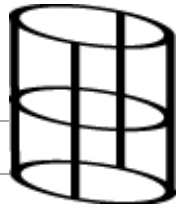
ObjectDB 4.0 User Manual

First steps and API guide



Written by Salvi Pascual, translation by Yasmila Sealy

Contents

Introduction

Accessing to databases in PHP, not compatible with object-oriented paradigm is complex when it is needed to store and retrieve objects instead of separate variables. Many programmers break the encapsulation of code which could be well modeled and easily maintainable; others create their own methods of data access, which costs much of the development of the whole system.

ObjectDB creates a general architecture for data access, which (through inheritance) can be increased to support specific methods for every application. With this library the programmer can obtain, save, and modify (among other operations) objects directly from the database without having to memorize the extensive API of PHP and possessing high knowledge of SQL. ObjectDB helps to write clearer code (and therefore more maintainable), formable according to object-oriented methodologies (currently the most used) and makes the programmer completely independent of the database system to use

ObjectDB is designed to reduce work to a minimum. In use, tedious operations such as creating relationships between tables and finding the last inserted tuple, become routine and use a few lines of code. Although it might be thought that the work of an extra layer of abstraction slows down the application, caching mechanisms of ObjectDB avoid redundant operations and streamline the work so that no visible notice of time delays.

Getting Started

The following is a detailed example of how to configure and launch an application that performs simple read-write access to a database using ObjectDB. It was developed with PHP v5.x and MySQL v5.x; it must work for other versions of MySQL and other database systems, depending of the ObjectDB library used. Because of the poor support provided by PHP v4.x for object-oriented applications, ObjectDB does not work for these versions or lower.

Three basic actions should be taken to create easy access to the database:

1. Creating tables in the database
2. Creating ObjectDB classes
3. Setting up the access of ObjectDB to the database

Creating tables in the database

When creating tables in the database, the only absolute requirement is that the key of the table is numeric, ever-increasing and by the name of id_nombretabla. It can be created a table supported by ObjectDB, for a MySQL database with the following script in SQL language:

```
CREATE TABLE student (
      id_student INT(11) KEY AUTO_INCREMENT,
      identification_number VARCHAR(11) NOT NULL,
      name varchar(50) NOT NULL,
      age INT(2) NULL,
      preferences TEXT
);
```

In the above example, note that the field *'identification_number'* is an alternative if an index, that does not meet the requirements expected by the library, is needed.

Creating ObjectDB classes

All classes that store data must extend ODBObject class, defined in the core of ObjectDB. This class, at the same time as serves as a basis for encapsulating information, has special features that facilitate the work of the programmer. The subject will be

discussed thoroughly in later sections. Besides the above, each attribute of the class will be defined as public and its name must match the name of the table fields. By convention the order of attributes must also match the order of the fields in the table, but the above is not required.

Below is shown a PHP language sample about how it would be defined a class to interact with the table defined above:

```
class student extends ODBObject{
      public $identification_number;
      public $name;
      public $age;
      public $preferences;
}
```

Although not essential, by convention each class should be described in a file by the name of *nombre_clase.php*; for the case of the above, it would be *student.php*. This file may begin with the inclusion of *ODBObject.php* file using this statement:

```
require_once "pathToFile/ODBObject.php";
```

Setting up the access of ObjectDB to the database

The first lines of *ODBConnection* class are static parameters which must be configured for each server. These parameters can be set dynamically, increasing the security of applications using direct loading of a file.

```
...
/* start - configure */
static public $host   = "localhost";
static public $port   = "3306";
static public $dbname = "test";
static public $user   = "root";
static public $pass   = "";
/* end - configure */
...
```

The explanation for the previous parameters is as follows:

```
$host
```

```
      Address (IP or DNS) of the server where the database is (localhost for the machine itself).
$port
      Port for accessing the database
$dbname
      Name of the database
$user
      Username to access the database
$pass
      Password to access the database
```

After configuring the connection to the database, set the table and ObjectDB class, simple and functional read-write accesses can be performed. This requires creating a new file called *test.php* with the following content:

```
// library should be included first in the script
require_once "../ObjectDB/ObjectDB-mysql-v3.0.php";
require_once "student.php";

// create and set values for a new student
$student = new student();
$student->setDataFromParamsList(3025,'Salvi Pascual',24,'color=orange,grossery=pie,serie=futurama');

// save ths new student in database
$student->save();

// load all students saved until now
$db = new ObjectDB();
$students_list = $db->getObjs('student');

// print students
echo 'List of students in database';
for($i=0; $i<count($students_list); $i++)
      echo $students_list[$i];
```

Preceding code creates a new student, assigns values to it, passing them as sorted parameters based on the fields in the table (excluding the key field) to *setDataFromParamsList* method and then saves the object in the database. Finally it creates a new *ObjectDB* object and uses it to load from the database all saved students up to now and then they are displayed. In sections below will be explained in greater detail the steps in this process.

Advanced Studies

Debugging the content of an object

ObjectDB may display the contents of an object and simplify for programmers the debugging process. Any object that inherits from *ODBObject*, when is displayed on screen (via echo or printf) produces a table with the relationship field name and value. In the example below shows the entered code and the resulting table.

```
require_once "../ObjectDB/ObjectDB-mysql-v3.0.php";
require_once "student.php";

// create and set values for a new student
$student = new student();
$student->setDataFromParamsList(3025,'Salvi Pascual',24,'color=orange,grossery=pie,serie=futurama');

// print this object in debug mode
echo $student;
```

The table below shows the output of the previous script

id_student	id_number	name	age	preferences
1	3025	Salvi Pascual	24	color=orange,grossery=pie,serie=futurams

The debugging algorithm's output may be changed by overwriting the __ *toString* method of the *ODBObject* class. After it is overwritten, the text string returned by the new method will be displayed when a variable is printed. The following code shows how this could be done for the student class, defined in the previous section.

```
class student extends ODBObject{
      public $identification_number;
      public $name;
      public $age;
      public $preferences;

      function __toString(){
            return 'My identification number is:' . $this->identification_number;
      }
```

```
}
```

Fill in the attributes of an object

There are three ways to fill the attributes of an object in ObjectDB:

1. **Insert data manually:** This is useful for modifying one or a few attributes of the object or for objects with a small amount of attributes. Not recommended for use as it enlarges and obfuscates the resulting code.
2. **By the *setDataFromArray* method of the *ODBObject* class:** Allows filling the attributes of the object using a string array passed by parameter to the setDataFromArray method. Mainly it is used to create objects with information obtained dynamically.
3. **By using the setDataFromParamsList method of the ODBObject class:** Fills in the attributes of the object with the parameters passed to the setDataFromParamsList method. It is comfortable and its use is recommended for most situations, however care must be taken to insert the right amount of parameters and in the order defined in the table.

No matter which way it could be used, it should never be tried to insert/change the key attribute of the table, which could, depending on the situation, generate an error or include unwanted behavior. Here is a code that describes the three ways to fill/modify attributes of an object.

```
require_once "../ObjectDB/ObjectDB-mysql-v3.0.php"; // this will include first
require_once "student.php";

// create a new student object, with blank attributes
$student = new student();

// change attributes manually
$student->identification_number = '3025'; // changes is allow because this is not the table key
$student->name = 'Salvi Pascual';
$student->age = 37;
$student->preferences = 'color=orange,grossery=pie,serie=futurama';

// change attributes by setDataFromArray
$attributes = Array('3025','Salvi Pascual','24','color=orange,grossery=pie,serie=futurama');
$student->setDataFromArray($attributes);

// change attributes by setDataFromParamsList
```

```
$student->setDataFromParamsList(3025,'Salvi Pascual',24,'color=orange,grossery=pie,serie=futurama');
```

Send a custom query to the server

However ObjectDB eliminates highly writing code in the SQL language, and encourages using predefined functions that cover most of the needs of the programmer, sometimes it is needed to send queries made by the programmer to the server to perform complex searches. It is for this reason that the *query* method of the *ObjectDB* controller class is used. In the following example, first it is obtained and displayed the top ten students whose age is over 25 years and prefer the orange color and then deletes the last student whose favorite serial is 24 hours (nothing personal).

```
require_once "../ObjectDB/ObjectDB-mysql-v3.0.php"; // this will include first
require_once "student.php";

// load the 10 first students with age>25 and orange as color preference
$db = new ObjectDB();
$students_list = $db->query("SELECT * FROM student WHERE age>'25' AND preferences LIKE '%color=orange%'
LIMIT 10");

// print students
echo 'List of students in database with age>25 and orange as color preference';
for($i=0; $iquery("DELETE FROM student WHERE preferences LIKE '%serie=24 hours%' ORDER BY id_student DESC
LIMIT 1");
```

Note that for the case of SELECT type queries the method returns an array of objects (such as the method *getObjs*) for queries unanswered, such as DELETE, INSERT, etc. the method returns NULL.

Modify saved object

When an object is saved in the database, its value can be uploaded and changed. This is necessary to obtain the object by using the *getObjs* function of the *ObjectDB* controlling class (exemplified in the previous section), change their values and save it again by using the *save* function of the *ODBObject* class (also shown in the previous section) or by *saveObj* function of the *ObjectDB* controlling class. The code then takes place due to the use of this latter form, not illustrated previously and continues the previous example.

```
require_once "../ObjectDB/ObjectDB-mysql-v3.0.php"; // this will include first
```

```
require_once "student.php";

// load all saved students
$db = new ObjectDB();
$students_list = $db->getObjs('student');

// catch first load student
$student = $students_list[0];

// change students parameters
$student->name = 'Cristobal Colón';
$student->age = 37;

// save changed student
$db->saveObj($student);
// $student->save(); // this works too

// show updated student
echo $student;
```

Get the last object saved

In general it is necessary to load the last stored object in the database. This functionality is extremely useful when, for example, need show only the last comment from a user or most current news. Versatile mechanisms exist to perform this action with ObjectDB, for example, to load all items of a table by means of *getObjs* and go through the list that the method returns to the end, or (in an effortless perspective) to implement the *query* method (previously seen) with the SQL language code shown below:

```
$db = new ObjectDB();
$result = $db->query("SELECT * FROM student ORDER BY id_student DESC LIMIT 1");
```

But to really take advantage of ObjectDB cache resources, and minimize accesses to the database, it is preferable to use the *getLastObject* method of the *ObjectDB* controlling class.

```
require_once "../ObjectDB/ObjectDB-mysql-v3.0.php"; // this will include first
require_once "student.php";
```

```
// load the last saved object
$db = new ObjectDB();
$last_student = $db->getLastObject('student');

// print the last saved student
echo $last_student;
```

Delete an object from the database

There are two ways to eliminate a single object from the database. The first is to use the *remove* method, typical of the *ODBObject* class and existing for any object that inherits from it. The second one is by the *removeObj* method of the *ObjectDB* controlling class, which must have as a parameter the object to delete. Then the two first students of the table are removed, if both exist.

```
require_once "../ObjectDB/ObjectDB-mysql-v3.0.php"; // this will include first
require_once "student.php";

// load all students in database
$db = new ObjectDB();
$students_list = $db->getObjs('student');

// remove first student
$students_list[0]->remove();

// remove second student
$db->removeObj($students_list[1]);
```

Removing multiple objects of the database

By using the ways discussed above a single object can be deleted by a reference to it. This is inefficient if the purpose consists in deleting all entries in a table, or many entries that satisfy a given condition. For that ObjectDB provides the *remove* method, located in the controlling class *ObjectDB*, which should be given as parameters the name of the table on which to act and an optional expression in SQL language to filter the removal. Failure to pass this expression to *remove* method, it will delete all such stored objects in the database.

```
require_once "../ObjectDB/ObjectDB-mysql-v3.0.php"; // this will include first
require_once "student.php";

// remove all student with age>32 years
$db = new ObjectDB();
$db = $db->remove('student',"age>'32'");
```

Relations between tables

One of the main capabilities of ObjectDB is the ease to create relationships between tables in the database without modifying the established tables' model and objects. The main objectives of ObjectDB, is that the programmer is able to create, delete and access relationships between tables with a few lines of code, and using a simple API to learn and use without having advanced knowledge of SQL. Behind the ease of use to relate tables, is once again the cache engine of ObjectDB. This highly reduces the accesses to the database server and thereby raises the final performance of applications which use the library.

Create a relationship table

To relate two tables ObjectDB, the first step is to create a table of relationship (or nexus) between the two tables to connect. The relation table name must consist of the string *relation*, followed by the character "underscore" (_), then the name of the first table to relate, again the underscore character and finally the name of the second table to relate. As an example, the name of a table link between the *student* table and a new table named teacher would be *relation_student_teacher*. The order of the names of the tables that make up the relation table is irrelevant, so it might as well to write *relation_teacher_student* or *relation_student_teacher*.

Relation tables must consist of three fields:

Key to the link table

> It represents the key field of the relation table. This fulfills the same requirements as the rest of the tables; must be numeric, autoincrementable and by the name of *id_nombretabla*. As an example, the table *relation_student_teacher* would have as key name *id_relation_student_teacher*.

Key to the first table to relate

This field is not a key attribute; it has the same name as the key to the first table to relate. For example, to the nexus table *relation_student_teacher*, this field would be *id_student*.

Key to the second table to relate

This field is not a key attribute; it has the same name as the key to the second table to relate. For example, to the nexus table *relation_student_teacher*, this field would be *id_teacher*.

The example below shows a snippet of code in SQL language, optimized for MySQL v5.x generated by the *student* table (used so far), the new table *teacher* and the link table to relate the two.

```
CREATE TABLE student (
      id_student INT(11) KEY AUTO_INCREMENT,
      identification_number VARCHAR(11) NOT NULL,
      name varchar(50) NOT NULL,
      age INT(2) NULL,
      preferences TEXT
);
```

```
CREATE TABLE teacher (
      id_teacher INT(11) KEY AUTO_INCREMENT,
      professorship_number VARCHAR(11) NOT NULL,
      name varchar(50) NOT NULL,
      experience_time INT(2) NULL,
      marital_status TINYINT(1),
      last_university_payment_date DATE
);
```

```
CREATE TABLE relation_student_teacher (
      id_relation_student_teacher INT(11) KEY AUTO_INCREMENT,
      id_student INT(11),
      id_teacher INT(11)
```

```
);
```

Below is the content for the *teacher.php* file, which defines the class corresponding to the table with the same name. The *student* class, located in the file *student.php,* is shown in previous sections. No need to create a class that represents the relation table, due to the process of working with relations never required to instantiate it.

```
// constants for marital status
define("SINGLE",  0x01, true);
define("MARRIED",       0x02, true);
define("DIVORCED",      0x03, true);
define("WIDOWED",       0x04, true);

class teacher extends ODBObject{
      public $professorship_number;
      public $name;
      public $experience_time;
      public $marital_status; // constants
      public $contract_date;  // date
}
```

Define a relationship between two objects

The method *addRelation* of the *ODBObject* class defines a relationship between two objects. An object of the *ODBObject* type must be passed as parameter to this method, existing in the database (it must be saved before) and which is previously created a relation table. If any of the objects, which are tried to connect, does not exist in the database, an exception is thrown *ODBEObjectNotInDatabase*; as ObjectDB policy does not save objects without the direct request of the developer. Below is an example of how to relate two objects, *student* and *teacher* type, defined previously.

```
require_once "../ObjectDB/ObjectDB-mysql-v3.0.php"; // this will include first
require_once "student.php";
require_once "teacher.php";

// create, set values and save a new student
$student = new student();
$student->setDataFromParamsList(3025,'Salvi Pascual',24,'color=orange,grossery=pie,serie=futurama');
$student->save();
```

```
// create, set values and save a new teacher
$teacher = new teacher();
$teacher->setDataFromParamsList('math201','Stephen W. Hawking',37,MARRIED,date("Y-m-d"));
$teacher->save();

// create relation between
$student->addRelation($teacher); // same than: $teacher->addRelation($student);
```

Since a student is related to his teacher the same way as the teacher with his student, the *$student->addRelation($teacher)* line can be exchanged for *$teacher->addRelation($student)* and get the same result. It will throw an exception if a relationship for two objects is saved more than once.

Get the relationship to an object

After relationships between objects have been created, the next is to learn how to consult them easily. For this purpose *ObjectDB* offers the *relations* method of the *ODBObject* class. It receives as parameter a string representing the name of the class to query, and returns the list of objects related to the object from which was queried. In the example below this operation is illustrated.

```
require_once "../ObjectDB/ObjectDB-mysql-v3.0.php"; // this will include first
require_once "student.php";
require_once "teacher.php";

// create, set values and save a new student
$student = new student();
$student->setDataFromParamsList(3025,'Salvi Pascual',24,'color=orange,grossery=pie,serie=futurama');
$student->save();

// values of university teachers
$teachers_values = Array(
      Array('math201','Stephen W. Hawking',37,MARRIED,date("Y-m-d")),
      Array('math201','Albert Einstain',68,MARRIED,date("Y-m-d")),
      Array('math201','Nicolas Coppernico',37,WIDOWED,date("Y-m-d")),
      Array('math201','Felix Varela',37,SINGLE,date("Y-m-d")),
      Array('math201','Meredick Jhones',37,DIVORCED,date("Y-m-d"))
);

// create and relation teachers
```

```
$teachers = Array(count($teachers_values));
for ($i=0; $isetDataFromArray($teachers_values[$i]);
      $teachers[$i]->save();
      // relation each teacher
      $student->addRelation($teachers[$i]);
}

//load and show all relations
$teachers = $student->relations('teacher');
foreach ($teachers as $teacher) echo $teacher;
```

To determine whether two objects are related

Sometimes it is useful to know if two objects are related to each other, for this the *ODBObject* class provides the method *isRelationedWith*, to which an object of the *ODBObject* type is passed as a parameter and returns *true* if any relationship exists and *false* otherwise. As *addRelation*, this method raises an exception *ODBEObjectNotInDatabase* whether any items are not found in the database. Below is an example of how to work with the method *isRelationedWith*.

```
require_once "../ObjectDB/ObjectDB-mysql-v3.0.php"; // this will include first
require_once "student.php";
require_once "teacher.php";

// create, set values and save a new student
$student = new student();
$student->setDataFromParamsList(3025,'Salvi Pascual',24,'color=orange,grossery=pie,serie=futurama');
$student->save();

// create, set values and save a new teacher
$teacher = new teacher();
$teacher->setDataFromParamsList('math201','Stephen W. Hawking',37,MARRIED,date("Y-m-d"));
$teacher->save();

$student->isRelationedWith($teacher); // this will return false

$student->addRelation($teacher);
$student->isRelationedWith($teacher); // now will return true
```

It is possible to use the *relations* method to get the list of teachers related to a specific student and then to go round it asking if anyone is the sought teacher, but this approach unnecessarily increases the number of lines of code and does not exploit the cache engine of *ObjectDB* to the maximum, so it is advisable not to develop this solution. Like using the method *addRelation* has the same result to write *$student->addRelation($teacher);* or *$teacher->addRelation($student);*.

Delete a relationship between two objects

The method *removeRelation* of the *ODBObject* class is used to delete a relationship between two objects. An object of the *ODBObject* type is passed as parameter to this method, with which is wanted to disconnect the object in use. This method will throw an exception *ODBEObjectNotInDatabase* if any of the objects are not in the database. It will also throw an exception of the *ODBENoRelationBetween* type if there is not relationship between two objects. The following example shows how to delete a relationship using *removeRelation*.

In the example below causes the same result to write

```
$student->removeRelation($teacher); or $teacher->removeRelation($student);
```

```
require_once "../ObjectDB/ObjectDB-mysql-v3.0.php"; // this will include first
require_once "student.php";
require_once "teacher.php";

// load the last saved student
$db = new ObjectDB();
$student = $db->getLastObject('student');

// load first relation of the load student
$teachers = $student->relations('teacher');
$teacher = $teachers[0];

// delete relation
$student->removeRelation($teacher); // same than: $teacher->removeRelation($student);
```

Destroy all relations of an object

On several occasions it is necessary to deprive an object of all their relationships. Clear examples of this need are when a customer buys the entire stock of a product at an online store, or before dismissing a teacher it is required to delete all relationships he holds with his students. The method *removeAllRelations* accomplishes this work. The last case can be described in the code example below.

```
require_once "../ObjectDB/ObjectDB-mysql-v3.0.php"; // this will include first
require_once "teacher.php";

// load last teacher in evaluation, to dismiss
$db = new ObjectDB();
$teacher = $db->getLastObject('teacher');

// remove all relations with his older students
$teacher->removeAllRelations('student');
```

A text string that represents the name of the table with which to remove all relationships is passed as a parameter to the *RemoveAllRelations* method. This method will throw an exception *ODBEObjectNotInDatabase* if the object is not in the database or has not been saved yet. It will also throw an exception of the *ODBETableNotDefined* type if the text string inserted as a parameter does not represent the name of a table in the database.

API

This section explains the API, dividing the explanation in the four classes that compose ObjectDB. The table shown below mentions each one of those classes and shows a little explanation about the responsibilities of each one.

Class	Description
ODBException	Controls exceptions in the ObjectDB class
ODBConnection	Configures the connection and low level work
ODBObject	Wrap the information classes and provides them with ObjectDB functionality
ObjectDB	Control class. Main library functionality

ODBException

The class shown below defines the *ODBException* object; it creates a basis for working with typical exceptions of *ObjectDB*.

```
class ODBException extends Exception {
      public $message;

      function ODBException($message=null);
      public function __toString();
}
```

The exception classes listed below inherit from *ODBException* object. These define *ObjectDB* communication with the programmer, and rise in specific cases, explained in later sections. If *ObjectDB* is changed, it is a mistake to throw objects of the *ODBException* type, instead of this, developers must create their own exceptions that inherit from *ODBException*.

```
class ODBENotConnected          extends ODBException{} // Raised if are problem with database connection
class ODBEConnectionNotActive   extends ODBException{} // Raised if connection closed previously
class ODBEObjectNotInDatabase   extends ODBException{} // Raised if object to treat not exist in db
class ODBEWrongSQLSentence      extends ODBException{} // Raised if the SQL sentence are incorrect
class ODBETableNotDefined       extends ODBException{} // Raised if try to use not defined table
class ODBEClassNotDefined       extends ODBException{} // Raised if try to use not defined class
class ODBECannotChangeKey       extends ODBException{} // Raised if try to change the key for one row
class ODBEObjectNotExist        extends ODBException{} // Raised if try to use an inexistent object
class ODBEMoreThanOneInstance   extends ODBException{} // Raised if double instantiate a singletton class
class ODBERelationAlreadyExist extends ODBException{} // Raised if try to set a relation previously defined
class ODBENoRelationBetween     extends ODBException{} // Raised if try to delete an unexisting relation
```

Constructor

This class should only be constructed by others who inherit it. A text string with a personalized message, which will be shown by throwing the exception, must be passed as parameter to the constructor. If any message is not passed, throwing the exception will show the default error returned by the database.

```
Parameters
      String | none: message to display when the exception is thrown.
Exceptions
```

```
      none
```

message

The *message* attribute stores a text string that contains a custom text to be shown by throwing the exception. This attribute is public, which can be modified or queried at any time after being created the object, but for convenience it can also be initialized in the constructor of the class.

```
Data type
      String
Flags
      public
```

__toString()

This method generates a string with information about the content of the class. It can literally be called by the expression *$obj->__toString();*, but it is ready to run treating the object as a string, making it easier to call it by means of expressions such as: *echo $obj;* or *throw $obj;*.

```
Return type
      String
Parameters
      none
Exceptions
      none
```

ODBConnection

The class shown below defines the object *ODBConnection*, the same object opens and destroys connections to the database and implements the basic functions of access to it. It is not advisable to use the functions of connection directly from an object *ODBConnection*; instead it would be better the use of more sophisticated methods, provided by the *ODBObject* and *ObjectDB* objects in order to ensure the work with the cache.

An *ODBConnection* object should never be created by the user, although the possibility is not limited to. This object is instantiated and used by the other classes of *ObjectDB* to perform accesses to the database. The only contact of the programmer for this purpose should be to modify the configuration parameters.

```
class ODBConnection {
      /* start - configure */
      static public $host   = "localhost";
      static public $port   = "3306"; // mysql default port
      static public $dbname = "database_name";
      static public $user   = "database_username";
      static public $pass   = "database_password";
      /* end - configure */

      public $connect = null;

      public function ODBConnection();
      public function query($sql);
      public function getTableKey($tbName);
      public function getRelationTableName($tbNameA,$tbNameB);
      public function getLastObject($tbName);
      public function testTable($tbName);
      public function close();
      public function __destruct();
}
```

Constructor

This class should only be build by other objects of the library, but its instantiation and use is not restricted for the programmer. When it is built, opens a connection to the database, given the insertion in the configuration parameters, and closes it when it is destroyed. In case the configuration parameters are incorrect, an exception of the *ODBENotConnected* type is thrown.

```
Parameters
      none
Exceptions
      ODBENotConnected: it is thrown when there are problems with the connection
```

Destroyer

It closes the connection when the object is destroyed.

```
Exceptions
      ODBEConnectionNotActive: it is thrown if the connection was previously closed.
```

host

The attribute *host* stores a text string representing the address of the database server to connect. It can be specified by the name of a machine or an IP address.

```
Data type
      String
Flags
      public
      static
```

port

It specifies the port on which the database server listens.

```
Data type
      Integer
Flags
      public
      static
```

dbname

It creates a string representing the name of the database on the server.

```
Data type
      String
Flags
      public
      static
```

user

The *user* attribute stores a text string that represents the user name to access the database server.

```
Data type
      String
Flags
      public
      static
```

pass

It saves a string that represents the user password to access the database server.

```
Data type
      String
Flags
      public
      static
```

connect

The *connect* attribute stores the connection token with the database. This value in normal circumstances should never be used, but is maintained with public visibility to cases where the developer would want to merge *ObjectDB* with native functions of the database in which he works.

```
Data type
      Integer
Flags
      public
```

query()

It makes a query to the database, in case of a SELECT query type, it returns an array of objects *ODBObject*, otherwise returns null. Parameter is passed as a string that represents a valid query in SQL language. If an incorrect query is passed to it, it throws an exception of the *ODBEWrongSQLSentence* type.

```
Return type
      Array de ODBObject
Parameters
      String: a valid query in SQL language.
Exceptions
      ODBEWrongSQLSentence: it is thrown if the parameter is not a valid query
```

getTableKey()

It returns the key of the table to the table name passed by a parameter

```
Return type
      String: name of the field that serves as key of the table.
Parameters
      String: name of a table
Exceptions
      none
```

getRelationTableName()

It returns the name of the table that relates two tables. The names of two tables in the database are passed as parameter to it, and it returns an exception of the *ODBETableNotDefined* type in case of the relation table for them does not exist.

```
Return type
      String: name of the relation table
Parameters
      String: name of a table in the database
      String: name of another table in the database
Exceptions
      ODBETableNotDefined: it is thrown if there is no relation table in the database
```

getLastObject()

It gets the last saved object in a table. The name of the table is passed as a parameter to get the last saved object. It throws an exception of the *ODBETableNotDefined* type if the table, which was passed as parameters, does not exist.

```
Return type
      ODBObject
Parameters
      String: name of a table in the database
Exceptions
      ODBETableNotDefined: it is thrown if the table, which was passed by parameters, does not exist.
```

testTable()

It tests if a table exists in the database. The name of the table to try is passed as a parameter.

```
Return type
      Boolean: TRUE if the table exists, FALSE otherwise
Parameters
      String: name of a table in the database
Exceptions
      none
```

close()

It closes a connection to the database. It throws an exception of the *ODBEConnectionNotActive* type if the connection was closed previously.

```
Return type
      none
Parameters
      none
Exceptions
      ODBEConnectionNotActive: it is thrown if the connection was closed previously.
```

ODBObject

The class shown below defines the object *ODBObject*, it stores information of a row of the database, and provides the programmer of features that allow it to interact therewith. This class should be inherited to create and increase functionality, and new instances should not be created manually, because the library methods handle them.

```
class ODBObject {
      public function setId($id=null);
      public function setSaveStatus();
      public function setDataFromArray($data);
      public function setDataFromParamsList();
      public function getId();
      public function getState();
      public function getTableName();
      public function getTableKeyName();
      public function isSaved();
      public function save();
      public function remove();
      public function isRelationedWith($object);
      public function relations($tbName);
      public function addRelation($object);
      public function removeRelation($object);
      public function removeAllRelations($tbName);
      public function __toString();
}
```

The following constants represent states that an object can have.

```
define("OBJECT_SAVED", 0x01, true); // define an ODBObject saved into database
define("OBJECT_NOT_SAVED", 0x02, true);   // define an ODBObject not in database
define("OBJECT_DELETED", 0x03, true);     // define an ODBObject deleted with remove() function
```

setId()

It sets the key for a table row, but only if the object has not been saved.

```
Return type
      none
Parameters
      String: key of the table
Exceptions
      ODBECannotChangeKey: it is thrown if intent to change the key of a saved object occurs.
```

setSaveStatus()

It sets an object as saved; a saved object cannot be set again. This method is used for internal work of ObjectDB, and it has only been set as public to have the visibility of the *ObjectDB* class. When saving an object using the appropriate functions, the library automatically takes care of marking it, so the programmer is not advised to use it.

```
Return type
      none
Parameters
      none
Exceptions
      none
```

setDataFromArray()

It fills an object using an array of information.

```
Return type
      none
Parameters
      Array; object's values ordered according to the fields of the database.
Exceptions
      ODBEObjectNotExist: it is thrown if intention of filling a deleted or nonexistent object occurs.
```

setDataFromParamsList()

It fills an object with a parameters list.

```
Return type
      none
Parameters
      As many parameters as columns have the database, without including the key of the table.
Exceptions
      ODBEObjectNotExist: it is thrown if a deleted or nonexistent object is filled.
```

getId()

It gets the key of the table for this row.

```
Return type
      String: the table for this row
Parameters
      none
Exceptions
      none
```

getState()

It gets the state of an object.

```
Return type
      Integer: It will be the following constants (OBJECT_SAVED, OBJECT_NOT_SAVED and OBJECT_DELETED).
Parameters
      none
Exceptions
      none
```

getTableName()

It returns the name of the table which stored this object.

```
Return type
      String: name of the table
```

```
Parameters
      none
Exceptions
      none
```

getTableKeyName()

It returns the name of the field that represents the key of the table.

```
Return type
      String: name of key field in the table
Parameters
      none
Exceptions
      none
```

isSaved()

It checks if the object is in the database.

```
Return type
      Boolean: TRUE if the object is in the database, FALSE otherwise
Parameters
      none
Exceptions
      none
```

save()

If the object has just been created, it stores this object in the database, if the object was already in the database and was modified, it is updated.

```
Return type
      none
Parameters
```

```
      none
Exceptions
      ODBEObjectNotExist: it is thrown if the object has been deleted or does not exist
```

remove()

It deletes an object from the database permanently.

```
Return type
      Boolean: TRUE if the object was removed, an exception is thrown otherwise.
Parameters
      none
Exceptions
      ODBEObjectNotInDatabase: it is thrown when the object is not in the database ODBEObjectNotExist: it
is thrown when the object was removed or does not exist
```

isRelationedWith()

It checks if there is relationship between two objects.

```
Return type
      Boolean: TRUE if a relationship exists, FALSE otherwise
Parameters
      ODBObject: An object with which to test whether a relationship exists.
Exceptions
      none
```

relations()

It gets the relationships that an object has with another

```
Return type
      Array: list of objects associated with this
Parameters
      String: name of the table with which relations will be obtained
```

```
Exceptions
      ODBEObjectNotInDatabase: it is thrown whether this object does not exist in the database
      ODBETableNotDefined: Launched if the relations table does not exist
      ODBEObjectNotExist: it is thrown when the object was removed or does not exist
```

addRelation()

It defines a relationship between two objects

```
Return type
      none
Parameters
      ODBObject: the object with which to define the relationship
Exceptions
      ODBEObjectNotInDatabase: it is thrown whether this object does not exist in the database
      ODBETableNotDefined: it is thrown if the relations table does not exist
      ODBEObjectNotExist: it is thrown if the object was removed or does not exist
      ODBERelationAlreadyExist: it is thrown if the relationship was previously defined
```

removeRelation()

It deletes a relationship between two objects.

```
Return type
      none
Parameters
      ODBObject: Object with which to delete the relationship
Exceptions
      ODBEObjectNotInDatabase: it is thrown whether this object does not exist in the database
      ODBETableNotDefined: it is thrown if the relations table does not exist
      ODBEObjectNotExist: it is thrown if the object was removed or does not exist
      ODBENoRelationBetween: it is thrown in the absence of a relationship
```

removeAllRelations()

It removes all relationships that may have an object with a given table.

```
Return type
      none
Parameters
      String: name of the table with which to remove all relationships
Exceptions
      ODBEObjectNotInDatabase: it is thrown whether this object does not exist in the database
      ODBETableNotDefined: it is thrown if the relations table does not exist
      ODBEObjectNotExist: it is thrown if the object was removed or does not exist
```

__toString()

This method generates a string with information about the content of the class. It can literally be called by the expression *$obj->__toString();*, but it is ready to run treating the object as a string, making it easier to call it with expressions such as: *echo $obj;* or *throw $obj ;*.

```
Return type
      String
Parameters
      none
Exceptions
      none
```

ObjectDB

The class shown below defines the *ObjectDB* object; it executes the main functions for the library. This class cannot be instantiated more than once, or an exception of the *ODBEMoreThanOneInstance* type will be generated.

```
class ObjectDB{
      public function ObjectDB();
      public function query($sql);
      public function getObjs($tbName,$where="");
      public function saveObj($object);
      public function removeObj($object);
      public function remove($tbName,$where="");
      public function getLastObject($tbName);
}
```

Constructor

This class follows the design pattern "Singletton", so it should not be instantiated more than once. It will generate an exception of the *ODBEMoreThanOneInstance* type otherwise.

```
Parameters
      None
Exceptions
      ODBEMoreThanOneInstance: it is thrown if the intention of building more than one instance occurs
```

query()

It executes a query, whether it is a SELECT, returns a list of objects, otherwise it returns NULL

```
Return type
      Array | NULL
Parameters
      String: a valid query
```

```
Exceptions
      ODBEWrongSQLSentence: it is thrown if he SQL statement is incorrect
```

getObjs()

It gets a list of objects to filter, given the name of the table and a SQL expression. Used for obtaining the entire table, the second parameter must be blank.

```
Return type
      Array: list of objects
Parameters
      String: name of the table from which the objects will be taken.
      String: expression in SQL language to filter the results (eg: id_table = '24 ').
Exceptions
      ODBETableNotDefined: it is thrown in the case of using an undefined table
      ODBEWrongSQLSentence: it is thrown if the SQL statement is incorrect
```

saveObj()

It saves or replaces an object in the database

```
Return type
      none
Parameters
      ODBObject: not saved object for saving or replacing
Exceptions
      ODBEObjectNotExist: it is thrown if a deleted or nonexistent object is used
```

removeObj()

It deletes an object from the database

```
Return type
      Boolean: TRUE if it is deleted, otherwise it throws exceptions
Parameters
```

```
      ODBObject: object to remove
Exceptions
      ODBEObjectNotInDatabase: it is thrown whether the object to remove does not exist in the database
      ODBEObjectNotExist: it is thrown if a deleted or nonexistent object is used
```

remove()

It deletes a list of objects given the name of the table and a SQL expression to filter.

```
Return type
      none
Parameters
      String: name of the table from which objects will be deleted
      String: expression in SQL language to filter the results (eg: id_table <= '24')
Exceptions
      ODBETableNotDefined: it is thrown in the case of using an undefined table
      ODBEWrongSQLSentence: Launched when the SQL statement is incorrect
```

getLastObject()

It gets the last saved object in a table.

```
Return type
      ODBObject: last saved object
Parameters
      String: name of the table from which to obtain the latest saved object
Exceptions
      ODBETableNotDefined: it is thrown in the case of using an undefined table
```