



Step By Step Tutorial!!

Working with databases has never been easier!



TutorialEnglish

QuickDB

Updated May 25, 2010 by [diego.sarmentero](#)Labels: [Featured](#)

Step by Step Tutorial!

Working with Databases has never been Easier!

Table of Contents

1. [Using QuickDB](#)
 1. [What is QuickDB?](#)
 2. [Why QuickDB?](#)
 3. [QuickDB Capabilities](#)
 4. [Adding QuickDB to a Java Project](#)
 5. [Add Database Driver](#)
 6. [Requirements](#)
2. [Restrictions](#)
3. [Conventions](#)
 1. [For Primary Keys](#)
 2. [Inheritance](#)
 3. [Collections](#)
 4. [Getters and Setters](#)
 5. [Data Types](#)
4. [Annotations](#)
 1. [Table](#)
 2. [Parent](#)
 3. [Column](#)
 4. [Column Definition](#)
 5. [Validations](#)
 6. [Mix Mode](#)
 7. [Data Types](#)
5. [Creating Tables](#)
6. [AdminBase](#)
 1. [Creating an Instance of AdminBase](#)
 2. [Operations](#)
 1. [save](#)
 2. [saveAll](#)
 3. [saveGetIndex](#)
 4. [modify](#)
 5. [modifyAll](#)
 6. [delete](#)
 7. [obtain](#)
 8. [obtainAll](#)
 9. [obtainWhere](#)
 10. [obtainSelect](#)
 11. [obtainJoin](#)
 12. [lazyLoad](#)
 13. [executeQuery](#)
 14. [checkTableExist](#)
 15. [Transactions](#)
7. [AdminBinding](#)
 1. [Operations](#)
 1. [save](#)
 2. [saveGetIndex](#)
 3. [modify](#)
 4. [obtain](#)
 5. [obtainWhere](#)
 6. [obtainSelect](#)
 7. [lazyLoad](#)
8. [AdminThread](#)
9. [Queries](#)
 1. [StringQuery](#)
 2. [QuickDB Query](#)
 1. [Query Class](#)
 2. [Where Class](#)
 3. [Class DateQuery](#)
 4. [Performing the search](#)
10. [Views](#)
 1. [Examples](#)
 1. [Creating a View Instance](#)
 2. [Getting View Values](#)
 3. [View Dynamics Queries](#)
11. [Step by Step Example](#)
 1. [With Annotations](#)
 2. [Without Annotations](#)



Search projects

[Project Home](#) [Downloads](#) [Issues](#) [Source](#)

Search for

Updated May 25, 2010 by [diego.sarmentero](#)

Using QuickDB

1:

Usando QuickDB

1.1

What is QuickDB?

QuickDB is a library that allows a developer to focus on the definition of the entities that represent the tables of the database and perform operations that allow the interaction between these entities and the database without having to perform tedious configurations, leaving to the library the task to infer the object structure and make the mapping of the object to the Database. The Data Model to be persist doesn't need to implement any interface or extend from another Class. QuickDB uses certain Annotations or it is also possible to use the Intuitive Mode (following some Conventions) to work with the Data Model, where QuickDB determines based on the Data Type of each Object and some other features how to perform the proper mappings (It is also possible to combine attributes with and without Annotations in a single Class).

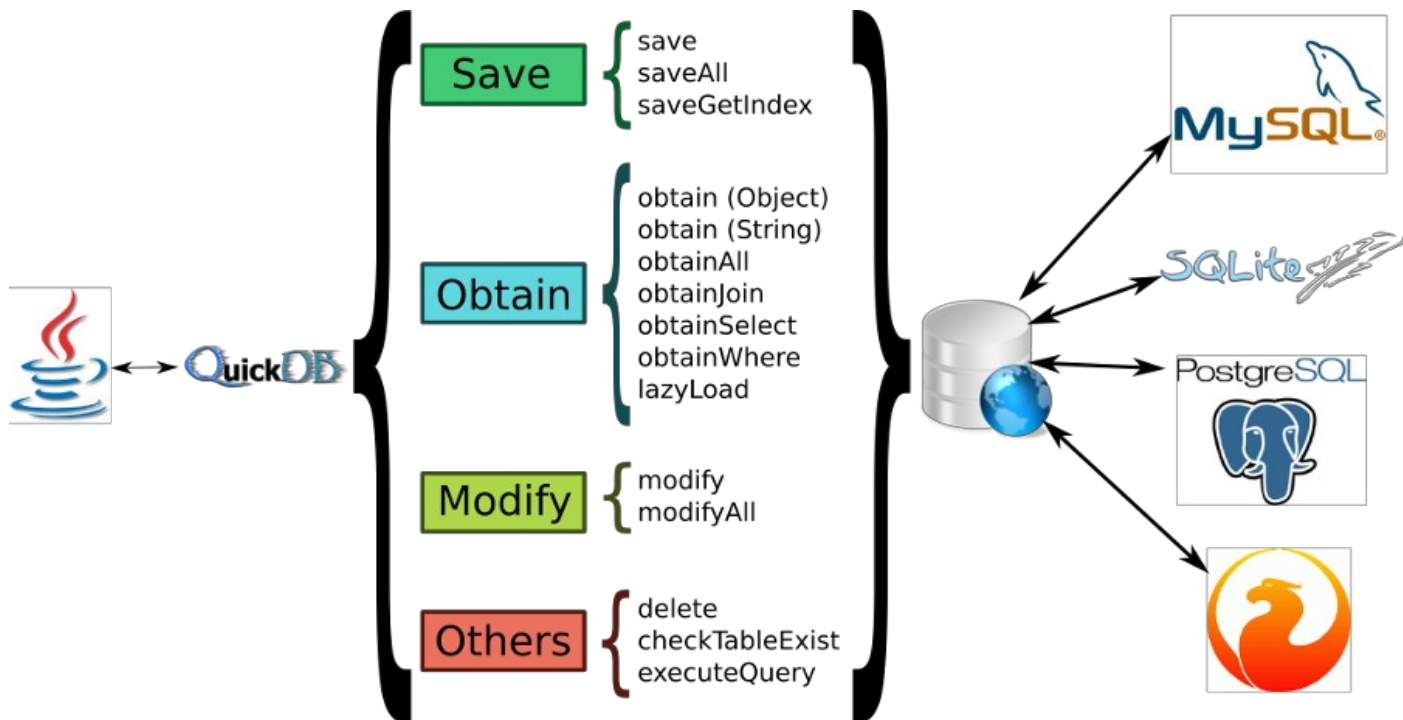
1.2

Why QuickDB?

Functionality is not synonymous with Complexity. Being able to do a lot of things don't have to involve waste a lot of time in configuration tasks.

1.3

Capacidades de QuickDB



This capabilities can be combined with Inheritance, Compound Objects, Collections (Many to Many Relation, One to Many Relation) and Automatic Table Creation.

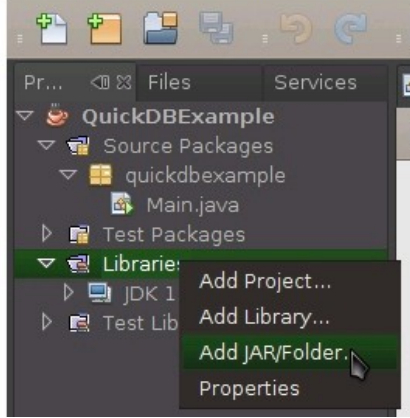
1.4

Adding QuickDB to a Java Project

To use QuickDB it is necessary to add the library into your Java Project, QuickDB use the different JDBC drivers depending on which Database is going to work with, in this way, it is not necessary for QuickDB to implement specific functionalities for each DBMS, but that should only be able to establish the connection with the Database through one of this drivers in runtime. This present the advantage to avoid the need of having in the project functionalities from another DBMS that is not going to be used.

To add QuickDB into a Project there are several ways, the most common are:

- Add the library into the Project using the IDE (Netbeans example):



- Add the library into the Project as a Maven Dependency:

```
<dependency>
  <groupId>cat.quickdb</groupId>
  <artifactId>QuickDB</artifactId>
  <version>1.2</version>
</dependency>
```

To be able to add QuickDB dependency into a Maven Project, it is necessary that the library can be reachable from the local repository or from an external repository, Para poder agregar la dependencia con QuickDB desde Maven es necesario que la librería este accesible en nuestro repositorio local o en un repositorio externo, what can be done in two different ways:

1. Download the library and then install it in the Local Repository:

```
mvn install:install-file -Dfile=path-to-QuickDB-file-jar \
  -DgroupId=cat.quickdb \
  -DartifactId=QuickDB \
  -Dversion=1.2 \
  -Dpackaging=jar \
  -DcreateChecksum=true
```

1. Add QuickDB Repository into your Maven Project:

```
<repositories>
  <repository>
    <id>QuickDB</id>
    <url>http://quickdb.googlecode.com/hg/mavenRepo</url>
  </repository>
</repositories>
```

1.5

Add Database Driver

Once you have a Project that include QuickDB, it is necessary to add the driver that will establish the connection between QuickDB and the Database.

QuickDB currently works with the following Databases:

- **MySQL:** <http://dev.mysql.com/downloads/connector/j/>
- **PostgreSQL:** <http://jdbc.postgresql.org/download.html>
- **SQLite:** <http://www.zentus.com/sqlitejdbc/>
- **Firebird:** <http://www.firebirdsql.org/index.php?op=files&id=jaybird>

The process to add any of this drivers is the same as described to include QuickDB.

1.6

Requirements

- **QuickDB works with Java version 1.5 or higher.**

The versions of the drivers with those who have carried out the tests are:

- MySQL: Versión 5.1.5 or Higher
- PostgreSQL: JDBC4 Versión 8.4-701 (or Compatible)
- SQLite: Versión 056 (or Compatible)

► [Sign in](#) to add a comment

©2010 Google - [Terms](#) - [Privacy](#) - [Project Hosting Help](#)

Powered by [Google Project Hosting](#)



Search projects

[Project Home](#)[Downloads](#)[Issues](#)[Source](#)

Search

for

Search

Restrictions

Updated May 26, 2010 by [diego.sarmentero](#)**2:**

Restrictions

To work with QuickDB, there are certain restrictions that must be borne in mind:

- All model classes should contain an integer primary key attribute, and should be the first attribute declared in the class.
- All model classes should contain an empty constructor.
- In case of a Parent-Son relation, if "Get" and "Set" methods of the primary key have the same name, then both Parent and Son must have the same primary key value (since they are public methods, parent's value would be overwritten by the son, preventing another class to be stored in the database extending from the same parent. Is solved using Annotations).
- To combine the use of attributes in a class with and without annotations (to set the table creation, ie when using the annotation `@ColumnDefinition`), you need to add the annotation `@Table` to Class.

► [Sign in](#) to add a comment

©2010 Google - [Terms](#) - [Privacy](#) - [Project Hosting Help](#)

Powered by [Google Project Hosting](#)



Conventions

Updated May 10, 2010 by [diego.sarmentero](#)

3:

Conventions

Conventions are used to remove the configuration tasks within QuickDB. There are many properties that can be deducted automatically without any configuration (although, if desired, anything can be configured as will be seen in the next section Annotations). Specified below are details to take into account in the Data Model, to be supported by QuickDB without the need for more than writing Entities. These conventions are necessary when not working with annotations, otherwise, the Entities can be written in any other way (provided that such characteristics are specified by the annotations).

3.1

For Primary Keys

Each class in the model, has to declare, as a first attribute, an integer variable named "id":

```
public class Person{
    private int id;
    ...
}
```

3.2

Inheritance

To QuickDB being able to recognize automatically the inheritance, both Parent and Son, must be in the same package. This makes possible to determine when to stop analyzing the inheritance recursively, and not reach the level of Object.

```
package com.example;

public class Parent{
    ...
}

package com.example;

public class Son extends Parent{
    ...
}
```

3.3

Collections

QuickDB supports only those collections that implements "java.util.List" or "java.util.Collection" interfaces, and when you work without annotation, the attribute name must match the name of the class of objects that make up this collection (with the first letter in lowercase if desired).

```
public class Person{
    private ArrayList phone;
}

public class Phone{
    ...
}
```

For collections that contain only primitive data types, see into "Annotation" section where it's explained how contemplate this case.

3.4

Getters y Setters

When working without annotations, QuickDB should automatically determine which methods will set and return the value of each attribute. To do so, we follow the convention by default of write "set" or "get" and then the name of the attribute with CamelCase:

```
public class Person{

    private int id;
    private String name;
```

```

public void setId(int id){...}
public int getId(){...}
public void setName(String name){...}
public String getName(){...}
}

```

3.5

Data Types

| Java | MySQL | PostgreSQL | Firebird | SQLite |
|-------------------|------------|-------------------|------------------|----------------|
| java.lang.Integer | INTEGER | integer | integer | Dynamic Typing |
| java.lang.Double | DOUBLE | double precision | double precision | Dynamic Typing |
| java.lang.Float | FLOAT | real | float | Dynamic Typing |
| java.lang.String | VARCHAR | character varying | varchar | Dynamic Typing |
| java.lang.Boolean | TINYINT(1) | boolean | boolean | Dynamic Typing |
| java.lang.Long | BIGINT | bigint | bigint | Dynamic Typing |
| java.lang.Short | SMALLINT | smallint | smallint | Dynamic Typing |
| java.sql.Date | DATE | date | date | Dynamic Typing |

► [Sign in](#) to add a comment

©2010 Google - [Terms](#) - [Privacy](#) - [Project Hosting Help](#)

Powered by [Google Project Hosting](#)



Annotations

Updated May 22, 2010 by [diego.sarmentero](#)

4:

Annotations

4.1

Table

Table may be accompanied by a parameter or not, the Table parameter indicates to which table in the database is mapped this entity. By not including the parameter is simply assumed that the table has the same name as the class.

- Directly specifying to which table is a Class referred:

```
import cat.quickdb.annotation.Table;

@Table("TableExample")
public class Example{
    ...
}
```

- Taking the name of the Class as Table's name.:

```
import cat.quickdb.annotation.Table;

@Table
public class Example{
    ...
}
```

4.2

Parent

Parent indicates that this class inherits from another, and therefore the attributes of the parent class will be stored in another table (this notation does not include any attribute).

```
import cat.quickdb.annotation.Parent;
import cat.quickdb.annotation.Table;

@Parent
@Table
public class ExampleSon{
    ...
}
```

Using this notation, can be used a parent class that is not within the same package, which was a restriction on the work simply with the [Conventions](#).

4.3

Column

Column annotations are used to define completely an attribute of the entity with its corresponding mapping in a table in the database. The various parameters of the Column annotation, have all default values, so you only need to fill those, whom are necessary for a particular case.

- **name:** Field Name in the Table of the database, if not specified bears the name of the attribute.
- **type:** indicates the data type of the attribute, which can be PRIMARYKEY, FOREIGNKEY, COLLECTION, PRIMITIVE (if not specified it is assumed that the type is PRIMITIVE, which is one of the language primitives, also contemplating String and java.sql.Date).
- **getter:** This parameter give the possibility to explicitly express, which will be the method who will return the value of that attribute (if not specified it is assumed that, if the attribute is named "attribute", getter method will be "getAttribute").
- **setter:** The same as above but for setting methods.
- **collectionClass:** When the attribute marked with this annotation is a collection, can be said explicitly what type of objects will contain this collection.
- **ignore:** By setting as True, QuickDB will not consider this attribute in the persistence of the object.

All unspecified parameters, are set to default following the behavior of the [Conventions](#).

If a class has a "name" attribute that represents a field with the same name in their respective table in the database, is type String (for example), and their respective get and set methods are "getName" and "setName"; then for Column annotation would be enough:

```
@Column
private String name;
```

Which would be the equivalent as leave the attribute without annotation, because it follows the [Conventions](#). If, however, the attribute mentioned above doesn't follow any of the conditions, could be indicate which field (in the database) will reference it, its type, and its get and set methods:

```
import cat.quickdb.annotation.Column;
import cat.quickdb.annotation.Properties.TYPES;
import cat.quickdb.annotation.Table;

@Table("TableExample")
public class Example{

    @Column(name="userName", type=TYPES.PRIMITIVE, getter="readName", setter="writeName")
    private String name;
}
```

In the case of parameter "collectionClass" of column, there are two ways to specify:

- By placing only the name of the class whose objects form part of the collection, if it is within the same package that contains the Class who has the Collection attribute:

```
@Column(collectionClass="ExampleSon")
private ArrayList examples;
```

- Or by specifying the full path (Package.Class) in case of it is in another package:

```
@Column(collectionClass="quickdb.example.ExampleSon")
private ArrayList examples;
```

For Collections of Primitive Data types (int, double, float, short, long, boolean, and String and Date too), annotations should be used and are specified as follows:

```
@Column(collectionClass="java.lang.Boolean")
private ArrayList booleans;

@Column(collectionClass="java.sql.Date")
private ArrayList dates;

@Column(collectionClass="java.lang.Double")
private ArrayList doubles;

@Column(collectionClass="java.lang.Float")
private ArrayList floats;

@Column(collectionClass="java.lang.Integer")
private ArrayList integers;

@Column(collectionClass="java.lang.String")
private ArrayList strings;
```

It's important to note that it is only necessary to specify those parameters that are necessary, if we wish to indicate only the type, we could write:

```
@Column(type=TYPES.PRIMARYKEY)
private int id;
```

The different **Types** that can be specified are:

- **PRIMITIVE:** It refers to those types that are mapped directly to the data types supported by the database.
- **PRIMARYKEY:** It refers to the primary key of the Table and it's important to specify it, as it indicates to the library that, in cases of insertions, not try to force the value of the field, and let the database handle it.
- **FOREIGNKEY:** It refers to foreign keys, which are represented in the database with references to other tables, and in the entities with references to other objects, this way, when you have a Class with a reference to another object, wich should also be mapped in a table, it must be marked as type "FOREINGKEY".
- **COLLECTION:** Specifies whether the attribute is library Collection, to handle it as a one-to-many or many-to-many relation. (creates the necessary Tables to carry out this function).

4.4

Column Definition

Column Definition sets the properties to the creation of the Table in the database, when try to make the first insertion if the table does not exist. The parameters passed are:

- **type:** The data type with which will be create their respective column in the database. Default is VARCHAR.
- **length:** Length of the Data Type. By default is no length specified to the types, except for VARCHAR who has assigned a predetermined length of

150.

- **NotNull**: Specifies whether the column can accept or not null values. By default is true.
- **defaultValue**: The value by default of each field in the column. By default is an empty string.
- **autoIncrement**: By default is false.
- **unique**: Specifies whether the field's value type is unique. By default is false.
- **primary**: Specifies whether the column is primary key type. By default is false.
- **format**: Specifies the column format. By default is DEFAULT. (Only for MySQL)
- **storage**: Sets the Storage type to use. By default is DEFAULT. (Only for MySQL)

Example:

```
import cat.quickdb.annotation.Column;
import cat.quickdb.annotation.ColumnDefinition;
import cat.quickdb.annotation.Definition;
import cat.quickdb.annotation.Properties.TYPES;
import cat.quickdb.annotation.Table;

@Table
public class Person{

    @Column(type=TYPES.PRIMARYKEY)
    @ColumnDefinition(type=Definition.DATATYPE.INT, length=11, autoIncrement=true, primary=true)
    private int id;

    @Column
    @ColumnDefinition
    private String name;
}
```

4.5

Validations

Validations allows automatic checks on the entity to determine if the object will be stored or modified in the database. If the Validation condition is not fulfilled the upgrade process would be suspended.

A validation can be carried out through the following conditions:

- **conditionMatch**: Receives an array of strings with different regular expressions to evaluate.
- **maxLength**: Specifies the maximum length that can have a field (including the value specified).
- **numeric**: Makes numeric type checks on the field. Receive an array with a set of numerical values, being organized in this way: value.
- **date**: Makes checks on an object of type date. Receives an array with a set of numerical values, being organized in this way: (part_of_date, condition, value).

As regular expressions for "conditionMatch", Validation annotation has four armed expressions which you can choose, rather than create new ones, these are:

- **conditionURL**: Evaluates the text entered in the field, as a well-formed URL expression.
- **conditionMail**: Evaluates the text entered in the field, as a well-formed e-mail address.
- **conditionSecurePassword**: Evaluates the text entered in the field, as secure password (taking into account capital letters, lowercase letters, numbers and special characters).
- **conditionNoEmpty**: Evaluates that the variable does not contain an empty string.

For "numeric" checks, conditions which can be chosen are:

- EQUAL
- LOWER
- GREATER
- EQUALORLOWER
- EQUALORGREATER

For "date" checks, parts of the date available are:

- YEAR
- MONTH
- DAY

And the conditions are the same as those of "numeric".

1. Complete example using all available validations.

```
import cat.quickdb.annotation.Validation;
import java.sql.Date;

public class ValidUser{

    private int id;
    @Validation(maxLength=20, conditionMatch={Validation.conditionNoEmpty})
    private String name;
    @Validation(conditionMatch={Validation.conditionSecurePassword})
    private String pass;
    @Validation(conditionMatch={Validation.conditionMail})
    private String mail;
    @Validation(conditionMatch={Validation.conditionURL})
    private String url;
}
```

```

    @Validation(date={Validation.YEAR, Validation.EQUALORGREATER, 1988})
    private Date birthDate;
    @Validation(numeric={Validation.GREATER, 18})
    private int age;
}

```

1. Example of numerical validations.

```

import cat.quickdb.annotation.Validation;

public class ValidComplexNumeric{

    private int id;
    @Validation(numeric={Validation.EQUAL, 5})
    private String number1;
    @Validation(numeric={Validation.EQUALORGREATER, 2, Validation.EQUALORLOWER, 9})
    private int number2;
    @Validation(numeric={Validation.GREATER, 1, Validation.LOWER, 3})
    private int number3;
}

```

1. Example of date validations.

```

import cat.quickdb.annotation.Validation;
import java.sql.Date;

public class ValidComplexDate{

    private int id;
    @Validation(date={Validation.DAY, Validation.GREATER, 5,
        Validation.MONTH, Validation.EQUAL, 5,
        Validation.YEAR, Validation.EQUALORLOWER, 2009})
    private Date date;
}

```

4.6

Mixed Mode

It's possible to work with Classes that specifies some attributes with Annotations, and others without (the latter being those who follow [Conventions](#)). You only have to consider placing the @Table annotation to the level of Class (As mentioned in section [Restrictions](#)).

4.7

Data Types

Data Types that can be defined with the annotation ColumnDefinition are:

| Java | MySQL | PostgreSQL | Firebird | SQLite |
|-----------|------------|------------------|------------------|----------------|
| BIT | BIT | bit | bit | Dynamic Typing |
| BOOLEAN | TINYINT(1) | boolean | boolean | Dynamic Typing |
| SMALLINT | SMALLINT | smallint | smallint | Dynamic Typing |
| INT | INT | integer | int | Dynamic Typing |
| INTEGER | INTEGER | integer | integer | Dynamic Typing |
| BIGINT | BIGINT | bigint | bigint | Dynamic Typing |
| REAL | REAL | real | float | Dynamic Typing |
| DOUBLE | DOUBLE | double precision | double precision | Dynamic Typing |
| FLOAT | FLOAT | real | real | Dynamic Typing |
| DECIMAL | DECIMAL | decimal | decimal | Dynamic Typing |
| NUMERIC | NUMERIC | numeric | numeric | Dynamic Typing |
| DATE | DATE | date | date | Dynamic Typing |
| TIME | TIME | time | time | Dynamic Typing |
| TIMESTAMP | TIMESTAMP | timestamp | timestamp | Dynamic Typing |
| DATETIME | DATETIME | timestamp | timestamp | Dynamic Typing |

| | | | | |
|-----------|-----------|-------------------|-----------|----------------|
| DATETIME | DATETIME | timestamp | timestamp | Dynamic Typing |
| CHAR | CHAR | character | char | Dynamic Typing |
| VARCHAR | VARCHAR | character varying | varchar | Dynamic Typing |
| BINARY | BINARY | bytea | --- | Dynamic Typing |
| VARBINARY | VARBINARY | bit varying | --- | Dynamic Typing |
| TEXT | TEXT | text | nchar | Dynamic Typing |

► [Sign in](#) to add a comment

©2010 Google - [Terms](#) - [Privacy](#) - [Project Hosting Help](#)

Powered by [Google Project Hosting](#)



CreatingTable

Updated May 18, 2010 by santi...@gmail.com

5:

Creating Tables

QuickDB infers, through the data type, how to create the corresponding table in the database. For Creating the tables are taken into account the concepts explained in the section [Conventions](#) and [Annotations](#). In absence of annotations, establishing the Table taking the name of the class as the table's name, using the attribute (required) integer "id" as auto incremental primary key and creating the other columns according the data types specified in the [Conventions](#), all with NOT NULL property. In case of Strings, the field is created with a length equal to that of the entered String or equal to 150 if the first input is smaller than this size. In case of Inheritance, is added a field in the table with the name "parent_id" which will contain the primary key value of the Parent table. For collections, field is ignored when mapping the class to a table, and a new relational table between the Class and the elements contained in the collection, is created. In case of working with annotations, all these properties mentioned and several more, can be completely configured using the annotation [@!ColumnDefinition](#) previously explained.

Following is a complete example using all annotations described in the previous section and the resulting tables after the first insertion:

```
import cat.quickdb.annotation.Table;

@Table("ModelParentTest")
public class ModelParent{
    private int id;
    private String description
}

import cat.quickdb.annotation.Table;
import cat.quickdb.annotation.Column;
import cat.quickdb.annotation.ColumnDefinition;
import cat.quickdb.annotation.Definition.DATATYPE;
import cat.quickdb.annotation.Properties.TYPES;

@Table("CollecAnnotation")
public class CollectionAnnotation{

    @Column(type=TYPES.PRIMARYKEY)
    @ColumnDefinition(autoIncrement=true, length=11,
        primary=true, type=DATATYPE.INT)
    private int idCollection;
    @Column(name="itemName", setter="setItemName", getter="getItemName")
    private String item;
    @Column(ignore=true)
    private String nothing;
}

import cat.quickdb.annotation.Table;
import cat.quickdb.annotation.Column;
import cat.quickdb.annotation.Parent;
import cat.quickdb.annotation.ColumnDefinition;
import cat.quickdb.annotation.Definition.DATATYPE;
import cat.quickdb.annotation.Properties.TYPES;
import cat.quickdb.annotation.Validation;
import java.sql.Date;
import java.util.ArrayList;

@Table("AnnotationModel")
public class ModelAnnotation extends ModelParent{

    @Column(type=TYPES.PRIMARYKEY)
    @ColumnDefinition(autoIncrement=true, length=11,
        primary=true, type=DATATYPE.INT)
    private int idCollection;
    @Column(name="modelName")
    @ColumnDefinition(length=300, defaultValue="test")
    private String name;
    @Validation(numeric={Validation.GREATER, 18})
    @ColumnDefinition(type=DATATYPE.INTEGER)
    private int age;
    @ColumnDefinition(type=DATATYPE.DATETIME)
    private Date birth;
    @Column(getter="getterSalary")
    private double salary;
    @Column(type=TYPES.COLLECTION, collectionClass="cat.quickdb.annotations.model.CollectionAnnotation")
    private ArrayList array;
    @Column(type=TYPES.FOREIGNKEY, name="foreignCollec")
    @ColumnDefinition(type=DATATYPE.INTEGER)
    private CollectionAnnotation collec;
```

}

Table: "ModelParentTest" (Class: "ModelParent")

| Column Name | Data Type | NOT NULL | AUTO INC | Flags | Default Value |
|-------------|--------------|-------------------------------------|-------------------------------------|-------|---------------|
| id | INTEGER | <input checked="" type="checkbox"/> | <input checked="" type="checkbox"/> | | NULL |
| description | VARCHAR(150) | <input checked="" type="checkbox"/> | <input type="checkbox"/> | | NULL |

Table: "CollecAnnotation" (Class: "CollectionAnnotation")

| Column Name | Data Type | NOT NULL | AUTO INC | Flags | Default Value |
|--------------|--------------|-------------------------------------|-------------------------------------|-------|---------------|
| idCollection | INTEGER | <input checked="" type="checkbox"/> | <input checked="" type="checkbox"/> | | NULL |
| itemName | VARCHAR(150) | <input checked="" type="checkbox"/> | <input type="checkbox"/> | | NULL |

Table: "AnnotationModelCollecAnnotation" (relationship between the class that contains the collection and the items in the collection)

| Column Name | Data Type | NOT NULL | AUTO INC | Flags | Default Value |
|-------------|-----------|-------------------------------------|-------------------------------------|-------|---------------|
| id | INTEGER | <input checked="" type="checkbox"/> | <input checked="" type="checkbox"/> | | NULL |
| base | INTEGER | <input checked="" type="checkbox"/> | <input type="checkbox"/> | | NULL |
| related | INTEGER | <input checked="" type="checkbox"/> | <input type="checkbox"/> | | NULL |

Table: "AnnotationModel" (Class: "ModelAnnotation")

| Column Name | Data Type | NOT NULL | AUTO INC | Flags | Default Value |
|---------------|--------------|-------------------------------------|-------------------------------------|-------|---------------|
| idModel | INTEGER | <input checked="" type="checkbox"/> | <input checked="" type="checkbox"/> | | NULL |
| salary | DOUBLE | <input checked="" type="checkbox"/> | <input type="checkbox"/> | | NULL |
| modelName | VARCHAR(300) | <input checked="" type="checkbox"/> | <input type="checkbox"/> | | 'test' |
| age | INTEGER | <input checked="" type="checkbox"/> | <input type="checkbox"/> | | NULL |
| birth | DATETIME | <input checked="" type="checkbox"/> | <input type="checkbox"/> | | NULL |
| foreignCollec | INTEGER | <input checked="" type="checkbox"/> | <input type="checkbox"/> | | NULL |
| parent_id | INTEGER | <input checked="" type="checkbox"/> | <input type="checkbox"/> | | NULL |

It should be noted that many of the annotations used, could have been overlooked following the conventions and the same results would be obtained in the definition of data types.

▶ [Sign in](#) to add a comment



AdminBaseEn

Updated May 26, 2010 by [diego.sarmentero](#)

6:

AdminBase

AdminBase is the main class of the Library and with which all the operations related to the database are perform. An instance of AdminBase should be created to make use of the interaction between the Database and the Data Model.

6.1

By creating the instance, [AdminBase](#) must received certain parameters with which will attempt to create the connection with the Database:

AdminBase Parameters are:

- Host
- Port
- Database Name
- User
- Password
- Schema (Solo para Postgre)

Examples:

- For MySQL:

```
AdminBase admin = AdminBase.initialize(AdminBase.DATABASE.MYSQL,
    "localhost", "3306", "testQuickDB", "root", "pass");
```

- For Postgre:

```
AdminBase admin = AdminBase.initialize(AdminBase.DATABASE.POSTGRES,
    "localhost", "5432", "testQuickDB", "postgres", "postgres", "testing");
```

- For Firebird:

```
AdminBase admin = AdminBase.initialize(AdminBase.DATABASE.FIREBIRD,
    "localhost", "employees.gdb", "SYSDBA", "firebird");
```

- For SQLite:

```
AdminBase admin = AdminBase.initialize(
    AdminBase.DATABASE.SQLite, "test");
```

6.2

Operations

The operations represents the actions that can be performed between the Object Data Model and the Database. To explain each of the operations is taken as an example the following class:

```
public class Person{
    private int id;
    private String name;
    private int age;

    public Person(){
    }
    public Person(String name, int age){
        this.name = name;
        this.age = age;
    }
}
```

```
//Getters
//Setters
}
```

6.2.1

save

To save an object in the database, it is necessary to create an instance of it and then invoke the method "save" which is the responsible to execute that task.

Example:

```
Person person = new Person("leeloo", 22);
admin.save(person);
```

6.2.2

saveAll

To save a collection of objects in the database, it is necessary to create the instantiate the collection including the objects from the Data Model and then invoke the method "saveAll" which is the responsible to execute that task.

Example:

```
ArrayList array = new ArrayList();
array.add(add new Person("name1", 20));
array.add(add new Person("name2", 20));
array.add(add new Person("name3", 32));

admin.saveAll(array);
```

6.2.3

saveGetIndex

To save an object in the database and retrieve the Key generated for that Object, it is necessary to create an instance of it and then invoke the method "saveGetIndex" which is the responsible to execute that task.

Example:

```
Person person = new Person("leeloo", 22);
admin.saveGetIndex(person);
```

6.2.4

modify

To modify an object in the database, it is necessary to retrieve the Object first from the Database, apply the proper modifications and then invoke the method "modify" which is the responsible to execute that task.

Example:

```
Person p = new Person();
admin.obtain(p, "age=22");

p.setName("Leonardo");
admin.modify(p);
```

6.2.5

modifyAll

It is the same case as "modify", you get first the collection desired, modify the values of that collection and execute "modifyAll."

Example:

```
Person p = new Person();
arrayList array = admin.obtainAll(p, "age=22");

for(Object o : array){
    //Modify the Object
}

admin.modifyAll(array);
```

6.2.6

delete

Retrieve the Object to be erase and then execute "delete".

Example:

```
Person p = new Person();
admin.obtain(p, "age=22");

admin.delete(p);
```

6.2.7

obtain

The "obtain" method is overloaded, because is possible to work with any of the two QuickDB query systems (which are explained in another section.) Therefore "Obtain" can receive as parameter the object from which the data is desired, this being the QuickDB system called "Query":

```
Person p = new Person();
admin.obtain(p).where("street", Address.class).equal("unnamed street").find();
```

Or "obtain" can receive two parameters, the first one is the object from which the data is desired, and the second a String specifying the search conditions based on the characteristics of the second QuickDB Query System called "StringQuery":

```
Person p = new Person();
admin.obtain(p, "address.street = 'unnamed street'");
```

6.2.8

obtainAll

To obtain from the Database a collection of Objects, is only necessary to create an empty instance of the particular object and then execute the method "obtainAll" from AdminBase to retrieve all the objects that satisfied some condition.

Example:

```
Person person = new Person();
ArrayList array = admin.obtainAll(person, "age=22");
```

Or:

```
ArrayList array = admin.obtainAll(Person.class, "age=22");
```

It is also possible to perform the search using the QuickDB Query System as will be mention in the Query Section.

6.2.9

obtainWhere

This method allows the developer to specify only the WHERE section in SQL query and will return the resulting object. Is faster than QuickDB Query System because not need to infer the structure of the object to perform the JOINS relevant to carry out the query when other objects are involve, but this only allow simple queries on the same object.

Example:

```
Person p = new Person();
admin.obtainWhere(p, "age=22");
```

6.2.10

obtainSelect

This method returns the resulting object of execute the SQL query specified explicitly. The specified query must refer to the object that is passed by parameter and must be fully defined.

Example:

```
Person p = new Person();
admin.obtainSelect(p, "SELECT * FROM Person WHERE age=22");
```

6.2.11

obtainJoin

Through this method it is possible to obtain an Array (Object), where each element of this array will turn to be a string, with the number of elements equal to the columns specified in the search (ie, this method returns a representation of the resulting table as an object array). It is useful to complete the data of a graphic component as a Table, etc.

You must specify the complete SQL query:

```
Object[] objects = admin.obtainJoin("SELECT Person.name, Person.age FROM Person", 2);
```

6.2.12

lazyLoad

By this method it is possible to load an object gradually as needed.

For example, for the following classes:

```
public class Person{
    private int id;
    private String name;
    private Phone phone;
    //Getters - Setters
}

public class Phone{
    private int id;
    private String number;
    private Company company;
    //Getters - Setters
}

public class Company{
    private int id;
    private String description;
    //Getters - Setters
}
```

Only the attributes of Person are loaded (1), then the values of Phone's attributes are added (2) and finally those of Company (3):

```
Person p = new Person();

//Person Attributes(1)
admin.lazyLoad(p, "age=22");
//Phone is NULL at this moment

//Attributes from Phone are added
admin.lazyLoad(p);
//Phone is already complete, except for Company which is NULL

//Attributes from Company are added
admin.lazyLoad(p);
```

6.2.13

executeQuery

"executeQuery" is perhaps the simplest method, since all it does is execute a statement in the database and the only thing returned is True if ends successfully or False otherwise.

Example:

```
admin.executeQuery("DELETE FROM Person");
```

6.2.14

checkTableExist

Determines if a given table exist (returns True), or otherwise if that Table does not exist (returns False).

Example:

```
if(admin.checkTableExist("Person")){
    System.out.println("The Table Exists");
}else{
    System.out.println("The Table Does Not Exists");
}
```

6.2.15

Transacciones

By default QuickDB manages transactions at the level of the object which begins the operation, it means, no matter if it is a simple object, composed with another objects, with collections, or with inheritance, must complete the operation (along with all the involving objects) successfully, otherwise it will discard all changes that were made during the operation. But it is also possible to establish that all operations are implemented independently, no matter what happens to the related objects, as follows:

```
admin.setAutoCommit(true);
```

It is also possible to manage the transactions explicitly, for the cases where it is necessary to save a group of object or neither.

```
Example example = new Example();
Person person = new Person("leeloo", 22);

admin.openAtomicBlock();
admin.save(example);
admin.save(person);
admin.closeAtomicBlock();
```

If we wanted to validate that all the operations were carried out with success or otherwise cancel the transaction and make a rollback, it could be done as follows:

```
Example example = new Example();
Person person = new Person("leeloo", 22);

boolean value = true;
admin.openAtomicBlock();
value &= admin.save(example);
value &= admin.save(person);
if(value){
    admin.closeAtomicBlock();
}else{
    admin.cancelAtomicBlock();
}
```

► [Sign in](#) to add a comment

©2010 Google - [Terms](#) - [Privacy](#) - [Project Hosting Help](#)

Powered by [Google Project Hosting](#)



AdminBindingEn

Updated May 10, 2010 by [diego.sarmentero](#)

7:

AdminBinding

AdminBinding allows the developer to create the data model extending the AdminBinding Class (but this is not required, this is another available resource to simplify some processes), which serves as a "link" between the entities and AdminBase to allow the operations that interact with the database to be executed from the object itself and not use AdminBase intermediary as directly. AdminBinding not cover all the functionality that AdminBase provides, as it only handles those operations that are specific to the object that contains them, leaving out those that refer to collections, etc.

In this way, the interaction could be even more natural, just telling the object which operation to execute. It is necessary before starting to work with the entities of the model to execute the following line of code, which initializes the connection with the database for all the instances of the model:

```
AdminBase.initializeAdminBinding(AdminBase.DATABASE.MYSQL,
    "[HOST]", "[PORT]", "[DATABASE]", "[USER]", "[PASSWORD]");
```

Where the properties are the same as for the initialization of AdminBase:

- The DBMS to be used.
- Host
- Port
- Database
- User
- Password
- Schema (Only for Postgre)

The Methods inherited from AdminBinding Class are:

- save
- saveGetIndex
- delete
- modify
- obtain
- obtainSelect
- obtainWhere
- lazyLoad

Methods as "obtainAll", "saveAll", etc. Have been left out because they do not specifically refer to the object that contains them, but refer to collections or other operations involving external objects.

Examples are based on the following entity:

```
import cat.quickdb.db.AdminBinding;

public class Person extends AdminBinding{

    private int id;
    private String name;
    private int age;

    public Person(){
    }
    public Person(String name, int age){
        this.name = name;
        this.age = age;
    }

    //Getters
    //Setters
}
```

7.1

Operations

The operations are specified in the same way as is done in "AdminBase" with the difference that the first attribute is omitted, which one was the object that involved the operation, and now this object is the one that execute the action by itself.

Example:

7.1.1

save

```
Person person = new Person("diego", 23);
person.save();
```

7.1.2

saveGetIndex

```
Person person = new Person("diego", 23);
int id = person.saveGetIndex();
```

7.1.3

modify

```
Person p = new Person();
p.obtain("age=23");

p.setName("Leonardo");
p.modify();
```

7.1.4

obtain

(Both of QuickDB Query Systems are supported)

```
Person p = new Person();
p.obtain("name = 'diego'");

p.obtain().where("street", Address.class).equal("unnamed street").find();
```

7.1.5

obtainSelect

```
Person p = new Person();
p.obtainSelect("SELECT * FROM Person WHERE age=23");
```

7.1.6

obtainWhere

```
Person p = new Person();
p.obtainWhere("age=23");
```

7.1.7

lazyLoad

```
public class Person extends AdminBinding{
    private int id;
    private String name;
    private Phone phone;
    //Getters - Setters
}

public class Phone extends AdminBinding{
    private int id;
    private String number;
    private Company company;
    //Getters - Setters
}

public class Company extends AdminBinding{
    private int id;
    private String description;
    //Getters - Setters
}
```

Operation is executed as follows:

```
Person p = new Person();
//Person Attributes(1)
```

```
p.lazyLoad("age=23");  
//Phone is NULL at this moment  
  
//Attributes from Phone are added  
p.lazyLoad("");  
//Phone is already complete, except for Company which is NULL  
  
//Attributes from Company are added  
p.lazyLoad("");
```

► [Sign in](#) to add a comment

©2010 Google - [Terms](#) - [Privacy](#) - [Project Hosting Help](#)

Powered by [Google Project Hosting](#)



Search projects

[Project Home](#)[Downloads](#)[Issues](#)[Source](#)

Search

Current pages

for

Search

AdminThreadEn

Updated May 10, 2010 by [diego.sarmentero](#)**8:**

AdminThread

AdminThread exposes virtually the same functionality as AdminBase, with the difference that all the transactions executed are created in a different execution thread. This feature allows to execute operations involving objects or very large collections of objects without blocking the application. One detail to note is that the operations are executed in a separate thread, it is not known at what time is going to be executed, and therefore none of the methods return a value.

There are two ways to initialize an AdminThread object:

```
AdminThread adminThread1 = new AdminThread(admin);

AdminThread adminThread1 = new AdminThread(DATABASE.MYSQL,
    "localhost", "3306", "testQuickDB", "root", "pass");
```

The first one consist to pass to the constructor an instance of [AdminBase](#) and which will be uses for the different operations, and the second one is to pass the same data that is passed to the method of initialization from AdminBase to allow AdminThread to create his own instance of [AdminBase](#).

AdminThread support the following operations:

- **save**
- **modify**
- **delete**
- **execute**
- **lazyLoad**
- **obtainWhere**
- **obtainSelect**
- **obtain (Object)**
- **obtain (String)**
- **saveAll**
- **modifyAll**
- **setAutoCommit**

Operations as "saveGetIndex" have been left out, since the purpose of the operation is to obtain the return value, and none of the tasks set by AdminThread has a return value. It is also important to note that although operations as "obtain..." are included, these do not ensure that the object is loaded at that time, and we do not know when the thread is going to be executed.

► [Sign in](#) to add a comment



Queries

Updated May 22, 2010 by santi...@gmail.com

9:

Queries

QuickDB implements two systems of Queries, which makes much easier to obtain an specific object, since he promotes the query creation and the specification of the conditions from a totally objects orientated perspective.

9.1

StringQuery

StringQuery is one of QuickDB's query systems and it's with which it interacts when the method "obtain", of AdminBase, is invoked passing as parameter the object on which the query applies, and a String containing the query. The use of the String is to make the query based only on the object's properties and make the desired comparisons referring only to the attributes of the object (as if they were public).

Example:

Based on the following classes:

```
public class Person{
    private int id;
    private String name;
    private java.sql.Date birth;
    private ArrayList<Phone> phone;
    private Address address;
    //Getters - Setters
}

public class Employee extends Person{
    private int id;
    private int code;
    private String rolDescription;
    //Getters - Setters
}

public class Address{
    private int id;
    private String street;
    private int number;
    //Getters - Setters
}

public class Phone{
    private int id;
    private String areaCode;
    private String number;
    //Getters - Setters
}
```

To get the Employee Object where the inherited attribute Address has as street "unnamed street" should only be done:

```
Employee e = new Employee();
admin.obtain(e, "address.street = 'unnamed street'");
```

Which would be equivalent to the SQL query:

```
SELECT Employee.id, Employee.code, Employee.rolDescription, Employee.parent_id
FROM Employee
JOIN Person ON Employee.parent_id = Person.id
JOIN Address ON Person.address = Address.id
WHERE Address.street = 'unnamed street'
```

This way is specified in the string the attributes of the object of which you want to make the query, taking into account that starts from the level of the class of object that is passed by parameter, and consulting the inherited attributes (of any level, ie the direct Parent of the Class, or the Parent of Parent of the Class) as their own. As can be seen in the example above, "address" is a reference to another object inherited from the parent class, which contains the attribute "street". Therefore, is placed "address.street", where "address" is the name of the attribute within the class "Person", which the queries system interprets as inherited by the class "Employee", and then asks for the attribute "street" Class "Address" by separating the instance and the attribute by a dot (.) as referring to public attributes.

StringQuery supports the following operators:

- AND, &&, &
- OR, ||, |
- =
- !
- <
- >
- LIKE, like

StringQuery greatly helps to create very complex queries in a easy way, but also presents some limitations as date operations, few operators, and having to specify the whole sentence in a String, thats why was created the system queries that is explained below.

9.2

QuickDB Query

This queries system carries out the creation of the query by invoking methods of classes "Query", "Where" and "DateQuery" (in case of operations with Dates), which has the advantage of promoting well-formed queries creations and minimize the work, as it is responsible for establishing the JOINS between tables wh.en necessary, etc. This queries system is used when invoke the "Obtain" method, passing as the only parameter the object on which we want to perform the operation.

9.2.1

Class Query

Where class provides us the following methods to making the query:

- **where(String field, {Class baseClass})**: Which initiates the query. "Field" is the name of the field on whom you want to perform a certain verification, and the attribute "baseClass" is optional, if not specified, it is assumed that the attribute belongs to the class of object passed to "Obtain" as a parameter or to any of its parent class, otherwise, must be specified the Class to which the attribute belongs.

All conditions stated in braces {} are optional.

- **and(FIELD, {CLASS})**: Concatenates the previous verification with the verification who starts with this new attribute (FIELD) through "AND".
- **or(FIELD, CLASS...)**: Concatenates the previous verification with the verification who starts with this new attribute (FIELD) through "OR".
- **group(FIELDS, {CLASS})**: Specifies by which fields (FIELDS) will be group the query, by placing them within a string (String) separated each by a comma (,). For each field within the string must specify the class to which belong them, unless they all belong to the base class directly or by inheritance.
- **whereGroup(FIELD, {CLASS})**: Corresponds to the SQL's "HAVING", and specifies a condition to those fields by which the query was grouped (requires to have applied a previously group with "group(...)").
- **sort(BOOLEAN, FIELDS, CLASS...)**: Sort the results of the query based on the fields specified in the string (Following the same methodology previously expressed of to separate each one by a comma (,)). For each field within the string must specify the class to which belong them, unless they all belong to the base class directly or by inheritance. The expressed boolean value determines the sort type, ascending if TRUE, and descending if FALSE.

9.2.2

Class Where

Where class provides us the following methods to making the query:

The following methods can receive a parameter by **Value** to perform the test, or a **String** with the name of a field and the **Class** to which it belongs to build the validation for the query.

- **equal(OBJECT, {OBJECT})**
- **greater(OBJECT, {OBJECT})**
- **lower(OBJECT, {OBJECT})**
- **equalORgreater(OBJECT, {OBJECT})**
- **equalORlower(OBJECT, {OBJECT})**
- **notEqual(OBJECT, {OBJECT})**

The following methods doesn't receive any parameter, but they specify a condition to the previously specified attribute:

- **not()**
- **isNull()**
- **isNotNull()**

Other methods:

- **between(OBJECT1, OBJECT2)**: This method can receive any parameter of primitive object type, or some other object that implements the method "toString()" in a coherent way, to see if the specified attribute value prior to the call of this operation is within the range that exists between these two values (both values must be of the same data type).
- **in({OBJECTS})**: This method receives a set of objects with which validate if the attribute specified prior to the call of this function, corresponds to any of the values received in this function by parameter.
- **match(EXPRESSION)**: This method receives by parameter a string and evaluates if the attribute specified prior to the call of this function contains the received string.If received a simple string, will verify if the attribute corresponds to a string like "EXPRESSION" (being any kind of string), on the contrary, if in the EXPRESSION is used any of the "%" characters (which represents any kind of characters) or "" (representing a character any), will validate by the exact EXPRESSION received without altering the beginning and end.
- **date()**: With this method, is specified that the attribute previously entered corresponds to a Date attribute type, so returns a reference to "DateQuery" to carry out the necessary date checks.

9.2.3

Class DateQuery

- **differenceWith(VALUE, {CLASS})**: This method determines the difference in days between the attribute previously expressed and the value passed by parameter, or another field of a table in the database (if specify the field and the Class).
- **month()**: Returns the month value of the attribute previously entered.
- **day()**: Returns the day value of the attribute previously entered.
- **year()**: Returns the year value of the attribute previously entered.

Then these returned values must be compared using one of the methods of the class "Where."

9.2.4

Searching

Once concatenated the methods to form the query, just remains the "execution" of the same, and there are two ways to do it, by adding as the last method in the concatenation:

- **find()**: which completes with the resulting data the object passed as parameter to "Obtain".
- **findAll()**: which returns a collection with the objects resulting from executing the query.

Examples:

Based on the following data model:

```
public class UserParent{
    private int id;
    private String description;
    private ReferenceQuery reference;
}

public class UserQuery extends UserParent{
    private int id;
    private String name;
}

public class ReferenceParent{
    private int id;
    private String valueParent;
}

public class ReferenceQuery extends ReferenceParent{
    private int id;
    private String value;
}

public class CompleteQuery{
    private int id;
    private String name;
    private double salary;
    private int age;
    private Date birth;
    private boolean cond;
}
```

Queries:

- Simple Query:

```
UserQuery user = new UserQuery();
admin.obtain(user).where("name").equal("son name").find();
```

- Simple Query of Inherited Attributes:

```
UserQuery user = new UserQuery();
admin.obtain(user).where("description").equal("parent description2").find();
```

- Query based on the attribute's value of a reference:

```
UserQuery user = new UserQuery();
admin.obtain(user).where("valueParent", ReferenceQuery.class).equal("son value").find();
```

- Query based on the inherited attribute's value of a reference:

```
UserQuery user = new UserQuery();
admin.obtain(user).where("valueParent", ReferenceQuery.class).equal("value Parent").find();
```

- Query using "between", "and" and "lower"

```
ArrayList array = admin.obtain(user).where("birth").between("1980-01-01", "2010-12-31").and("salary").lower(2000).findAll();
```

- Query using “in”, “or” and “match”

```
ArrayList array = admin.obtain(user).where("age").in(22, 23, 24, 26).or("name").match("Sarmentero").findAll();
```

- Query using “group”, “whereGroup” and “greater”

```
ArrayList array = admin.obtain(user).where("age").greater(10).group("salary").whereGroup("salary").greater(2000).findAll();
```

- Query using “differenceWidth”, “equal”

```
java.sql.Date date = new Date(104, 4, 22);  
ArrayList array = admin.obtain(user).where("birth").date().differenceWith(date.toString()).equal(2).findAll();
```

► [Sign in](#) to add a comment

©2010 Google - [Terms](#) - [Privacy](#) - [Project Hosting Help](#)

Powered by [Google Project Hosting](#)



Views

Updated May 20, 2010 by [fersoave](#)

10:

Views

The Views on QuickDB are used to create simpler representations of complex Data Structures, such as Objects compound by other Objects, etc. A View provides the possibility of obtaining data more quickly, as not having to process all objects hierarchically involved, but the values assigned directly to the View attributes, greatly reducing processing time.

To build up a View is necessary to create a Class extending from "View", which specifies a method called "query()" that must be implemented by the SubClass. This method "query()" returns a data "Object" type which may be a String containing a string specifying the query, or may be a data "QuickDB Query" type allowing to form the query with the facilities of this query system.

There are 3 other methods which implementation its optional, depending on the type of View one wants to create:

- **renameColumns():** It specifies in a string the new names that each of the fields will take from the resulting query. This names will be used to obtain the values and assign them to the specific attribute inside the View. If this method it's not implemented, the new names from the attributes of the View are taken by default in the order they appear.
- **columns():** It specifies in a string which attributes will be taken into account on the query, if this method it's not implemented, the names from each of the View attributes are taken in the order they appear.
- **classes():** It specifies in an Array of Class type the classes each of the specified attributes in "columns()" belong to. If this method it's not implemented, is assumed that all attributes are from the SuperClass specified on the query.

10.1

Examples

Based on the following Data Model:

```
public class ObjectViewTest1{
    private int id;
    private String name;
    private String account;
    private ObjectViewTest2 obj2;
}

public class ObjectViewTest2{
    private int id;
    private String description;
    private Date date;
}
```

The following Views are created:

1. Using the Query System "QuickDB Query", and rewriting all configuration methods of View

```
import cat.quickdb.db.View;
import cat.quickdb.query.Query;
import java.sql.Date;

public class ViewObject extends View{

    private String name;
    private String description;
    private String account;
    private Date dateView;

    @Override
    public Object query(){
        ObjectViewTest1 o1 = new ObjectViewTest1();
        Query query = Query.create(this.getAdminBase(), o1);
        query.where("account", ObjectViewTest1.class).equal("account test");
        return query;
    }

    @Override
    public String columns(){
        return "name, description, account, date";
    }

    @Override
    public String renameColumns(){
        return "name, description, account, dateView";
    }
}
```

```

@Override
public Class[] classes(){
    return new Class[]{ObjectViewTest1.class, ObjectViewTest2.class, ObjectViewTest1.class, ObjectViewTest2.class};
}
}

```

1. Using the query specification on a text string.

```

import cat.quickdb.db.View;
import java.sql.Date;

public class ViewObjectString extends View{

    private String name;
    private String description;
    private String account;
    private Date dateView;

    @Override
    public Object query(){
        return "SELECT ObjectViewTest1.name 'name', ObjectViewTest2.description 'description', " +
            "ObjectViewTest1.account 'account', ObjectViewTest2.date 'dateView' " +
            "FROM ObjectViewTest1 " +
            "JOIN ObjectViewTest2 ON ObjectViewTest1.obj2 = ObjectViewTest2.id " +
            "WHERE ObjectViewTest1.account = 'account test'";
    }
}

```

1. Using the Query System “QuickDB Query”, without implementing the method of renaming the resulting columns.

```

import cat.quickdb.db.View;
import cat.quickdb.query.Query;
import java.sql.Date;

public class ViewObjectWithoutRename extends View{

    private String name;
    private String description;
    private String account;
    private Date date;

    @Override
    public Object query(){
        ObjectViewTest1 o1 = new ObjectViewTest1();
        Query query = Query.create(this.getAdminBase(), o1);
        query.where("account", ObjectViewTest1.class).equal("account test");
        return query;
    }

    @Override
    public String columns(){
        return "name, description, account, date";
    }

    @Override
    public Class[] classes(){
        return new Class[]{ObjectViewTest1.class, ObjectViewTest2.class, ObjectViewTest1.class, ObjectViewTest2.class};
    }
}

```

10.1.1

Creating an Instance of the View

There are 2 ways to initialize a View, the first consist of executing the method “**initializeViews(...)**” of “AdminBase” which will initialize all instances created or to be created from the Data Model. And the other way is to give an active instance of AdminBase to the View:

```

//1:
AdminBase.initializeViews(AdminBase.DATABASE.MYSQL, "[HOST]", "[PORT]", "[DB]", "[USER]", "[PASSWORD]");

//2:
ViewObject view = new ViewObject();
view.initilizeAdminBase(admin);

```

10.1.2

Obtaining the Values of the View

Once the instance of View is created, you just need to use the methods “**obtain()**” or “**obtainAll()**” to get View results based on the objects from the Database:

```
//1:  
view.obtain();  
  
//2:  
ArrayList array = view.obtainAll();
```

10.1.3

Dynamics Query in a View

A View is specified based on a consult that represents it, but if during runtime you would like to change that query in order to add an other condition for the obtained Views to be particular to a situation, you can resort to the method “dynamicQuery” of the View, which, as seen in method “query()”, can be worked through String or using the Query System of QuickDB, which concatenates on the query that represents the View a new set of conditions to be evaluated.

Example:

```
//Using String:  
ViewObjectString view = new ViewObjectString();  
view.initializeView(admin);  
view.dynamicQuery("AND name LIKE '%Dynamic%'");  
view.obtain();  
  
//Using QuickDB Query  
ViewObject view = new ViewObject();  
view.initializeView(admin);  
view.dynamicQuery( ((Query) view.query()).and("name").match("Dynamic") );  
view.obtain();
```

► [Sign in](#) to add a comment

©2010 Google - [Terms](#) - [Privacy](#) - [Project Hosting Help](#)

Powered by [Google Project Hosting](#)



Example

Updated May 25, 2010 by [diego.sarmentero](#)

11:

Step by Step Example

This example will show a concrete application of QuickDB for persistence and management of a fairly complete data structure using both types with and without annotations. *(This example may differ for QuickDB 1.3 version currently in development due to some changes being made in the code notations)*

11.1:

Without Annotations

Create "Person" Class:

```
public class Person {
    private int id;
    private String name;
    private String surname;

    public Person() {
    }

    public Person(String name, String surname) {
        this.name = name;
        this.surname = surname;
    }

    public void setId(int id) {
        this.id = id;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void setSurname(String surname) {
        this.surname = surname;
    }

    public int getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public String getSurname() {
        return surname;
    }
}
```

Create "Address" Class:

```
public class Address {
    public int id;
    public String street;
    public int number;

    public Address() {
    }

    public Address(String street, int number) {
        this.street = street;
        this.number = number;
    }

    public int getId() {
```

```

        return id;
    }

    public int getNumber() {
        return number;
    }

    public String getStreet() {
        return street;
    }

    public void setId(int id) {
        this.id = id;
    }

    public void setNumber(int number) {
        this.number = number;
    }

    public void setStreet(String street) {
        this.street = street;
    }
}

```

Create "Buy" Class: In this case, because this class has an attribute which is a collection of primitive types, based on QuickDB restrictions it is necessary to annotate that attribute as follow:

```

import java.util.ArrayList;
import cat.quickdb.annotation.Column;
import cat.quickdb.annotation.Table;

@Table
public class Buy {

    private int id;
    private String article;
    private double price;
    @Column(collectionClass="java.lang.Integer")
    private ArrayList<Integer> codes;

    public Buy() {
    }

    public Buy(String article, double price, ArrayList<Integer> codes) {
        this.article = article;
        this.price = price;
        this.codes = codes;
    }

    public void setArticle(String article) {
        this.article = article;
    }

    public void setCodes(ArrayList<Integer> codes) {
        this.codes = codes;
    }

    public void setId(int id) {
        this.id = id;
    }

    public void setPrice(double price) {
        this.price = price;
    }

    public String getArticle() {
        return article;
    }

    public ArrayList<Integer> getCodes() {
        return codes;
    }

    public int getId() {
        return id;
    }

    public double getPrice() {
        return price;
    }
}

```

Create "Customer" Class:

```

import java.util.ArrayList;

```

```

public class Customer extends Person{
    private int id;
    private String email;
    private Address address;
    private ArrayList buy;

    public Customer() {
    }

    public Customer(String name, String surname, String email, Address address,
        ArrayList buy) {
        super(name, surname);
        this.email = email;
        this.address = address;
        this.buy = buy;
    }

    public void setAddress(Address address) {
        this.address = address;
    }

    public void setBuy(ArrayList buy) {
        this.buy = buy;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public Address getAddress() {
        return address;
    }

    public ArrayList getBuy() {
        return buy;
    }

    public String getEmail() {
        return email;
    }

    public void setId(int id) {
        this.id = id;
    }

    public int getId() {
        return id;
    }
}

```

Interacting with AdminBase

Now for the rest of this example, having the Data Model described before, a complete instance of Customer is going to be stored in the Data Base (The Data Base at this point is empty and no table has been created).

```

import java.util.ArrayList;
import cat.quickdb.db.AdminBase;

public class App {

    public static void main(String[] args) {
        //Create AdminBase instance for MySQL
        AdminBase admin = AdminBase.initialize(AdminBase.DATABASE.MYSQL, "localhost",
            "3306", "test", "root", "");

        //Create integer collection for Buy
        ArrayList codes1 = new ArrayList();
        codes1.add(4357673);
        codes1.add(5457334);
        //Create integer collection for Buy
        ArrayList codes2 = new ArrayList();
        codes2.add(4567);
        codes2.add(3345);

        //Create Buy Objects
        Buy buy1 = new Buy("cat food", 16.5, codes1);
        Buy buy2 = new Buy("citric juice", 7, codes2);
        Buy buy3 = new Buy("space ship", 599.99, null);
        //Add Buy Objects to a collection
        ArrayList buys = new ArrayList();
        buys.add(buy1);
        buys.add(buy2);
        buys.add(buy3);

        //Create Address Object
        Address address = new Address("unnamed street", 123);
    }
}

```

```

//Create Customer Object compound with
//the objects created before
Customer customer = new Customer("Diego", "Sarmentero",
    "diego.sarmentero@gmail.com", address, buys);

//Store Customer Object in the Database
admin.save(customer);
}
}
}

```

Tables automatically created to store the information:

- Address (Columns: id, street, number)
- Buy (Columns: id, article, price)
- BuyInteger (Columns: id, base, object)
- Customer (Columns: id, email, address, parent_id)
- CustomerBuy (Columns: id, base, related)
- Person (Columns: id, name, surname)

Recover complete "Customer" Object stored in the database:

```

import java.util.ArrayList;
import cat.quickdb.db.AdminBase;

public class App {

    public static void main(String[] args) {
        AdminBase admin = AdminBase.initialize(AdminBase.DATABASE.MYSQL, "localhost",
            "3306", "test", "root", "");

        Customer customer = new Customer();
        //In this way we will obtain the Customer Object
        //along with all the objects related to it
        //in the same state as they were stored
        admin.obtain(customer).where("name").equal("Diego").find();

        //Para modificar el Objeto solo es necesario manipularlo
        //To modify the object only is necessary to manipulate it
        //as any other object
        customer.setName("Leonardo");
        //And then execute the modification
        admin.modify(customer);
    }
}

```

11.2:

With Annotations

Create "Person" Class:

```

import cat.quickdb.annotation.*;

@Table
public class Person {

    @Column(type=Properties.TYPES.PRIMARYKEY)
    @ColumnDefinition(autoIncrement=true, primary=true,
        type=Definition.DATATYPE.INTEGER, length=11)
    private int id;
    @Column
    private String name;
    @Column
    private String surname;

    public Person() {
    }

    public Person(String name, String surname) {
        this.name = name;
        this.surname = surname;
    }

    public void setId(int id) {
        this.id = id;
    }

    public void setName(String name) {
        this.name = name;
    }
}

```

```

    public void setSurname(String surname) {
        this.surname = surname;
    }

    public int getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public String getSurname() {
        return surname;
    }
}

```

Create "Address" Class:

```

import cat.quickdb.annotation.*;

@Table
public class Address {

    @Column(type=Properties.TYPES.PRIMARYKEY)
    @ColumnDefinition(autoIncrement=true, primary=true,
        type=Definition.DATATYPE.INTEGER, length=11)
    public int id;
    @Column(type=Properties.TYPES.PRIMITIVE) //Por defecto es PRIMITIVE
    public String street;
    @Column
    @ColumnDefinition(type=Definition.DATATYPE.INTEGER)
    public int number;

    public Address() {
    }

    public Address(String street, int number) {
        this.street = street;
        this.number = number;
    }

    public int getId() {
        return id;
    }

    public int getNumber() {
        return number;
    }

    public String getStreet() {
        return street;
    }

    public void setId(int id) {
        this.id = id;
    }

    public void setNumber(int number) {
        this.number = number;
    }

    public void setStreet(String street) {
        this.street = street;
    }
}

```

Create "Buy" Class: In this case, because this class has an attribute which is a collection of primitive types, based on QuickDB restrictions it is necessary to annotate that attribute as follow:

```

import java.util.ArrayList;
import cat.quickdb.annotation.*;

@Table
public class Buy {

    @Column(type=Properties.TYPES.PRIMARYKEY)
    @ColumnDefinition(autoIncrement=true, primary=true,
        type=Definition.DATATYPE.INTEGER, length=11)
    private int id;
    @Column(name="item")
    private String article;
    @Column
    @ColumnDefinition(type=Definition.DATATYPE.DOUBLE)
    private double price;
    @Column(type=Properties.TYPES.COLLECTION, collectionClass="java.lang.Integer")

```

```

private ArrayList<Integer> codes;

public Buy() {
}

public Buy(String article, double price, ArrayList<Integer> codes) {
    this.article = article;
    this.price = price;
    this.codes = codes;
}

public void setArticle(String article) {
    this.article = article;
}

public void setCodes(ArrayList<Integer> codes) {
    this.codes = codes;
}

public void setId(int id) {
    this.id = id;
}

public void setPrice(double price) {
    this.price = price;
}

public String getArticle() {
    return article;
}

public ArrayList<Integer> getCodes() {
    return codes;
}

public int getId() {
    return id;
}

public double getPrice() {
    return price;
}
}

```

Create "Customer" Class:

```

import java.util.ArrayList;
import cat.quickdb.annotation.*;

@Parent
@Table
public class Customer extends Person{

    @Column(type=Properties.TYPES.PRIMARYKEY)
    @ColumnDefinition(autoIncrement=true, primary=true,
        type=Definition.DATATYPE.INTEGER, length=11)
    private int id;
    @Column
    private String email;
    @Column(type=Properties.TYPES.FOREIGNKEY)
    private Address address;
    @Column(type=Properties.TYPES.COLLECTION)
    private ArrayList buy;

    public Customer() {
    }

    public Customer(String name, String surname, String email, Address address,
        ArrayList buy) {
        super(name, surname);
        this.email = email;
        this.address = address;
        this.buy = buy;
    }

    public void setAddress(Address address) {
        this.address = address;
    }

    public void setBuy(ArrayList buy) {
        this.buy = buy;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public Address getAddress() {

```

```

    return address;
}

public ArrayList getBuy() {
    return buy;
}

public String getEmail() {
    return email;
}

public void setId(int id) {
    this.id = id;
}

public int getId() {
    return id;
}
}

```

Interacting with AdminBase

Now for the rest of this example, having the Data Model described before, a complete instance of Customer is going to be stored in the Data Base (The Data Base at this point is empty and no table has been created):

```

import java.util.ArrayList;
import cat.quickdb.db.AdminBase;

public class App {

    public static void main(String[] args) {
        //Create AdminBase instance for MySQL
        AdminBase admin = AdminBase.initialize(AdminBase.DATABASE.MYSQL, "localhost",
            "3306", "test", "root", "");

        //Create integer collection for Buy
        ArrayList codes1 = new ArrayList();
        codes1.add(4357673);
        codes1.add(5457334);
        //Create integer collection for Buy
        ArrayList codes2 = new ArrayList();
        codes2.add(4567);
        codes2.add(3345);

        //Create Buy Objects
        Buy buy1 = new Buy("cat food", 16.5, codes1);
        Buy buy2 = new Buy("citric juice", 7, codes2);
        Buy buy3 = new Buy("space ship", 599.99, null);
        //Add Buy Objects to a collection
        ArrayList buys = new ArrayList();
        buys.add(buy1);
        buys.add(buy2);
        buys.add(buy3);

        //Create Address Object
        Address address = new Address("unnamed street", 123);

        //Create Customer Object compound with
        //the objects created before
        Customer customer = new Customer("Diego", "Sarmentero",
            "diego.sarmentero@gmail.com", address, buys);

        //Store Customer Object in the Database
        admin.save(customer);
    }
}

```

Tables automatically created to store the information:

- Address (Columns: id, street, number)
- Buy (Columns: id, item, price)
- BuyInteger (Columns: id, base, object)
- Customer (Columns: id, email, address, parent_id)
- CustomerBuy (Columns: id, base, related)
- Person (Columns: id, name, surname)

Recover complete "Customer" Object stored in the database:

```

import java.util.ArrayList;
import cat.quickdb.db.AdminBase;

public class App {

```

```
public static void main(String[] args) {
    AdminBase admin = AdminBase.initialize(AdminBase.DATABASE.MYSQL, "localhost",
        "3306", "test", "root", "");

    Customer customer = new Customer();
    //In this way we will obtain the Customer Object
    //along with all the objects related to it
    //in the same state as they were stored
    admin.obtain(customer).where("name").equal("Diego").find();
}
}
```

Source Code used in this Example (Maven Project): [SOURCE](#)

► [Sign in](#) to add a comment

©2010 Google - [Terms](#) - [Privacy](#) - [Project Hosting Help](#)

Powered by [Google Project Hosting](#)