CSY2028
Web Programming
Topic 12

Tom Butler
thomas.butler@northampton.ac.uk

Topic 12

- Abstracting SQL queries
- Further refinements to the database manipulation functions
- Try/catch statements
- Using Object-Oriented Programming to write a class that can be used to manipulate any database table
- Reusable form HTML
- Reinventing the wheel?

Last Week

- Last week we ended with several functions that could be used to easily manipulate any database table
- There were functions for:
  - Inserting records
  - Selecting records
- The functions could be used to very quickly find/insert the data in any database table

Reusable insert function

```php
function insert($pdo, $table, $record) {
        $keys = array_keys($record);

        $values = implode(', ', $keys);
        $valuesWithColon = implode(', :', $keys);

        $query = 'INSERT INTO ' . $table . ' (' . $values . ') VALUES (:' . $valuesWithColon . ')';

        $stmt = $pdo->prepare($query);

        $stmt->execute($record);
}
```

```php
$person1 = [
        'firstname' => 'John',
        'surname' => 'Smith',
        'email' => 'john@example.org',
        'birthday' => '1989-12-02'
];

insert($pdo, 'person', $person1);

$job1 = [
        'title' => 'Assistant manager',
        'job_ref' => '1333',
        'description' => 'Assistant to the manager',
        'salary' => '25,000'
];

insert($pdo, 'job', $job1);
```

Reusable insert function

- This function allows you to quickly insert a record into any database by providing only the name of the table and the data to be inserted
- The same can be done with UPDATE and DELETE

Reusable update function

```php
function update($pdo, $table, $record, $primaryKey) {

         $query = 'UPDATE ' . $table . ' SET ';

         $parameters = [];
         foreach ($record as $key => $value) {
                $parameters[] = $key . ' = :' .$key;
         }

         $query .= implode(', ', $parameters);
         $query .= ' WHERE ' . $primaryKey . ' = :primaryKey';

         $record['primaryKey'] = $record[$primaryKey];

         $stmt = $pdo->prepare($query);

         $stmt->execute($record);
}
```

Reusable update function

- This works in a very similar way to the insert() function, it takes a table name and some data and generates the relevant SQL query. The arguments are:

```
function update($pdo, $table, $record, $primaryKey) {
```

Reusable update function

- Firstly create the `UPDATE table SET` part of the query and store it in a variable called `$query`

```
function update($pdo, $table, $record, $primaryKey) {

	$query = 'UPDATE ' . $table . ' SET ';
```

Reusable update function

- Then create an array to store each field that will be updated
- This will be in the format:
  - fieldName = :fieldName

```
function update($pdo, $table, $record, $primaryKey) {

        $query = 'UPDATE ' . $table . ' SET ';

        $parameters = [];
        foreach ($record as $key => $value) {
                $parameters[] = $key . ' = :' .$key;
        }
```

Reusable update function

- The $parameters array will store each line of the SET part of the query

```
UPDATE person SET
firstname = :firstname,
surname = :surname,
birthday = :birthday
```

```
$record = [
        'firstname' => 'John',
        'surname' => 'Smith',
        'birthday' => '1990-02-23'
];

$parameters = [];
foreach ($record as $key => $value) {
         $parameters[] = $key . ' = :' .$key;
}

var_dump($parameters);
```

Reusable update function

```
$record = [
        'firstname' => 'John',
        'surname' => 'Smith',
        'birthday' => '1990-02-23'
];

$parameters = [];
foreach ($record as $key => $value) {
         $parameters[] = $key . ' = :' .$key;
}

var_dump($parameters);
```

```
Output:
array (size=3)
  0 => string 'firstname = :firstname' (length=22)
  1 => string 'surname = :surname' (length=18)
  2 => string 'birthday = :birthday' (length=20)
```

Reusable update function

- Each field is placed into an array
- If the loop was used to build a string, e.g.:

```php
$record = [
        'firstname' => 'John',
        'surname' => 'Smith',
        'birthday' => '1990-02-23'
];

$query = '';
foreach ($record as $key => $value) {
         $query .= $key . ' = :' .$key . ',';
}

var_dump($query);
```

- This would generate the string

```
firstname = :firstname, surname = :surname, birthady = :birthday,
```

Reusable update function

```
firstname = :firstname, surname = :surname, birthady = :birthday,
```

- The extra comma at the end would make the query invalid e.g.

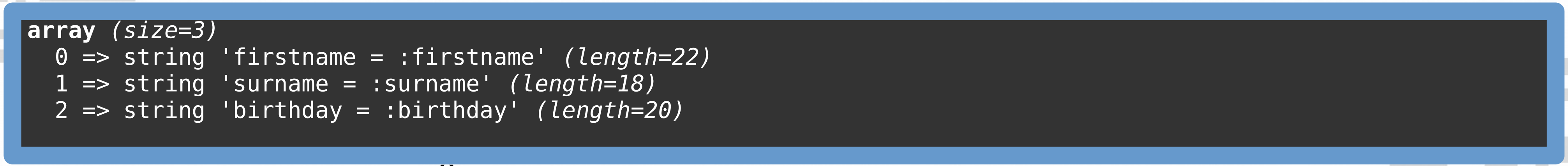
```
UPDATE person
SET firstname = :firstname, surname = :surname, birthady = :birthday,
WHERE id = :primaryKey
```

- Instead of

```
UPDATE person
SET firstname = :firstname, surname = :surname, birthady = :birthday
WHERE id = :primaryKey
```

Reusable update function

- Instead, by putting the fields in an array with the values:
  - firstname = :firstname
  - surname = :surname
  - birthday = :birthday



```
array (size=3)
  0 => string 'firstname = :firstname' (length=22)
  1 => string 'surname = :surname' (length=18)
  2 => string 'birthday = :birthday' (length=20)
```

- The php implode() function can be used to join the values without the extra comma at the end

Reusable update function

```
$record = [
        'firstname' => 'John',
        'surname' => 'Smith',
        'birthday' => '1990-02-23'
];

$parameters = [];
foreach ($record as $key => $value) {
         $parameters[] = $key . ' = :' .$key;
}

$string = implode(', ', $parameters);

echo $string;
```

```
Output:
firstname = :firstname, surname = :surname, birthady = :birthday
```

Reusable update function

- We can use this to build the SET part of the UPDATE query:

```
function update($pdo, $table, $record, $primaryKey) {

        $query = 'UPDATE ' . $table . ' SET ';

        $parameters = [];
        foreach ($record as $key => $value) {
                $parameters[] = $key . ' = :' .$key;
        }

        $query .= implode(', ', $parameters);
```

- The last part is the WHERE clause
- This will need a field and a value

Reusable update function

```
function update($pdo, $table, $record, $primaryKey) {

        $query = 'UPDATE ' . $table . ' SET ';

        $parameters = [];
        foreach ($record as $key => $value) {
                $parameters[] = $key . ' = :' .$key;
        }

        $query .= implode(', ', $parameters);
        $query .= ' WHERE ' . $primaryKey . ' = :primaryKey';

        $record['primaryKey'] = $record[$primaryKey];


}
```

Create the where statement using
the $primaryKey variable
and a placeholder for the value

Now write to the key 'primaryKey'
which is used as the placeholder by
reading the current primary Key

This reads the primary key out of the
$record array

Reusable update function

- The last part is just running the query

```php
function update($pdo, $table, $record, $primaryKey) {

         $query = 'UPDATE ' . $table . ' SET ';

         $parameters = [];
         foreach ($record as $key => $value) {
                $parameters[] = $key . ' = :' .$key;
         }

         $query .= implode(', ', $parameters);
         $query .= ' WHERE ' . $primaryKey . ' = :primaryKey';

         $record['primaryKey'] = $record[$primaryKey];

         $stmt = $pdo->prepare($query);

         $stmt->execute($record);
}
```

Reusable update function

- The update function can now be used to update the data in any table
  - With one caveat: Every field (including the primary key!) must be supplied in the $record array

```
$person1 = [
        'id' => 123,
        'firstname' => 'John',
        'surname' => 'Smith',
        'email' => 'john@example.org'
];
update($pdo, 'person', $person1, 'id');


$book1 = [
        'ISBN' => '1904642748',
        'title' => 'Treasure Island',
        'author' => 'Robert Louis Stevenson'
];

update($pdo, 'book', $book1, 'ISBN');
```

Reusable delete function

- The same can be done with DELETE like we built SELECT (See last week)

```php
function find($pdo, $table, $field, $value) {
	$stmt = $pdo->prepare('SELECT * FROM ' . $table . ' WHERE ' . $field . ' = :value');
	$criteria = [
		'value' => $value
	];
	$stmt->execute($criteria);

	return $stmt->fetch();
}
```

```php
function delete($pdo, $table, $field, $value) {
	$stmt = $pdo->prepare('DELETE FROM ' . $table . ' WHERE ' . $field . ' = :value');
	$criteria = [
		'value' => $value
	];
	$stmt->execute($criteria);
}
```

Reusable functions

- We now have reusable functions for each of main query types
- This allows us to quickly extract a record, make changes to it and save it back:

```
//Load a record where the email is john@example.org
$record = find($pdo,'person', 'email', 'john@example.org')->fetch();

//Update the first name
$record['firstname'] = 'Jonathan';

//Save the record back to the database
update($pdo, 'person', $record, 'id');
```

Refining this further

- There are a couple of places this can be improved:
  - 1) You need to know whether or not to run **insert()** or **update()** depending on whether a record with that ID exists or not
  - 2) You must supply the table name and **$pdo** instance each time you call one of the functions

Insert or update?

- This code will fail if the record with the ID 123 does not yet exist

```php
$person1 = [
        'id' => 123,
        'firstname' => 'John',
        'surname' => 'Smith',
        'email' => 'john@example.org'
];


update($pdo, 'person', $person1, 'id');
```

- However, the insert query will fail if a record with the ID 123 does exist

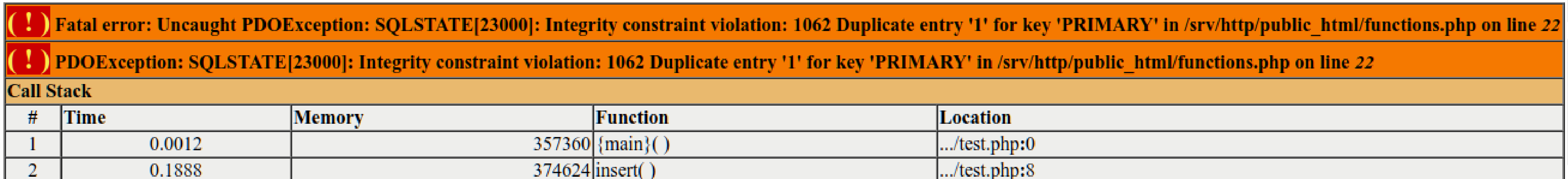
```php
$person1 = [
        'id' => 123,
        'firstname' => 'John',
        'surname' => 'Smith',
        'email' => 'john@example.org'
];


insert($pdo, 'person', $person1);
```

Insert query

- The insert query looks like this:

```php
function insert($pdo, $table, $record) {
        $keys = array_keys($record);

        $values = implode(', ', $keys);
        $valuesWithColon = implode(', :', $keys);

        $query = 'INSERT INTO ' . $table . ' (' . $values . ') VALUES (:' . $valuesWithColon . ')';

        $stmt = $pdo->prepare($query);

        $stmt->execute($record);
}
```

- The query is run at the line $stmt->execute()
- If a record with the inserted ID already exists you'll see an error



```
( ! ) Fatal error: Uncaught PDOException: SQLSTATE[23000]: Integrity constraint violation: 1062 Duplicate entry '1' for key 'PRIMARY' in /srv/http/public_html/functions.php on line 22
( ! ) PDOException: SQLSTATE[23000]: Integrity constraint violation: 1062 Duplicate entry '1' for key 'PRIMARY' in /srv/http/public_html/functions.php on line 22
Call Stack
#   Time      Memory   Function     Location
1   0.0012    357360   {main}( )    .../test.php:0
2   0.1888    374624   insert( )    .../test.php:8
```

Detecting unsuccessful inserts

- The type of error that PDO generates is an *Exception*
- An exception is a special kind of error
- By default, it will stop the execution of any more code and just display the error message on the screen
- However, you can intercept the error with a try/catch statement
- Once the error has been intercepted, the program can continue and perform additional tasks

Try/catch statement

- A try/catch statement is a bit like an if else
- You can put some code in the \`try\` block
- But if it causes an error, the code in the \`catch\` block will be run

```
try {
  //something that might cause an error
}
catch (Exception $e) {
     //Some code that will run if an error occurred in the try block
}
```

Detecting unsuccessful inserts

- The `insert` function causes an error if a record with the provided primary key already exists:

```
$person1 = [
        'id' => '1',
        'firstname' => 'John',
        'surname' => 'Smith',
        'email' => 'john@example.org',
        'birthday' => '1989-12-02'
];

insert($pdo, 'person', $person1);
```

Detecting unsuccessful inserts

- We could use a try-catch block to try inserting the record, but if unsuccessful (because there is an error) to run an UPDATE query instead
- We've already got the functions for insert() and update()

```
$person1 = [
        'id' => '1',
        'firstname' => 'John',
        'surname' => 'Smith',
        'email' => 'john@example.org',
        'birthday' => '1989-12-02'
];

try {
     insert($pdo, 'person', $person1);
}
catch (Exception $e) {
     update($pdo, 'person', $person1, 'id');
}
```

```php
$person1 = [
        'id' => '1',
        'firstname' => 'John',
        'surname' => 'Smith',
        'email' => 'john@example.org',
        'birthday' => '1989-12-02'
];

try {
     insert($pdo, 'person', $person1);
}
catch (Exception $e) {
     update($pdo, 'person', $person1, 'id');
}
```

- This code will *try* to insert the record, but if the insert is not successful because there was an error message, it will run an *update* query to update the record with the given ID

Detecting unsuccessful inserts

- However, taking this a step further it's possible to move the if statement into its own function
- We can write a function called *save* which will save any $record
  - It will insert a new record if the primary key doesn't exist
  - Otherwise it will update the existing record

Save Function

```
function save($pdo, $table, $record, $primaryKey) {
      try {
            insert($pdo, $table, $record);
      }
      catch (Exception $e) {
            update($pdo, '$table, $$record, $primaryKey);
      }
}
```

```
$person1 = [
         'id' => '123',
         'firstname' => 'John',
         'surname' => 'Smith',
         'email' => 'john@example.org',
         'birthday' => '1989-12-02'
];

save($pdo, 'person', $person1, 'id');

$job1 = [
         'id' => '123',
         'title' => 'IT Technician',
         'salary' => '25000',
         'location' => 'Northampton'
];

save($pdo, 'job', $job1, 'id');
```

Save function

- The save function can now be used to quickly and easily insert or update a record in the database
- If the primary key is set and exists in the database it the record will be updated
- Otherwise, a new record will be inserted
  - This will also happen if you don't supply a value for the ID
  - If you have Auto_Increment enabled on the table, providing a no primary key will generate an ID

```php
$person1 = [
        'id' => '123',
        'firstname' => 'John',
        'surname' => 'Smith',
        'email' => 'john@example.org',
        'birthday' => '1989-12-02'
];

//an ID has been provided, it will try to insert a record with the specified ID
//if a record with the id 123 already exists, the record will be updated
save($pdo, 'person', $person1, 'id');

$person2 = [
        'firstname' => 'Sue',
        'surname' => 'Evans',
        'email' => 'sue@example.org',
        'birthday' => '1987-02-24'
];

//Because there is no `id` set, the insert will be successful and if the `id` column is
//auto_increment  then an ID will be generated
save($pdo, 'person', $person2, 'id');
```

Save function

- The save function can now be used in place of any insert or update function.
- We don’t need to distinguish between a new record being added and an existing record being updated
- The save() function will work regardless and gives us a consistent way of writing data to the database

Exercise 1

- Building on Topic 11’s exercise 2, use the new `save()` function to replace any insert/update queries

HTML Forms

- In Topic 6 you saw how to use $_POST directly in the execute() part of a query

```
$stmt = $pdo->prepare('INSERT INTO person (email, firstname, surname)
                       VALUES (:email, :firstname, :surname)
');

$criteria = [
        'firstname' => $_POST['firstname'],
        'surname' => $_POST['surname'],
        'email' => $_POST['email']
];

$stmt->execute($criteria);
```

```
$stmt = $pdo->prepare('INSERT INTO person (email, firstname, surname)
                       VALUES (:email, :firstname, :surname)
');


$stmt->execute($_POST);
```

HTML Forms

- This works as long as your form fields have names that exactly match the field names in the database
- However, there is one small problem with this code:

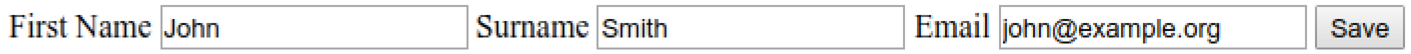
```
$stmt = $pdo->prepare('INSERT INTO person (email, firstname, surname)
                        VALUES (:email, :firstname, :surname)
');

$stmt->execute($_POST);
```

HTML Forms

- When a HTML form is submitted the submit button is pressed
- The $_POST array contains a value for the submit button

```
<form action="" method="POST">
        <label>First Name</label>
        <input type="text" name="firstname" />

        <label>Surname</label>
        <input type="text" name="surname" />

        <label>Email</label>
        <input type="text" name="email" />

        <input type="submit" name="submit" value="Save" />
</form>

<?php

if (isset($_POST['submit'])) {
        var_dump($_POST);
}
```

```
array (size=4)
  'firstname' => string 'John' (length=4)
  'surname' => string 'Smith' (length=5)
  'email' => string 'john@example.org' (length=16)
  'submit' => string 'Save' (length=4)
```



First Name	John	Surname	Smith	Email	john@example.org	Save

HTML Forms

- Because the table doesn't have a field called `submit` it's impossible to pass $_POST directly to the query:

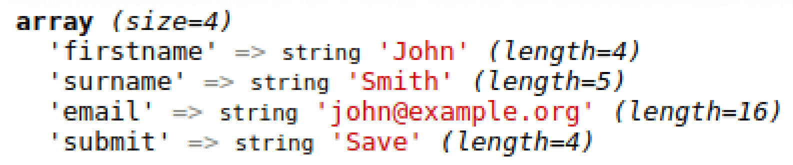
```
$stmt = $pdo->prepare('INSERT INTO person (email, firstname, surname)
                       VALUES (:email, :firstname, :surname)
');

$criteria = [
        'firstname' => $_POST['firstname'],
        'surname' => $_POST['surname'],
        'email' => $_POST['email']
];

$stmt->execute($criteria);
```

```
$stmt = $pdo->prepare('INSERT INTO person (email, firstname, surname)
                       VALUES (:email, :firstname, :surname)
');


$stmt->execute($_POST);
```

HTML Forms

- This will cause an error because the $_POST array contains more fields than there are placeholders in the query

```
$stmt = $pdo->prepare('INSERT INTO person (email, firstname, surname)
                       VALUES (:email, :firstname, :surname)
');


$stmt->execute($_POST);
```



```
array (size=4)
  'firstname' => string 'John' (length=4)
  'surname' => string 'Smith' (length=5)
  'email' => string 'john@example.org' (length=16)
  'submit' => string 'Save' (length=4)
```

HTML Forms

- One way to resolve this is to remove the submit button from the $_POST array

```php
$stmt = $pdo->prepare('INSERT INTO person (email, firstname, surname, birthday)
                       VALUES (:email, :firstname, :surname, :birthday)
');

unset($_POST['submit']);


$stmt->execute($_POST);
```

- This will remove the submit index from the array and the query will execute correctly because the $_POST array has the same keys (and number of keys) as placeholders in the query

HTML Forms

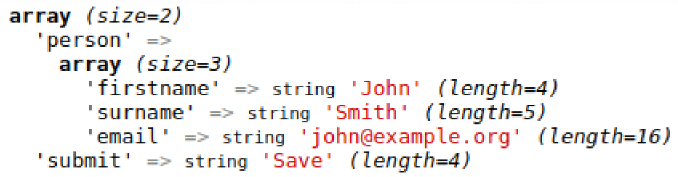
- Another way to resolve this is to change the structure of the form
- You can make form fields into arrays by putting them inside square brackets:

```
<form action="" method="POST">
        <label>First Name</label>
        <input type="text" name="person[firstname]" />

        <label>Surname</label>
        <input type="text" name="person[surname]" />

        <label>Email</label>
        <input type="text" name="person[email]" />

        <input type="submit" name="submit" value="Save" />
</form>

<?php

if (isset($_POST['submit'])) {
        var_dump($_POST);
}
```

```
array (size=2)
  'person' =>
    array (size=3)
      'firstname' => string 'John' (length=4)
      'surname' => string 'Smith' (length=5)
      'email' => string 'john@example.org' (length=16)
  'submit' => string 'Save' (length=4)
```

HTML Forms

- This will create a two-dimensional array
- The $_POST array now has two keys:
  - Person
  - Submit
- The person index contains an array with the fields firsname, surname and email



```
array (size=2)
  'person' =>
    array (size=3)
      'firstname' => string 'John' (length=4)
      'surname' => string 'Smith' (length=5)
      'email' => string 'john@example.org' (length=16)
  'submit' => string 'Save' (length=4)
```

HTML Forms

- The data concerning the person can be read from $_POST['person'] and accessed using:
  - $_POST['person']['firstname'];
  - $_POST['person']['surname'];
- Etc
- Or used directly:

```
$stmt = $pdo->prepare('INSERT INTO person (email, firstname, surname, birthday)
                       VALUES (:email, :firstname, :surname, :birthday)
');


$stmt->execute($_POST['person']);
```

Reusable HTML forms

- A form for *adding a record* is almost identical to a form for *editing a record*
- Edit forms must provide the ID of the record being edited and pre-fill the contents of the boxes
- However, all the fields for the add form are present on edit form!
- Our add.php and edit.php are practically identical. They have almost the same HTML form and almost the same logic (if the form is submitted, send the data to the database)

Reusable HTML forms

- The problem with this approach is that if a field is added to the database, e.g. favouriteColour, we need to open up add.php and edit.php, adding the HTML code for the form with the new input, and also possibly change the query code

HTML Forms

- An edit form will look something like this:

```
<?php

//Find a person record from the database using the id supplied by $_GET['id']
$record = find($pdo, 'person', $_GET['id'], 'id');
?>
<form action="" method="POST">

        <input type="hidden" name="person[id]" value="<?php echo $record['id']; ?>" />

        <label>First Name</label>
        <input type="text" name="person[firstname]" value="<?php echo $record['firstname']; ?>" />

        <label>Surname</label>
        <input type="text" name="person[surname]" value="<?php echo $record['surname']; ?>" />

        <label>Email</label>
        <input type="text" name="person[email]" value="<?php echo $record['email']; ?>" />

        <input type="submit" name="submit" value="Save" />
</form>

<?php

if (isset($_POST['submit'])) {
        update($pdo, 'person', $_POST['person'], $id);
}
```

HTML Forms

- With a small tweak, this can be changed to work as an ADD form or an EDIT form
- If $_GET['id'] is set, it will edit the record if, it's not a record will be added

```
<?php
//Find a person record from the database using the id supplied by $_GET['id']
if (isset($_GET['id'])) {
      $record = find($pdo, 'person', $_GET['id'], 'id');
}
else {
      $record = false;
}
?>
<form action="" method="POST">

        <input type="hidden" name="person[id]" value="<?php if ($record) echo $record['id']; ?>" />

        <label>First Name</label>
        <input type="text" name="person[firstname]" value="<?php if ($record) echo $record['firstname']; ?

        <label>Surname</label>
        <input type="text" name="person[surname]" value="<?php if ($record) echo $record['surname']; ?>" /

        <label>Email</label>
        <input type="text" name="person[email]" value="<?php if ($record) echo $record['email']; ?>" />

        <input type="submit" name="submit" value="Save" />
</form>

<?php

if (isset($_POST['submit'])) {
        save($pdo, 'person', $_POST['person'], 'id');
}
```

First check to see whether
The ID variable is set in the URL

If not, set the $record variable
to false

Note the hidden input
For the id!

Only set the value of the
form fields if the record
was set

Use the new save function from earlier
that inserts or updates depending on
whether the ID exists or not

Exercise 2

- Continue refining the code from Exercise 1:
  - 1) Amend edit.php so it can be used to either add a new record or update an existing one
    - *edit.php?id=123* should edit the record with the id 123 but just going to *edit.php* (without an id) should allow inserting a new record
  - 2) delete *add.php* and amend any links to it to go to edit.php as it's no longer required
  - 3) Make the same change to *addmessage.php* so that it can be used to either add or edit a message
  - 4) Add an extra field, *favouriteColour* to the `person` table and amend the form to include this field. You should only have to add the field to *edit.php*

Objects and classes

- The second problem with the insert(), update(), find() and delete() functions is that they all need to be supplied the table name and $pdo instance every time they are used.
- Some of the functions also need to know the name of the primary key field
- By using a class, these can be supplied once and reused

```
//Find person with id 123
$record = find($pdo, 'person', 'id', 123);

//Find all the records in the person table
$records = findAll($pdo, 'person');

//Delete the field where the id is 123
delete($pdo, 'person', 'id', 123);

//Delete the field where the id is 123
save($pdo, 'person', $_POST['person'], 'id');
```

Regardless of which function is called, You need to supply it at minimum the `$pdo` variable and the name of the table

It would be better to avoid this repetition

- By moving these functions into a *class* , this repeated information can be removed

Classes and Objects

- Just the find and delete functions for now

```
class DatabaseTable {
        public function find($pdo, $table, $field, $value) {
                $stmt = $pdo->prepare('SELECT * FROM ' . $table . ' WHERE ' . $field . ' = :value');
                $criteria = [
                    'value' => $value
                ];
                $stmt->execute($criteria);

                return $stmt->fetch();
        }

        public function delete($pdo, $table, $field, $value) {
                $stmt = $pdo->prepare('DELETE FROM ' . $table . ' WHERE ' . $field . ' = :value');
                $criteria = [
                        'value' => $value
                ];
                $stmt->execute($criteria);

                return $stmt->fetch();
         }
}
```

```
$database = new DatabaseTable();

//Find person with id 123
$record = $database->find($pdo, 'person', 'id', 123);

//Delete the field where the id is 123
$database->delete($pdo, 'person', 'id', 123);
```

Constructors

- You can add a special function called a *constructor* that gets run when the object is created
- This can optionally take arguments
- To define a *constructor*, create a function called __construct()
- Note: That is the word *construct* prefixed by **two** underscores

Constructors

- The constructor is automatically called when an object is created

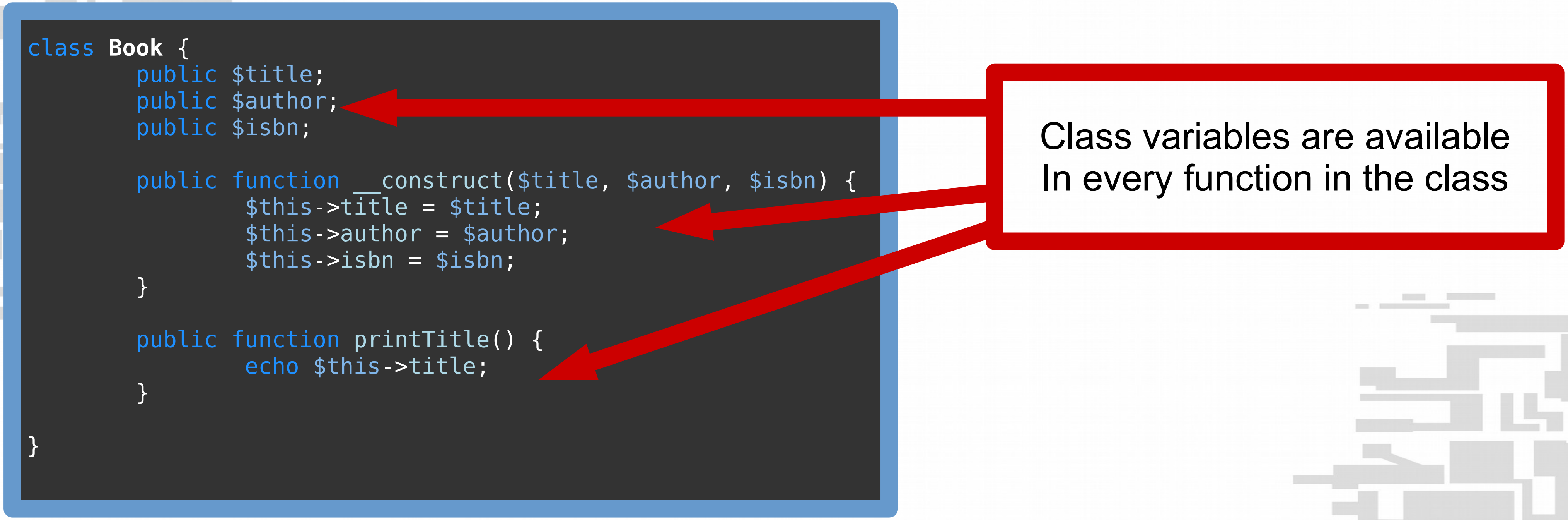
```php
class Book {
        public $title;
        public $author;
        public $isbn;

        public function __construct() {
                echo 'Constructor has been run';
        }
}

$book1 = new Book();

$book2 = new Book();
```

```
Output:
Constructor has been run
Constructor has been run
```

Constructors

- The constructor can take *arguments* like any other function
- You can set the constructor arguments when an object is created
- For objects that store data such as the instances of the *book class*, constructors can be used as a shorthand to enable creating an object and setting its values with considerably less code

Constructors



```
class Book {
        public $title;
        public $author;
        public $isbn;

        public function __construct($title, $author, $isbn) {
                $this->title = $title;
                $this->author = $author;
                $this->isbn = $isbn;
        }

        public function printTitle() {
                echo $this->title;
        }

}
```

Class variables are available
In every function in the class

Constructors

```
class Book {
        public $title;
        public $author;
        public $isbn;

        public function __construct($title, $author, $isbn) {
                $this->title = $title;
                $this->author = $author;
                $this->isbn = $isbn;
        }

        public function printTitle() {
                echo $this->title;
        }

}
```

Arguments are only available in function they are declared in

To make arguments available in other functions, you must store the values in class variables

Constructors

```
class Book {
        public $title;
        public $author;
        public $isbn;

        public function __construct($title, $author, $isbn) {
                $this->title = $title;
                $this->author = $author;
                $this->isbn = $isbn;
        }

        public function printTitle() {
                echo $this->title;
        }

}
```

To do this, you need to write to the class variables by referencing them using $this->variableName

Only then will those variables be available in other functions

Constructors

- Once you have declared the constructor, when you create an instance of the class you must supply values which will be used as those constructor arguments

```
book1 = new Book('Moby Dick', 'Herman Melville','0007925565');

$book1->printTitle();
```

- Which is the same as:

```
book1 = new Book();
$book1->title = 'Moby Dick';
$book1->author = 'Herman Melville';
$book1->isbn = '0007925565';

$book1->printTitle();
```

Classes

- A constructor can be added to the DatabaseTable class that takes the $pdo instance and the table name

```
class DatabaseTable {
        private $table;
        private $pdo;

        public function __construct($pdo, $table) {
                $this->pdo = $pdo;
                $this->table = $table;
        }
```

Objects and Classes

- Now, the delete and find functions can be modified to use the class variables instead of arguments:

```php
class DatabaseTable {
        private $table;
        private $pdo;

        public function __construct($pdo, $table) {
                $this->pdo = $pdo;
                $this->table = $table;
        }

        public function find($field, $value) {
                $stmt = $this->pdo->prepare('SELECT * FROM ' . $this->table . ' WHERE ' . $field . ' = :value');
                $criteria = [
                    'value' => $value
                ];
                $stmt->execute($criteria);

                return $stmt->fetch();
        }

        public function delete($field, $value) {
                $stmt = $this->pdo->prepare('DELETE FROM ' . $this->table . ' WHERE ' . $field . ' = :value');
                $criteria = [
                        'value' => $value
                ];
                $stmt->execute($criteria);

                return $stmt->fetch();
         }
}
```

Objects and classes

- And can be used to quickly query or delete from any table.

```
$personTable = new DatabaseTable($pdo, 'person');

//Find person with id 123
$record = $personTable->find('id', 123);

//Delete the person where the id is 123
$personTable->delete('id', 123);

//And used with other tables:

$jobsTable = new DatabaseTable($pdo, 'job');

//Find job with id 100
$record = $jobsTable->find('id', 100);

//Delete the job where the id is 222
$jobsTable->delete('id', 222);
```

The find and delete functions don't need to be given the table name and database connection each time they are used

Further extension?

- Quickly loop through a filtered list of records?

```
$authors = new DatabaseTable($pdo, 'author');

foreach ($authors->find('surname', 'Stevenson') as $author) {
        echo '<p>' . $author['firstname'] . '</p>';
        echo '<p>' . $author['surname'] . '</p>';
}
```

- And maybe even:

```
$authors = new DatabaseTable($pdo, 'author');

foreach ($authors->find('surname', 'Stevenson') as $author) {
        echo '<p>' . $author['firstname'] . '</p>';
        echo '<p>' . $author['surname'] . '</p>';


      foreach ($authors['books'] as $book) {
              echo '<p>' . $book['title'] . '</p>';
      }
}
```

Reinvent the wheel?

- This kind of tool is called an *Object-Relational-Mapper*
- There are many libraries out there that do this:
  - Doctrine - http://www.doctrine-project.org/
  - Propel - http://propelorm.org/
  - Maphper (my own implementation)
    https://github.com/Level-2/Maphper

Reinvent the wheel?

- It's often said that it's a bad idea to reinvent the wheel
- This is usually true
- However, it depends what your goals are:
  - If you're trying to quickly develop some software, using someone else's code is probably the better choice
  - If you're trying to learn how to code, it's usually better to try it yourself!

Reinventing the wheel

- Sometimes it's better to try something for yourself:
  - The current solution may not quite meet your needs
  - Just because someone else has solved the problem doesn't mean they've done it in a good way
  - You'll get a much better understanding of the underlying problem being solved
  - If the code breaks, you'll find it much easier to fix if you're the one who wrote it
  - **The more you code, and the wider variety of problems you solve, the better programmer you'll become**

Reinventing the wheel

- By attempting to solve the problem yourself you will better understand someone else's solution
- Just because someone else has already solved the problem, doesn't mean they solved it in the only, or best way
- You will then be able to evaluate your own code and compare it to someone else's
  - "I never thought of doing that!"
  - "Why would they do it that way? My way is simpler!"
- There is more than one way to solve any programming problem!

Exercise 3

- 1) Move all the functions, \`findAll\`, \`find\`, \`save\`, \`insert\`, \`update\`, \`delete\` into a class called DatabaseTable
- 2) Use the class throughout the sample project you've been working on.
  - Hint: You can put it in one file and \`require\` it where needed
  - Hint: To call a function inside the same class you can use $this→functionName() e.g. if your table has a function called \`update\` you can call it using the code \`$this->update()\`

Sample class usage

```
$personTable = new DatabaseTable($pdo, 'person');

//Find the person with the is 123
$person1 = $personTable->find('id', 123);

//update that person’s surname
$person1['surname'] = 'Jones';
$personTable->save($person1);

//Insert a new record into the table
$person2 = [
        'firstname' => 'Tom',
        'surname' => 'butler',
        'email' => 'thomas.butler@northampton.ac.uk'
];

$personTable->save($person2);
```