
- `PEAR_ERROR_DIE` The script will terminate and an error message will be printed on instantiation of a `PEAR_Error`.
- `PEAR_ERROR_CALLBACK` If a error occurs, the callback passed to `$options` is called.
- `PEAR_ERROR_EXCEPTION` If Zend Engine 2 is present, then an exception will be thrown using the `PEAR_Error` object.

`PEAR_ERROR_CALLBACK` is roughly analogous to PHP's native `set_error_handler`. If you want to set a callback, you pass `setErrorHandling()` a second `$options` argument which is either (1) the name of the callback function you want to get called in the event of an error, or (2) an array whose first element is the name of a class and whose second element is the class method you want to be invoked in the event of an error.

You can temporarily change your error handling for a specific block of code -- the most convenient way is with `pushErrorHandling(PEAR_ERROR_XXX[, $options])`, which takes the same arguments as `setErrorHandling`. When you want to restore the status quo ante you need

only call `popErrorHandler()`. This relieves you of the burden of having to know what the previous state of error handling was.

## DB continued

Step Three is to execute some queries against your database, now that you have a connection object to do it with.

```
$result = $db->query(
    "SELECT lastname,firstname, id FROM student WHERE lastname like 'M%');
```

For SELECT queries, the `query()` method returns either a `DB_Result` object on success, or guess what, a `DB_Error` object on failure. (Remember that failure means a database error; an empty result set is not an error.)

If the query is a non-SELECT (e.g., update, insert, delete), then it will return either `DB_OK` (an integer constant) or a `DB_Error`.

If you are doing a SELECT query, you can loop through your result set using `while`.

```
while ($data = $result->fetchRow()) {
    // display $data
}
```

Is `$data` a numerically indexed array, a string-indexed array, or something else? The answer depends on the current *fetch mode* which you can set it via `DB`'s `setFetchMode` (`DB_FETCHMODE_XXX`) method, to which you pass one of `DB_FETCHMODE_ORDERED`, `DB_FETCHMODE_ASSOC` or `DB_FETCHMODE_OBJECT`.

If you use `DB_FETCHMODE_ORDERED` (the default) then you have to refer to the columns by their numeric index. I generally prefer `DB_FETCHMODE_ASSOC`, and set it accordingly in the same block of code that sets up the database connection.

You can also change the fetch mode on a per-query basis simply by passing the desired fetch mode constant in an argument to `fetchRow()`:

```
while ($data = $result->fetchRow(DB_FETCHMODE_ASSOC) {
    echo "id $student[id]: $student[lastname], $student[firstname]<br />";
}
```

## Prepare/Execute and Placeholders with PEAR/DB

PEAR DB has excellent support for prepared statements and placeholders.

```
$sth = $db->prepare('UPDATE student SET email = ? where id = ?');
$db->execute($sth, array('dmintz@davidmintz.org',226));
```

If you are doing multiple updates, inserts or deletes that are otherwise the same but differ only in the input parameters, it is more efficient to do one `prepare()` followed by multiple `execute()` statements. Even if you aren't repeating the same SQL statement, `prepare/execute` absolves you

of the responsibility of quoting your string values; the database driver takes care of it automatically. (The same is true of the mysqli interface itself. Since we specified mysqli in our DSN, PEAR/DB is actually using native PHP/mysqli under the hood.)

It gets better. PEAR/DB also provides autoPrepare() and autoExecute() methods to take a lot of the tedium out of writing SQL queries. With autoExecute(), you need only have in hand a table name and an array whose keys correspond to table columns, and whose values are the input data. Then you can do an insert without writing any SQL at all (because DB writes it for you):

```
$table_name = 'student';
// suppose $fields_values is validated $_POST data
$fields_values = array(
    'lastname' => 'Somebody',
    'firstname' => 'John',
    'email' => 'john.somebody@example.org'
);
$result = $db->autoExecute($table_name, $fields_values,
    DB_AUTOQUERY_INSERT);
```

As you might surmise, the last argument tells the driver that this is an insert. To do an update, you would do something like this:

```
$fields_values = array(
    'email' => 'john.somebody@example.org'
);
$result = $db->autoExecute($table_name, $fields_values,
    DB_AUTOQUERY_UPDATE, 'id = 123');
```

Note in particular the string 'id = 123'. This optional argument, if provided, will be used to create a WHERE clause. If you leave it out, you will change every email in the table to john.somebody@example.org.

See <http://pear.php.net/manual/en/package.database.db.intro-auto.php>

The query() method also supports the placeholder syntax. The first argument is a string of SQL with placeholders; the second is the input parameter(s), and can be either a scalar or an array:

```
$result =& $db->query('SELECT * FROM student WHERE lastname LIKE ?', 'M%');
$result =& $db->query('SELECT * FROM student WHERE lastname LIKE ? or EMAIL
like ?', array('M%', '%somewhere.com' ));
```

This technique usually is more convenient than manually calling mysql\_escape\_string() or otherwise doing your own escaping. But if you do need to do it yourself and wish to maintain consistency and database-neutrality, you can use DB's quoteSmart() or escapeSimple() methods. Both return a string escaped for safe use in your DBMS. The former wraps the string in quotes while the latter does not.

Incidentally, support for prepared statements was just introduced in MySQL as of version 4.1 -- which was certified as production ready in October 2004 -- but the PEAR/DB mysql driver has emulated prepared statements since well before then, for compatibility reasons. The standard

native PHP MySQL API does not support prepared statements; the newer mysqli extension for MySQL 4.1.3+ does. <http://www.zend.com/php5/articles/php5-mysqli.php>

## Freeing Results and Disconnecting

Again, when you are through with a result set, it is good practice to free the memory it's using by calling `$result->free()`. When you are finished with a database connection, it is good practice to disconnect by calling `$db->disconnect()`.

Both of these things are accomplished for you automatically by PHP when the script exits, but it is nonetheless good hygiene to clean up after yourself, and it will help performance on a high traffic site.

## More PEAR/DB Convenience Functions

With each of the following methods, you do not have to get a `DB_Result` object by calling `query` on a `DB` object, and then get at the data through the result object, then free the result. Those steps are wrapped up in each of these `DB` methods. Please see <http://pear.php.net/manual/en/package.database.db.php> for full documentation.

```
array &getCol (string $query [, mixed $col = 0 [, mixed $params = array()]])
```

Runs the query provided and puts (only) the first column of data into an array then frees the result set. Nice, for example, if you want table metadata but all you want to know are the column names.

```
$columns = $db->getCol("SHOW COLUMNS FROM $table");
```

would put just the names of the columns (without the rest of the stuff about column types) in an array `$columns`.

```
array &getRow (string $query [, array $params = array() [, integer $fetchmode = DB_FETCHMODE_DEFAULT]])
```

Runs the query provided and puts just the first row of data into an array, then frees the result set.

```
mixed &getOne (string $query [, mixed $params = array()]])
```

Runs the query provided and returns the data from the first column of the first row, then frees the result set.

```
array &getAll (string $query [, array $params = array() [, integer $fetchmode = DB_FETCHMODE_DEFAULT]])
```

Runs a query and returns all the data as an array. For complete documentation see <http://pear.php.net/manual/en/package.database.db.db-common.getall.php>

```
array &getAssoc (string $query [, boolean $force_array = FALSE [, mixed $params = array() [, integer $fetchmode = DB_FETCHMODE_DEFAULT [, boolean $group = FALSE]]]])
```

Runs the query provided and puts the entire result set into an associative array then frees the result set. For complete documentation see <http://pear.php.net/manual/en/package.database.db.db-common.getassoc.php>

## Pros and Cons of PEAR (and DB)

Some PHP gurus knock PEAR on the grounds that it is slow and bloated. However, many other reputable PHP developers use PEAR and use it gladly.

Another criticism of PEAR is that the documentation is uneven. Some packages have excellent docs, and some have none other than the auto-generated API documentation (which in some cases may be plenty, but sometimes is not). You may occasionally find yourself googling for tutorials and articles, or consulting a book like David Sklar's outstanding *Essential PHP Tools: Modules, Extensions, and Accelerators*, or examining the source code itself. Incidentally, the online Pear/DB documentation is first-rate:

<http://pear.php.net/manual/en/package.database.db.php> Lots of PEAR packages also come with code examples that are illuminating.

If you are coding a MySQL web application, the decision between mysqli and PEAR/DB depends on your priorities. If portability and ease/speed of development are at a premium, you want PEAR. If performance is your top priority, you want mysqli.

## The PHP 5 Exception Mechanism

We covered PHP error handling functions in Session 4 and we talked briefly about PEAR error handling mechanisms in this session. PHP 5 provides yet another feature known as Exceptions, similar to the Exception mechanism found in other languages.

An Exception is a PHP class that encapsulates error information. The three languages constructs involved are try, catch and throw. If you are in a function or method body and want to signal an error condition and abort further execution of your method, then you throw a new instance of Exception with the throw keyword.

```
throw new Exception("just because");
```

The Exception constructor accepts an optional string error message and optional integer error code. Here is a function that throws an Exception for no reason except demonstration:

```
function throwIt() {  
    $x = 4;  
    printf("%s(): set x to 4 at line %d<br />", __FUNCTION__, __LINE__);  
    $y = 5;  
    printf("%s(): set y to 5 at line %d<br />", __FUNCTION__, __LINE__);  
    throw new Exception("woo Hooo!");  
    $z = 6;  
    printf("%s(): you won't see z set to 6 at line %d<br />", __FUNCTION__, __LINE__);  
}
```

```
        return true;
    }
}
```

Of course this is a contrived example. In real life you throw an Exception when something exceptional happens that warrants changing the flow of execution – a file not found, file access denied, database unavailable, database record that you expect to find not found, etc.

The client code that invokes `throwIt()` should look something like this:

```
try {
    // calls to methods/functions that might throw an Exception
}
catch (Exception $e) {
    // exception handling
}
```

In the code that calls that function, you wrap all the statements that might trigger an Exception inside a try block, which must be followed by at least one catch block. If an exception results from one of the statements in the try, execution of that block stops and jumps down to the catch block, where your Exception-handling code can handle it as it sees fit – including throwing another Exception (or re-throwing the same one) on up the calling stack.

```
try {
    printf ("this is line %d<br />\n",__LINE__);
    printf ("this is line %d. Gonna try throwIt()<br />\n",__LINE__);
    throwIt();
    printf ("you won't see this at line %d<br />\n",__LINE__);
} catch (Exception $e) {
    echo '<pre>';
    print_r($e);
    echo '</pre>';
}
```

When you run this code, you see that in the `throwIt()` function, the code following the `throw` statement is never executed. The same is true of the statement following the call to `throwIt()` in the catch block.

The output of `print_r($e)` shows the data that Exception instance contains.

Exception Object

```
(
    [message:protected] => woo Hooo!
    [string:private] =>
    [code:protected] => 0
    [file:protected] => /
usr/local/apache/htdocs/php/spring_2005/examples/8/exception.php
    [line:protected] => 9
    [trace:private] => Array
        (
            [0] => Array
                (
                    [file] => /
usr/local/apache/htdocs/php/spring_2005/examples/8/exception.php
```

```

        [line] => 9
        [function] => throwIt
    )
    [1] => Array
    (
        [file] => /
usr/local/apache/htdocs/php/spring_2005/examples/8/exception.php
        [line] => 19
        [function] => throwIt
    )
)
)
)

```

Many of the code examples you have seen so far are replete with statements of the type `doSomeFunction()` or `die($errorMessage)`, or other error checking at almost every step. Putting all the code that might fail inside a method, throwing exceptions, and using `try/catch` constructs in the client code relieves client's burden of having to check every return value, and allows for cleaner, more readable code.

It is possible to write custom exceptions by extending the `Exception` class. If you have multiple catch blocks, the first catch that matches the `Exception` thrown will be the only one executed. Therefore multiple catch blocks should appear ordered from more specific to more general, as show below:

```

class SQLException extends Exception {
    function __construct($message,$code=0) {
        parent::__construct($message,$code);
    }
}

class SomeSpecificSQLException extends SQLException {
    function __construct($message,$code=0) {
        parent::__construct($message,$code);
    }
}

class MyDatabase {
    function doDatabaseStuff() {
        // pick a number between 1 and 3.
        // if 3, throw Exception; 2, SQLException; 1, SomeSpecificSQLException
        $num = rand(1,3);
        if ($num == 1) {
            throw new SomeSpecificSQLException(
                "this is the most specific, a subclass of SQLException"
            );
        } elseif ($num == 2) {
            throw new SQLException(
                "a SQL Exception, subclass of Exception");
        }
    }
}

```

```

        } else {
            throw new Exception("a general Exception");
        }
    }
}

$db = new MyDatabase() ;

try {
    $db->doDatabaseStuff();
}
catch (SomeSpecificSQLException $e) { // most specific
    echo "caught SomeSpecificSQLException exception: ".$e->getMessage();
}
catch (SQLException $e) { // second most specific
    echo "caught SQLException exception: ".$e->getMessage();
}
catch(Exception $e) { // most general
    echo "caught Exception: ".$e->getMessage();
}
}

```

You may rarely if ever need to get so exquisite as to subclass your own Exception subclasses in your own coding, but it is good to understand how the language works.

Finally, and not the least important point to take away: *an uncaught exception is a fatal error*. If you use third party PHP 5 libraries, you need to be aware of what Exceptions that code can throw. (PHP 5 does not yet support throws clauses in function declarations, and it is not an error to fail to handle a possible exception.) Naturally, PHP 5 provides a way out: you can set a generic Exception handler with:

```
string set_exception_handler ( callback exception_handler )
```

This function sets the default exception handler if an exception is not caught within a try/catch block. Execution will stop after the *exception\_handler* is called. The *exception\_handler* function (you can name it whatever you want) must be defined before calling `set_exception_handler()`. The callback function has to accept one parameter, which will be the exception object that was thrown.

## Sample Application: PHP University

Please download and install the sample database application `php_university.zip` from the class website. It is a rough work in progress, only partially complete, but it contains code that demonstrates object-oriented database access using PEAR and PHP 5 in a hypothetical workshop registration system. For the most part, there are classes called Student, Workshop and Registration that manage database access, and scripts that act as a front end controlling the flow and calling methods on objects of those classes to display, delete, update and insert data in an underlying MySQL database. Play around with this application: do some fetching, inserts,

updates, and deletes on the workshop and student tables through the interface provided. Try login.php and log in with an email address and password – steal any one you like from the student table – and then register and un-registering yourself for some workshops. Then look at the underlying source code and see how it works.

## Assignment

For this assignment you will contribute some code to the PHP\_University demo application. Rewrite workshops.php so that it displays a list of all the workshops and the number of students enrolled in each (unconditionally), providing links for the user to click to see the student roster. Format the workshop data in an HTML table whose four columns from left to right are the name of the workshop, the date and time, the instructor, and the number of students enrolled.

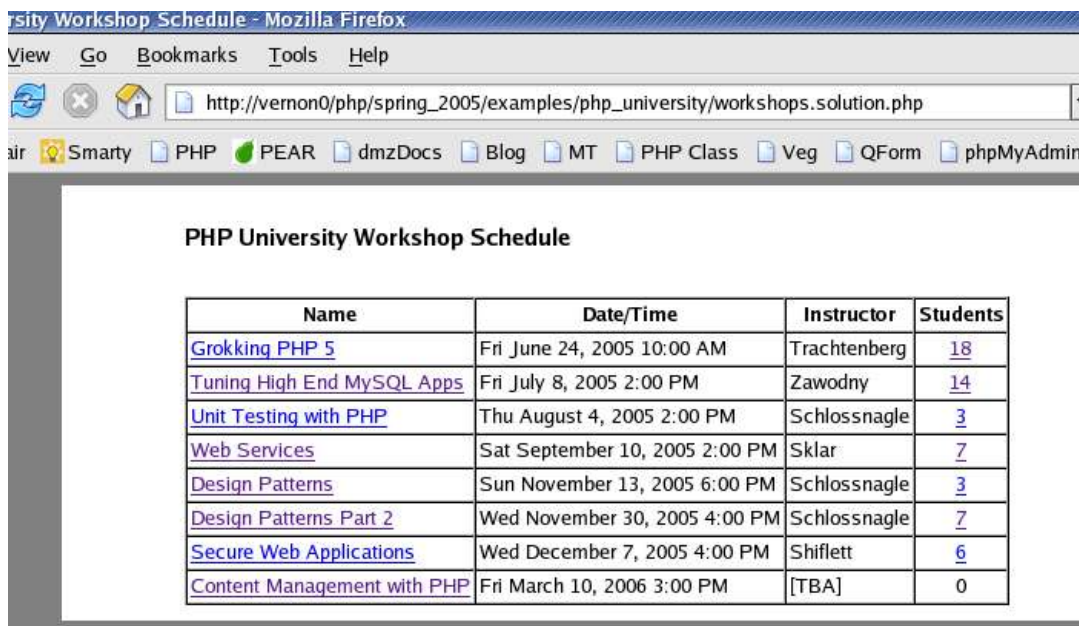
In the leftmost column, the workshop name will be linked to `edit_workshop.php?id=n`, where *n* is the workshop id. When you click on of these links you come to the workshop editing page.

If the number of students is greater than zero, then it is wrapped in a hypertext link pointing back to the script itself (workshops.php) with query string `'?show_students=n'` appended, where *n* is the workshop id number of the workshop record.

If workshops.php is requested with a query string parameter `'show_students=n'`, then it fetches and displays the firstname, lastname and id number for all the students enrolled in the workshop whose id you were passed in the URL. The student names should be linked to `edit_student.php?id=n`, where *n* is the id of each student record.

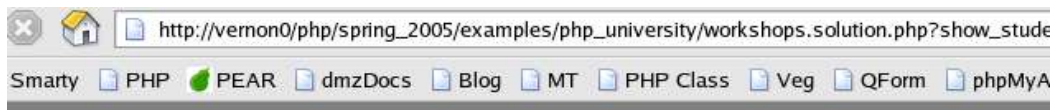
Use methods from the supplied classes for all database access: `Database::getInstance`, `Workshop::fetchWorkshopSchedule()`, and `Registration::getStudentListingForWorkshop()` will do everything you need for this task. Do not forget to run your output through `htmlspecialchars()`.

When your workshops.php is accessed with no query parameters it will look about like this:



Name	Date/Time	Instructor	Students
<a href="#">Grokking PHP 5</a>	Fri June 24, 2005 10:00 AM	Trachtenberg	<a href="#">18</a>
<a href="#">Tuning High End MySQL Apps</a>	Fri July 8, 2005 2:00 PM	Zawodny	<a href="#">14</a>
<a href="#">Unit Testing with PHP</a>	Thu August 4, 2005 2:00 PM	Schlossnagle	<a href="#">3</a>
<a href="#">Web Services</a>	Sat September 10, 2005 2:00 PM	Sklar	<a href="#">7</a>
<a href="#">Design Patterns</a>	Sun November 13, 2005 6:00 PM	Schlossnagle	<a href="#">3</a>
<a href="#">Design Patterns Part 2</a>	Wed November 30, 2005 4:00 PM	Schlossnagle	<a href="#">7</a>
<a href="#">Secure Web Applications</a>	Wed December 7, 2005 4:00 PM	Shiflett	<a href="#">6</a>
<a href="#">Content Management with PHP</a>	Fri March 10, 2006 3:00 PM	[TBA]	0

When you click on the number in the Students column in the row for “Tuning High End MySQL Apps” you should see something like this:



### PHP University Workshop Schedule Admin

14 students enrolled for Tuning High End MySQL Apps

[Aguiar, Christiana](#)  
[Andrews, Elizabeth](#)  
[Bone, Lucy](#)  
[Carlson, Geoffrey](#)  
[Ditaranto, Edna](#)  
[Eriksen, Vigdis](#)  
[Fresco, Betsabe](#)  
[Gashi, Sal](#)  
[Hsu, John](#)  
[LaMothe, Albert](#)  
[Peña, Vicky](#)  
[Schmiedel, Christine](#)  
[Shatova, Ella](#)  
[White, Sarah](#)

Name	Date/Time	Instructor	Students
<a href="#">Grokking PHP 5</a>	Fri June 24, 2005 10:00 AM	Trachtenberg	<a href="#">18</a>
<a href="#">Tuning High End MySQL Apps</a>	Fri July 8, 2005 2:00 PM	Zawodny	<a href="#">14</a>
<a href="#">Unit Testing with PHP</a>	Thu August 4, 2005 2:00 PM	Schlossnagle	<a href="#">3</a>
<a href="#">Web Services</a>	Sat September 10, 2005 2:00 PM	Sklar	<a href="#">7</a>
<a href="#">Design Patterns</a>	Sun November 13, 2005 6:00 PM	Schlossnagle	<a href="#">3</a>
<a href="#">Design Patterns Part 2</a>	Wed November 30, 2005 4:00 PM	Schlossnagle	<a href="#">7</a>
<a href="#">Secure Web Applications</a>	Wed December 7, 2005 4:00 PM	Shiflett	<a href="#">6</a>
<a href="#">Content Management with PHP</a>	Fri March 10, 2006 3:00 PM	[TBA]	0

---

16-Apr-2005 05:29 pm