



Tutorial Paso a Paso!!

Trabajar con Bases de Datos nunca fue más fácil!

Índice de contenido

1	Usando QuickDB.....	4
1.1)	Que es QuickDB?.....	4
1.2)	Por qué QuickDB?.....	4
1.3)	Capacidades de QuickDB.....	4
1.4)	Agregando QuickDB a un Proyecto Java.....	5
1.5)	Agregar el Driver de la Base de Datos.....	6
1.6)	Requerimientos.....	6
2	Restricciones.....	7
3	Convenciones.....	8
3.1)	Para Claves Primarias.....	8
3.2)	Herencia.....	8
3.3)	Colecciones.....	9
3.4)	Getters y Setters.....	9
3.5)	Tipos de Datos.....	9
4	Anotaciones.....	10
4.1)	Table.....	10
4.2)	Parent.....	10
4.3)	Column.....	10
4.4)	ColumnDefinition.....	13
4.5)	Validaciones.....	13
4.6)	Modo Mixto.....	16
4.7)	Tipos de Datos.....	16
5	Creación de Tablas.....	18
6	AdminBase.....	21
6.1)	Creando Instancia de AdminBase.....	21
6.2)	Operaciones.....	22
6.2.1)	save.....	22
6.2.2)	saveAll.....	22
6.2.3)	saveGetIndex.....	22
6.2.4)	modify.....	23
6.2.5)	modifyAll.....	23
6.2.6)	delete.....	23
6.2.7)	obtain.....	23
6.2.8)	obtainAll.....	24
6.2.9)	obtainWhere.....	24
6.2.10)	obtainSelect.....	24
6.2.11)	obtainJoin.....	25
6.2.12)	lazyLoad.....	25
6.2.13)	executeQuery.....	26
6.2.14)	checkTableExist.....	26
6.2.15)	Transacciones.....	26
7	AdminBinding.....	28
7.1)	Operaciones.....	29
7.1.1)	save.....	29
7.1.2)	saveGetIndex.....	29
7.1.3)	modify.....	29
7.1.4)	obtain.....	29
7.1.5)	obtainSelect.....	30

7.1.6)	obtainWhere.....	30
7.1.7)	lazyLoad.....	30
8	AdminThread.....	31
9	Consultas.....	32
9.1)	StringQuery.....	32
9.2)	QuickDB Query.....	34
9.2.1)	La Clase Query nos brinda los siguientes métodos con los que ir armando la consulta:	34
9.2.2)	La Clase Where brinda los siguientes métodos con los que ir armando la consulta:....	34
9.2.3)	La Clase DateQuery brinda los siguientes métodos con los que ir armando la consulta:	35
9.2.4)	Realizando la Búsqueda.....	36
10	Vistas.....	39
10.1)	Ejemplos.....	39
10.1.1)	Creando una Instancia de la Vista.....	42
10.1.2)	Obteniendo los Valores de la Vista.....	42
10.1.3)	Consultas Dinámicas en una Vista.....	42

1 Usando QuickDB

1.1) Que es QuickDB?

QuickDB es una librería que permite a un desarrollador centrarse en la definición de las Entidades que mapean las Tablas de la Base de Datos y realizar operaciones que permitan la interacción entre estas Entidades y la Base de Datos sin tener que realizar tediosas configuraciones, dejando simplemente que la librería a través de la estructura del objeto infiera como realizar el mapeo del objeto a la Base de Datos.

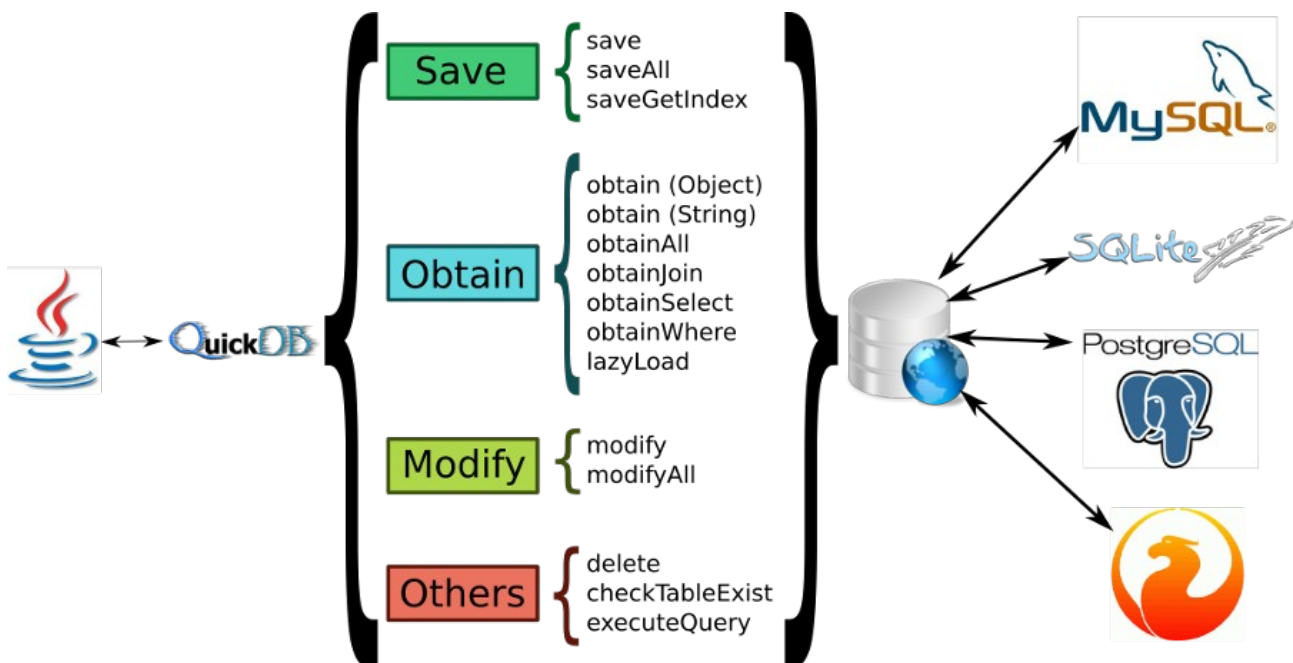
El Modelo de Datos que se vaya a persistir NO es necesario que implemente ninguna interfaz o herede de ninguna clase. Para trabajar con este Modelo de Datos QuickDB utiliza ciertas Anotaciones o también puede recurrirse al Modo Intuitivo (siguiendo ciertas Convenciones), donde QuickDB en base al tipo de dato de cada objeto y otras características determina como realizar los respectivos mapeos (también es posible la combinación de atributos con y sin anotaciones, como se vera mas adelante).

1.2) Por qué QuickDB?

Porque la funcionalidad no es sinónimo de complejidad.

Ser capaz de realizar diversas operaciones no debe implicar perder mucho tiempo en tareas de configuración.

1.3) Capacidades de QuickDB



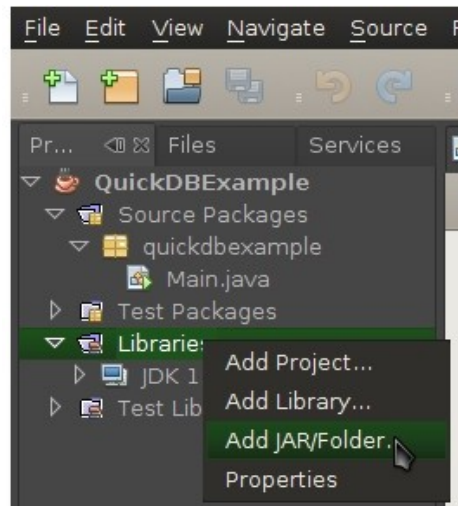
Estas capacidades pueden ser combinados con el uso de Herencia, Objetos compuestos, Colecciones (relación muchos-a-muchos, relación uno-a-muchos) y Creación automática de tablas, como ya se vera mas adelante.

1.4) Agregando QuickDB a un Proyecto Java

Para utilizar QuickDB es necesario incorporar esta librería en un proyecto Java, QuickDB utiliza los distintos drivers JDBC dependiendo de la Base de Datos con la que vaya a conectarse, de esta forma, no es necesario que QuickDB implemente funcionalidades específicas para cada motor de Base de Datos, sino que solo debe ser capaz de conectarse a uno de estos drivers, lo cual realiza en tiempo de ejecución. Teniendo como ventaja a su vez, no cargar al proyecto donde se incluya QuickDB con código que no va a utilizarse para cierto proyecto específico.

Para incorporar QuickDB a un Proyecto hay diversas formas, las mas comunes suelen ser:

- Agregar la librería al Proyecto desde el IDE (Ejemplo con Netbeans):



- Agregar la librería como dependencia en un Proyecto Maven:

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

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, para lo cual hay 2 formas de habilitar su disponibilidad:

1. Descargar la librería y luego instalarla en el Repositorio Local:

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

2. Agregar el Repositorio de QuickDB al Proyecto Maven:

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

1.5) Agregar el Driver de la Base de Datos

Una vez que ya se cuenta con un Proyecto armado con QuickDB incorporado, es necesario agregar el driver que conectara QuickDB con la Base de Datos.

QuickDB actualmente trabaja con las siguientes Bases de Datos:

DBMS	Descarga de Driver
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

El proceso para agregar cualquiera de los drivers es el mismo descripto para la incorporación de QuickDB.

1.6) Requerimientos

- **QuickDB trabaja con la Versión 1.5 de Java o Superior.**

Los Versiones de los Drivers con los que se han realizado las pruebas son las siguientes:

- MySQL: Versión 5.1.5 o Superior
- PostgreSQL: JDBC4 Versión 8.4-701 (o Compatible)
- SQLite: Versión 056 (o Compatible)
- Firebird: Versión 2.1.6 (o Compatible)

2 Restricciones

Para trabajar con QuickDB hay ciertas restricciones que hay que tener en cuenta:

- Todas las clases del modelo deben contener un atributo entero de clave primaria, y debe ser el primer atributo declarado en la Clase.
- Todas las clases del modelo deben contener un Constructor Vacío.
- En el caso de una relación de Clases Padre-Hijo, si los métodos Get y Set de la clave primaria llevan el mismo nombre, entonces tanto el Padre como el Hijo deben tener el mismo valor de Clave Primaria (ya que al ser públicos los métodos el valor del Hijo sobrescribiría al del Padre), impidiendo que otra Clase a ser almacenada en la Base de Datos extienda de la misma Clase Padre (Se soluciona utilizando Anotaciones).
- Para combinar el uso en una Clase de atributos con y sin anotaciones, es necesario agregar la anotación `@Table` a la Clase.

3 Convenciones

Las convenciones se utilizan para eliminar las tareas de configuración dentro de QuickDB, hay muchas propiedades que pueden ser deducidas automáticamente sin necesidad de configuración (aunque todo es posible de configurar si se desea como se vera en la sección siguiente de Anotaciones), y a continuación se muestran cuales son los detalles a tener en cuenta en el Modelo de Datos para que sea soportado por QuickDB sin necesidad de realizar mas que la escritura de las Entidades.

Estas convenciones son necesarias al no trabajar con Anotaciones, de lo contrario, pueden escribirse las Entidades de cualquier otra forma siempre que se especifiquen dichas características mediante las anotaciones.

3.1) Para Claves Primarias

Cada Clase del modelo debe declarar como primer atributo una variable entera de nombre “id”:

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

3.2) Herencia

Para que QuickDB pueda reconocer la herencia automáticamente, es necesario que la Clase de la que se extienda se encuentre dentro del mismo paquete que la Clase hija, esto posibilita poder determinar cuando dejar de analizar la herencia de forma recursiva y no llegar hasta el nivel de Object.

```
package com.example;  
  
public class Parent{  
    ...  
}  
  
package com.example;  
  
public class Son extends Parent{  
    ...  
}
```


3.3) Colecciones

Al trabajar con colecciones, solo son soportadas aquellas que implementan la interfaz "java.util.List" o "java.util.Collection", y cuando se trabaja sin anotaciones además el nombre del atributo debe ser igual al nombre de la clase de los objetos que compondrán dicha colección (con la primer letra en minúscula si se desea):

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

public class Phone{
    ...
}
```

Para colecciones que contengan solo tipos de datos primitivos, se explicara en la sección "Anotaciones" como contemplar este caso.

3.4) Getters y Setters

Al no trabajar con anotaciones, QuickDB debe determinar automáticamente cuales serán los métodos desde los que se obtendrán los valores de los atributos, y cuales serán los métodos a través de los cuales se le podrá dar valores a los atributos. Para hacer esto se sigue la convención por defecto de escribir "get" o "set" y luego el nombre del atributo con 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) Tipos de Datos

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

4 Anotaciones

4.1) Table

Table puede estar acompañada de un parámetro o no, el parámetro de Table indica a que tabla de la Base de Datos se mapeara esta entidad. Al no incluir el parámetro simplemente se asume que la Tabla lleva el mismo nombre que la Clase.

- Especificando directamente a cual Tabla hace referencia una Clase:

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

@Table("TableExample")
public class Example {

}
```

- Tomando como nombre de la Tabla el nombre de la Clase:

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

@Table
public class Example {

}
```

4.2) Parent

Parent indica que esta Clase hereda de otra y que por lo tanto los atributos de la Clase Padre se guardaran en otra tabla (esta anotación no lleva ninguna atributo).

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

@Parent
@Table
public class ExampleSon {

}
```

Utilizando esta anotación se puede utilizar Clases Padre que no estén dentro del mismo paquete, lo cual era una restricción al trabajar simplemente con las Convenciones.

4.3) Column

En cuanto a las anotaciones Column, estas se utilizan para definir de forma completa un atributo de la entidad con su correspondiente mapeo dentro de una Tabla en la Base de Datos.

Los distintos parámetros de la Anotación Column, poseen todos valores por defecto, por lo que solo es necesario completar aquellos que sean necesarios para determinado caso particular.

- **name:** Nombre del Campo de dicha Tabla en la Base de Datos, si no se especifica toma el mismo nombre del atributo.
- **type:** Indica el tipo de dato del atributo, el cual puede ser PRIMARYKEY, FOREIGNKEY, COLLECTION, PRIMITIVE (si no se especifica se asume que el tipo es PRIMITIVE, el cual es una de las primitivas del lenguaje contemplando además String y java.sql.Date).
- **getter:** Este parámetro presenta la posibilidad de poder expresar explícitamente cual sera el método del cual obtendremos el valor de dicho atributo (si no se especifica se asume que si el atributo lleva el nombre "atributo" el método getter sera "getAtributo").
- **setter:** El mismo caso anterior pero para los métodos de seteo.
- **collectionClass:** Cuando el atributo marcado con esta anotación es una colección se puede decir explícitamente que tipo de objetos contendrá esta colección.
- **ignore:** Colocándole como valor True le decimos a QuickDB que no tenga en cuenta este atributo al momento de persistir el objeto.

Todos los parámetros no especificados siguen por defecto el comportamiento establecido en las Convenciones.

Si una clase tuviera un atributo "name" y ese atributo representara un campo con el mismo nombre en su respectiva Tabla en la Base de Datos, fuera de tipo String (por ejemplo), y sus respectivos métodos de get y set fueran "getName" y "setName", entonces para la Anotación de Column bastaría con que se colocara:

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

Lo cual seria el equivalente a dejar dicho atributo sin anotación ya que sigue las convenciones.

Si por el contrario, el atributo antes mencionado no siguiera ninguna de las condiciones establecidas, podríamos indicar cual es el nombre del Campo en la Base de Datos al que hace referencia, su tipo, y sus métodos de get y set:

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

@Table("TableExample")
public class Example {

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

Para el caso del parámetro "collectionClass" de Column, hay 2 formas de especificarlo:

- Colocando solo el nombre de la Clase cuyos objetos formaran parte de la colección, si esta misma se encuentra dentro del mismo paquete de la Clase que contiene el atributo de Colección:

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

- O especificando la ruta completa (Paquete.Clase) en el caso que se encontrara en otro Paquete:

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

Para las Colecciones de tipos de Datos Primitivos (int, double, float, short, long, boolean, y tambien String y Date), es necesario utilizar anotaciones, y se especifican de la siguiente forma:

```
@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 integer;

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

Es importante tener en cuenta que solo es necesario especificar aquellos parámetros que sean necesarios, si solo quisiéramos indicar el tipo podríamos poner:

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

Los distintos **Tipos** que pueden especificarse son:

- **PRIMITIVE:** Se refiere a aquellos tipos que son mapeados directamente a los Tipos de Datos soportados por la Base de Datos.
- **PRIMARYKEY:** Hace referencia a la clave primaria de dicha Tabla y es de importancia especificarla, ya que indica a la librería que en casos de inserciones no se intente forzar el valor de este campo y se deje a la Base de Datos manejarlo.
- **FOREIGNKEY:** Hace referencia a las claves foráneas, las cuales son representadas en la Base de Datos por referencias a otras Tablas, y en las Entidades por referencias a otros objetos, de esta forma cuando se tiene en una Clase una referencia a otro objeto que también debe ser mapeado a una Tabla, este debe ser marcado como tipo "FOREIGNKEY".
- **COLLECTION:** Especifica si dicho atributo es del tipo colección para trabajarlo como una relación uno-a-muchos o muchos-a-muchos (crea las Tablas que fueran necesarias para llevar a cabo esta función).

4.4) ColumnDefinition

ColumnDefinition establece las propiedades con las que se creara la Tabla en la Base de Datos al tratar de hacer la primera inserción si la Tabla no existe. Los parámetros que recibe son:

- **type:** El tipo de dato con el que se creara dicha columna en la Base de Datos. Por defecto el tipo es VARCHAR
- **length:** Largo del tipo de Dato. Por defecto no especifica largo a los tipos, salvo a VARCHAR que le asigna un largo predeterminado de 150
- **notNull:** Especifica si la columna puede aceptar valores nulos o no. Por defecto su valor es true.
- **defaultValue:** El valor que tiene cada campo de la columna por defecto. Por defecto su valor es una cadena vacía.
- **autoIncrement:** Valor por defecto false.
- **unique:** Estable si el valor del campo es de tipo único. Valor por defecto false.
- **primary:** Estable si la columna es de tipo clave primaria. Valor por defecto false.
- **format:** Estable el formato de la columna. Por defecto DEFAULT. (Solo para MySQL)
- **storage:** Establece el tipo de Storage a utilizar. Por defecto DEFAULT. (Solo para MySQL)

Ejemplo:

```
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) Validaciones

Las validaciones permiten realizar comprobaciones automáticas sobre la Entidad para determinar si dicho Objeto sera guardado o modificado en la Base de Datos. Si la condición de Validación no se cumpliera el proceso de actualización se suspendería.

Una Validación se puede llevar a cabo a través de las siguientes condiciones:

- **conditionMatch:** Recibe un array de Strings con las distintas expresiones regulares a

evaluar.

- **maxLength:** Especifica el largo máximo que puede tener cierto campo (incluido el valor especificado).
- **numeric:** Realiza comprobaciones de tipos numericas sobre el campo. Recibe un array con un conjunto de valores numericos, estando organizados de esta forma: *[condición, valor]*.
- **date:** Realiza comprobaciones sobre un objeto de tipo fecha. Recibe un array con un conjunto de valores numericos, estando organizados de esta forma: *[parte_de_fecha, condición, valor]*.

En cuanto a las expresiones regulares para “conditionMatch”, la anotación *Validation* cuenta con 4 expresiones ya armadas de las que se puede elegir, en lugar de crear nuevas, estas son:

- **conditionURL:** Evalúa que el texto ingresado en dicho campo sea una expresión URL bien formada.
- **conditionMail:** Evalúa que el texto ingresado en dicho campo sea un e-mail bien formado.
- **conditionSecurePassword:** Evalúa que el texto ingresado en dicha variable sea un password seguro (contemplando mayúsculas, minúsculas, números y caracteres especiales).
- **conditionNotEmpty:** Evalúa que la variable no contenga una cadena vacía.

Para las comprobaciones de “numeric” las condiciones de las que se puede elegir son:

- **EQUAL**
- **LOWER**
- **GREATER**
- **EQUALORLOWER**
- **EQUALORGREATER**

Para las comprobaciones de “date” las partes de la fecha disponible son:

- **YEAR**
- **MONTH**
- **DAY**

Y las condiciones son las mismas que se utilizan en “numeric”.

Algunos ejemplos:

1. Ejemplo completo utilizando todas las validaciones disponibles.

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

public class ValidUser {

    private int id;
    @Validation(maxLength=20, conditionMatch={Validation.conditionNotEmpty})
    private String name;
    @Validation(conditionMatch={Validation.conditionSecurePassword})
    private String pass;
    @Validation(conditionMatch={Validation.conditionMail})
    private String mail;
    @Validation(conditionMatch={Validation.conditionURL})
    private String web;
    @Validation(date={Validation.YEAR, Validation.EQUALORGREATER, 2000})
    private Date birthDate;
    @Validation(numeric={Validation.GREATER, 18})
    private int age;

}
```

2. Ejemplo de validaciones numéricas.

```
import cat.quickdb.annotation.Validation;

public class ValidComplexNumeric {

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

}
```

3. Ejemplo de validaciones de fecha.

```
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) Modo Mixto

Es posible trabajar con Clases donde se especifiquen algunos atributos con Anotaciones y otros sin Anotaciones (siendo estos últimos los que siguen las convenciones), para lo cual solo hay que tener en cuenta de colocar la anotación @Table al nivel de la Clase (como se mencionó en la sección de “Restricciones”).

4.7) Tipos de Datos

Los tipos de Datos que pueden ser definidos con la anotación ColumnDefinition son:

Java	MySQL	PostgreSQL	Firebird	SQLite
BIT	BIT	bit	bit	Tipado Dinámico
BOOLEAN	TINYINT(1)	boolean	boolean	Tipado Dinámico
SMALLINT	SMALLINT	smallint	smallint	Tipado Dinámico
INT	INT	integer	int	Tipado Dinámico
INTEGER	INTEGER	integer	integer	Tipado Dinámico
BIGINT	BIGINT	bigint	bigint	Tipado Dinámico
REAL	REAL	real	real	Tipado Dinámico
DOUBLE	DOUBLE	double precision	double precision	Tipado Dinámico
FLOAT	FLOAT	real	float	Tipado Dinámico
DECIMAL	DECIMAL	decimal	decimal	Tipado Dinámico
NUMERIC	NUMERIC	numeric	numeric	Tipado Dinámico
DATE	DATE	date	date	Tipado Dinámico
TIME	TIME	time	time	Tipado Dinámico
TIMESTAMP	TIMESTAMP	timestamp	timestamp	Tipado Dinámico
DATETIME	DATETIME	timestamp	timestamp	Tipado Dinámico
CHAR	CHAR	character	char	Tipado Dinámico
VARCHAR	VARCHAR	character varying	varchar	Tipado Dinámico

Java	MySQL	PostgreSQL	Firebird	SQLite
BINARY	BINARY	bytea	---	Tipado Dinámico
VARBINARY	VARBINARY	bit varying	---	Tipado Dinámico
TEXT	TEXT	text	nchar	Tipado Dinámico

5 Creación de Tablas

QuickDB infiere a través del tipo de dato como crear la respectiva Tabla en la Base de Datos.

Para la Creación de las tablas se tiene en cuenta los conceptos explicados en la sección de “Convenciones” y “Anotaciones”.

En caso de no existir Anotaciones, se crea la Tabla tomando el nombre de la Clase como nombre de la Tabla, se utiliza el atributo (obligatorio) entero “id” como clave primaria de la Tabla auto incremental y se crean las demás columnas de acuerdo a los Tipos de Datos especificados en la sección “Convenciones”, todas con propiedad NOT NULL.

En el caso de los Strings se crea el campo con un largo igual a la cadena ingresada o igual a 150 si la primer cadena ingresada era menor a este tamaño. En cuanto a la Herencia, se agrega un campo en la Tabla con el nombre “*parent_id*” el cual contendrá el valor de la clave primaria de la Tabla Padre. Para las colecciones se ignora el campo al mapear la Clase a una Tabla y se crea una Tabla Relacional entre la Clase y los elementos contenidos en la colección.

Para el caso de trabajar con Anotaciones, todas estas propiedades mencionadas y varias mas, pueden ser completamente configuradas utilizando la anotación *@ColumnDefinition* explicada previamente.

A continuación un ejemplo completo utilizando todas las anotaciones descriptas en la sección anterior y las tablas resultantes al realizar la primera inserción:

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

@Table("ModelParentTest")
public class ModelParent {

    private int id;
    private String description;
}
```

```

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

@Table("CollecAnnotation")
public class CollectionAnnotation {

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

}

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

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

    @Column(type=TYPES.PRIMARYKEY)
    @ColumnDefinition(primary=true, autoIncrement=true,
        length=11, unique=true, type=DATATYPE.INTEGER)
    private int idModel;
    @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;

}

```

Tabla: “ModelParentTest” (Clase: “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

Tabla: “CollecAnnotation” (Clase: “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

Tabla: “AnnotationModelCollecAnnotation” (relación entre la Clase que contiene la colección y los items dentro de la colección)











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

Tabla: “AnnotationModel” (Clase: “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

Cabe destacar que muchas de las anotaciones utilizadas podrían haberse obviado y seguir las convenciones, y se obtendrían los mismos resultados en el caso de la definición de los tipos de datos.

6 AdminBase

AdminBase es la Clase principal de la Librería y es con la cual se realizarán todas las operaciones relacionadas con la Base de Datos. Una instancia de AdminBase debe ser creada para poder luego usar ese objeto para los fines antes mencionados.

6.1) Creando Instancia de AdminBase

Al crear la instancia, deben pasarse ciertos parámetros con los que AdminBase podrá realizar la conexión a la Base de Datos:

Los Parámetros de AdminBase son:

- **Host**
- **Puerto**
- **Nombre de la Base de Datos**
- **Usuario**
- **Password**
- **Schema** (Solo para Postgre)

Ejemplos:

- Para MySQL:

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

- Para Postgre:

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

- Para Firebird:

```
AdminBase admin = AdminBase.initialize(DATABASE.FIREBIRD, "localhost",
    "3050", "/home/gato/employee.fdb", "SYSDBA", "firebird");
```

- Para SQLite:

(en este caso solo se especifica el tipo de Base de Datos y el nombre de la misma)

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

6.2) Operaciones

Las operaciones son todas aquellas acciones que pueden realizarse entre un Objeto del Modelo de Datos y la Base de Datos.

Para explicar cada una de las operaciones, se tomara como ejemplo la siguiente clase:

```
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

Para guardar un Objeto en la Base de Datos, solo debemos crear una instancia del mismo y luego invocar el método “save” para que se encargue de llevar a cabo dicha tarea.

Ejemplo:

```
Person person = new Person("leeloo", 2);
admin.save(person);
```

6.2.2) saveAll

Para guardar una colección de objetos, procedemos a crear dicha colección y luego solo ejecutamos "saveAll" de AdminBase.

Ejemplo:

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

admin.saveAll(array);
```

6.2.3) saveGetIndex

Para guardar un Objeto en la Base de Datos y obtener el valor del índice generado para el mismo, solo debemos crear una instancia del mismo y luego invocar en AdminBase el método “saveGetIndex”.

Ejemplo:

```
Person person = new Person("leeloo", 2);  
int id = admin.saveGetIndex(person);
```

6.2.4) modify

Para modificar un Objeto en la Base de Datos, debemos obtener previamente el objeto almacenado en la base de datos, aplicar las modificaciones pertinentes sobre ese objeto y luego ejecutar la modificación.

Ejemplo:

```
Person p = new Person();  
admin.obtain(p, "age=23");  
  
p.setName("Leonardo");  
admin.modify(p);
```

6.2.5) modifyAll

Se aplica el mismo caso que para "modify", se obtiene primero la colección que se desea, se modifican los valores de esa colección y se ejecuta "modifyAll".

Ejemplo:

```
Person p = new Person();  
ArrayList array = admin.obtainAll(p, "age=23");  
  
for(Object o : array){  
    //Realizar modificaciones  
}  
  
admin.modifyAll(array);
```

6.2.6) delete

Se obtiene el Objeto que se desea eliminar y luego se ejecuta "delete".

Ejemplo:

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

6.2.7) obtain

El método "*obtain*" se encuentra sobrecargado, ya que esta la opción de trabajar con Obtain a través de cualquiera de los 2 Sistemas de Consultas de QuickDB (Los cuales se explicaran en la sección 9).

Por lo tanto "obtain" puede recibir como parámetro el Objeto sobre el que se desea obtener los datos, siendo este el sistema de QuickDB denominado "Query":

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

O bien “obtain” puede recibir 2 parámetros, siendo el primer el Objeto sobre el que se desea obtener los datos, y el segundo un String especificando las condiciones de la búsqueda basado en las características del segundo sistema de QuickDB denominado “StringQuery”:

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

6.2.8) obtainAll

Para obtener de la Base de Datos una Colección de Objetos, solo debemos crear una instancia del mismo vacía y luego decirle a AdminBase que recupere los objetos que cumplan con la condición.

Ejemplo:

```
Person person = new Person();
ArrayList array = admin.obtainAll(person, "age=23");
```

También esta la opción de realizar la búsqueda de Colecciones de Objetos con el sistema “Query” de QuickDB como ya se vera en la sección 9.

6.2.9) obtainWhere

Este método nos permite especificar únicamente la sección del WHERE en una consulta SQL y nos devolverá el objeto resultante.

Es mas rápido que los sistemas de Queries de QuickDB ya que no debe inferir sobre la estructura del Objeto para realizar los JOINS que sean pertinentes para llevar a cabo la consulta, pero por lo tanto solo sirve para consultas simples sobre el mismo objeto, no pudiendo involucrar las relaciones que tenga un Objeto con otro.

Ejemplo:

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

6.2.10) obtainSelect

Este método devuelve el Objeto resultante al ejecutar la consulta SQL especificada explícitamente.

La consulta especificada debe referirse al Objeto que se pasa por parámetro, y debe estar completamente definida.

Ejemplo:

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


6.2.11) obtainJoin

A través de este método es posible obtener un Array (Object[]), donde cada elemento de este array sera a su vez un String con la cantidad de elementos igual a las columnas que se especifico en la búsqueda (es decir, este método devuelve una representación de la tabla resultante como un objeto matriz). Es de utilidad para completar los datos de un Componente Gráfico como una Tabla, etc.

Debe especificarse la consulta SQL completa:

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

6.2.12) lazyLoad

Mediante este método es posible ir cargando un Objeto de forma progresiva a medida que sea necesario.

Por ejemplo, para las siguientes clases:

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

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

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

Realizamos la carga solo de los atributos de Person(1), luego agregamos los valores de los atributos de Phone(2) y por ultimo los de Company(3):

```
Person p = new Person();

//Atributos de Person(1)
admin.lazyLoad(p, "age=23");
//Phone es NULL en este momento

//Agregamos los atributos de Phone
admin.lazyLoad(p);
//Phone ya tiene sus valores, salvo por Company que es NULL

//Agrega los valores de Company
admin.lazyLoad(p);
```

6.2.13) executeQuery

“executeQuery” quizás sea el método mas simple, ya que lo único que hace es ejecutar una sentencia en la Base de Datos y lo único que devuelve es Verdadero si se ejecuto con Éxito o Falso en caso contrario.

Ejemplo:

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

6.2.14) checkTableExist

Determina si existe una determinada Tabla (devuelve True), o si de lo contrario dicha Tabla no existe (devuelve False).

Ejemplo:

```
if(admin.checkTableExist("Person")){  
    System.out.println("La Tabla Existe");  
}else{  
    System.out.println("La Tabla No Existe");  
}
```

6.2.15) Transacciones

QuickDB por defecto maneja las transacciones a nivel del Objeto con el que se comienza la operación, es decir, el Objeto, ya sea simple, compuesto de otros objetos, con colecciones, o con herencia, debe contemplar la operación solicitada (junto con todos los objetos que involucre) con éxito, de lo contrario se descartan todos los cambios que se realizaran durante la operación.

Pero también existe la posibilidad de establecer que todas las operaciones se ejecuten de forma independiente, sin importar lo que suceda con los Objetos relacionados, de la siguiente forma:

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

A su vez, es posible manejar las transacciones de manera explicita, para los casos donde se pretenda guardar cierta cantidad de Objetos o ninguno:

```
Example example = new Example();  
Person person = new Person("leeloo", 2);  
  
admin.openAtomicBlock();  
admin.save(example);  
admin.save(person);  
admin.closeAtomicBlock();
```

Si se quisiera validar que todas las operaciones se realizaron con éxito o de lo contrario cancelar la transacción y volver la base al estado previo, se podría realizar de la siguiente forma:

```
Example example = new Example();
Person person = new Person("leelo", 2);
boolean value = true;
admin.openAtomicBlock();
value &= admin.save(example);
value &= admin.save(person);
if (value) {
    admin.closeAtomicBlock();
} else {
    admin.cancelAtomicBlock();
}
```

7 AdminBinding

AdminBinding permite al desarrollador heredar de esta clase, y poder acceder a los métodos de interacción con la Base de Datos a través del propio Objeto, sin tener que interactuar directamente con AdminBase.

Como su nombre lo indica, la función de AdminBinding es justamente la de establecer un vínculo o enlace entre los Objetos del Modelo de Datos y AdminBase. De esta forma la interacción podría realizarse de forma mas natural inclusive al decirle al propio objeto la operación que se quiere ejecutar.

Es necesario antes de comenzar a trabajar con las entidades del modelo ejecutar la siguiente linea de código, la cual inicializara la conexión con la Base de Datos para todas las instancias del Modelo (se encuentren o no creadas a este momento):

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

Donde las Propiedades son las mismas que para la inicialización de AdminBase:

- La Base de Datos a Utilizar.
- Host
- Puerto
- Base de Datos
- Usuario
- Password
- Schema (Solo para Postgre)

Los Métodos disponibles al heredar de esta Clase son:

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

Métodos como "obtainAll", "saveAll", etc. Se han dejado afuera porque no se refieren específicamente al objeto que los contiene, sino que se refieren a colecciones u operaciones independientes del objeto.

Ejemplos Basados en la siguiente entidad:

```
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) Operaciones

Las operaciones se especifican de la misma forma que se hace en “AdminBase” con la diferencia de que se omite de pasar el primer atributo que era el Objeto en si sobre el que se deseaba realizar la operación, ya que ahora el Objeto pasaría a ser el que ejecutaría las operaciones.

Ejemplos:

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

(Los dos sistemas de Consultas de QuickDB son soportados, para el otro caso simplemente se ejecutaría el método “obtain()” sin parámetros)

```
Person p = new Person();

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

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

Para las siguientes Clases:

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

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

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

Se ejecutaría la operación de la siguiente forma:

```
Person p = new Person();

//Atributos de Person(1)
p.lazyLoad("age=23");
//Phone es NULL en este momento

//Agregamos los atributos de Phone
p.lazyLoad("");
//Phone ya tiene sus valores, salvo por Company que es NULL

//Agrega los valores de Company
p.lazyLoad("");
```

8 AdminThread

AdminThread expone prácticamente las mismas funcionalidades que AdminBase, con la diferencia de que todas las operaciones ejecutadas son creadas en un hilo de ejecución distinto. Esta funcionalidad permite realizar operaciones que involucren a Objetos o Colecciones de Objetos muy grandes sin bloquear la aplicación.

Un detalle a tener en cuenta, es que como las operaciones se ejecutan en un hilo aparte, no se sabe en que momento se van a ejecutar, y por lo tanto, ninguno de los métodos devuelve ningún valor.

Hay 2 formas de inicializar un objeto AdminThread:

```
AdminThread adminThread1 = new AdminThread(admin);  
  
AdminThread adminThread2 = new AdminThread(DATABASE.MYSQL,  
                                             "localhost", "3306", "testQuickDB", "root", "");
```

La primera consiste en pasarle al constructor una instancia ya creada de AdminBase de la cual hará uso para las operaciones, y la segunda consiste en pasarle los mismos datos que se le pasan al método de inicialización de AdminBase para que AdminThread cree su propia instancia de AdminBase.

Las operaciones soportadas por **AdminThread** son:

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

Operaciones como “saveGetIndex” se han dejado afuera, ya que la finalidad misma de la operación es obtener el valor de retorno, y ninguna de las funciones expuestas por AdminThread tiene valor de retorno.

También es importante tener en cuenta que aunque se cuenta con las operaciones de “obtain...”, estas no aseguran que el Objeto sea cargado en ese momento, ya que no se sabe cuando se ejecutara el hilo.

9 Consultas

QuickDB implementa 2 sistemas de Consultas, los cuales facilitan mucho la tarea de obtener determinado objeto, ya que promueve la creación de las consultas y especificación de las condiciones desde una perspectiva totalmente orientada a objetos.

9.1) *StringQuery*

StringQuery es uno de los sistemas de consultas de QuickDB y es con el que se interactúa cuando se invoca al metodo “*obtain*” de AdminBase y se pasa como parámetro el Objeto sobre el que se aplica la consulta, y un String conteniendo la consulta.

El uso del String consiste en realizar la consulta basándose únicamente en las propiedades del Objeto y realizar las comparaciones deseadas refiriéndose únicamente a los atributos del objeto (como si estos fueran públicos).

Ejemplo:

Basándose en las siguientes clases:

```
public class Employee extends Person{

    private int id;
    private int code;
    private String rolDescription;

    //Getters - Setters
}

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 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
}
```


Para obtener el Objeto Employee donde el atributo heredado Address tenga como calle "unnamed street" solo debe hacerse:

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

Lo que equivaldría al hacerlo con SQL a:

```
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'
```

De esta forma se especifica en la cadena los atributos del objeto por los que se quiere realizar la consulta, teniendo en cuenta que se parte desde el nivel de la clase del objeto que se pasa por parámetro, y consultando los atributos heredados (sean del nivel que sean, es decir del Padre directo de la Clase, o del Padre del Padre de la Clase) como si fueran propios.

Como se puede ver en el Ejemplo expuesto, “address” es una referencia a otro Objeto heredada desde la Clase Padre, que contiene el atributo “street”, es por ello que se coloca “address.street”, siendo “address” el nombre del atributo dentro de la Clase “Person”, el cual para el sistema de consultas se interpreta como que es heredado por la Clase “Employee”, y luego se pregunta por el atributo “street” de la Clase “Address” separando la instancia y el atributo por un punto (.) como si se refiriera a atributos públicos.

StringQuery soporta los siguientes operadores:

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

StringQuery ayuda de gran manera a crear consultas complejas de forma muy facil, pero a su vez presenta ciertas limitaciones en cuanto al manejo de fechas, pocos operadores, y tener que especificar toda la sentencia en un String, por lo cual se creo el sistema de consultas que se explicara a continuación.

9.2) QuickDB Query

Este sistema de Consultas realiza la creación de la consulta a través de la invocación de métodos de las Clases “Query”, “Where” y “DateQuery” (para el caso de operaciones con Fechas), lo cual tiene la ventaja de promover la creación de consultas bien formadas, y minimizar el trabajo, ya que se encarga de establecer los JOINS entre las tablas que sean necesarias, etc.

Este sistema de consultas es el que se utiliza al invocar el método “*obtain*” pasando solo como parámetro el objeto sobre el que queremos realizar la operación.

9.2.1) La Clase *Query* nos brinda los siguientes métodos con los que ir armando la consulta:

- **where(String field, [Class classBase]):** El cual comienza la consulta. “Field” corresponde al nombre del campo por el que se desea realizar determinada comprobación, y el atributo “classBase” es opcional, si no se especifica, se asume que el atributo pertenece a la Clase del Objeto que se paso a “*obtain*” como parámetro o a alguna de sus clases Padre, de lo contrario, debe especificarse la Clase a la que pertenece dicho atributo.

Todas las condiciones expresadas entre corchetes [] son opcionales.

- **and(FIELD, [CLASS]):** Concatena la comprobación realizada previamente junto con la comprobación que se comienza con este nuevo atributo (FIELD) mediante “AND”.
- **or(FIELD, [CLASS]):** Concatena la comprobación realizada previamente junto con la comprobación que se comienza con este nuevo atributo (FIELD) mediante “OR”.
- **group(FIELDS, {CLASS}):** Se especifican por cuales campos (FIELDS) se desea agrupar la consulta, colocando los mismos dentro de una cadena (String) separado cada uno por una coma (.). Por cada campo dentro de la cadena se deberá especificar la Clase a los que pertenecen los mismos, a no ser que todos pertenezcan a la clase base de forma directa o por herencia.
- **whereGroup(FIELD, [CLASS]):** Corresponde al “HAVING” de SQL, y especifica una condición para los campos por los que fue agrupada la consulta (requiere haber aplicado previamente una agrupación con “group(...)”).
- **sort(BOOLEAN, FIELDS, {CLASS}):** Ordena el resultado de la Consulta en base a los campos especificados en la cadena (siguiendo la misma metodología expresada previamente de separar cada uno por una coma(.)). Para cada uno de los campos se deberá especificar la Clase a la que pertenecen, a no ser que todos pertenezcan a la clase base de forma directa o por herencia. El valor booleano expresado determinara el tipo de ordenamiento, siendo TRUE de forma ascendente, y FALSE de forma descendente.

9.2.2) La Clase *Where* brinda los siguientes métodos con los que ir armando la consulta:

Los siguientes métodos pueden recibir por parámetro un **Valor** concreto con el que realizar la comprobación, o una **Cadena** con el nombre de un campo y la **Clase** a la que pertenece dicho campo para armar la validación correspondiente en la consulta.

- **equal(OBJECT, [OBJECT])**
- **greater(OBJECT, [OBJECT])**

- **lower(OBJECT, [OBJECT])**
- **equalORgreater(OBJECT, [OBJECT])**
- **equalORlower(OBJECT, [OBJECT])**
- **notEqual(OBJECT, [OBJECT])**

Los siguientes métodos no reciben ningún tipo de parámetro, sino que especifican una condición para el atributo especificado previamente:

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

Otros métodos:

- **between(OBJECT1, OBJECT2):** Este método puede recibir por parámetro cualquier objeto de tipo primitivo, o algún otro objeto que implemente el método “toString()” de manera coherente para consultar si el valor del atributo especificado previo a la llamada de esta operación se encuentra dentro del intervalo que existe entre estos 2 valores (ambos valores deben ser del mismo tipo de dato).
- **in({OBJECTS}):** Este método recibe un conjunto de objetos, con los que validara si el atributo especificado previo a la llamada de esta función, corresponde a alguno de los valores recibidos en esta función por parámetro.
- **match(EXPRESION):** Este método recibe una Cadena por parámetro y evalúa si el atributo especificado previo a la llamada de esta función contiene a la cadena recibida. Si se recibe una cadena simple, se comprobara si el atributo corresponde a una cadena del tipo: “**EXPRESION**” (siendo ** cualquier tipo de cadena), por el contrario si en la EXPRESION se utiliza alguno de los caracteres “%” (que representa cualquier conjunto de caracteres) o “_” (que representa un carácter cualquier), se validara por la EXPRESION exacta recibida sin alterar el principio y el final.
- **date():** Con este método, se especifica que el atributo ingresado previamente corresponde a un atributo del tipo Fecha, por lo que devuelve una referencia a “DateQuery” para poder realizar las comprobaciones de Fecha que sean necesarias.

9.2.3) La Clase DateQuery brinda los siguientes métodos con los que ir armando la consulta:

- **differenceWith(VALUE, [CLASS]):** Este método permite determinar la diferencia en días entre el atributo expresado previamente y el valor que se pasa por parámetro u otro campo de alguna tabla de la Base de Datos (si se especifica el Campo y la Clase).
- **month():** Devuelve el valor del mes del atributo ingresado previamente.
- **day():** Devuelve el valor del día del atributo ingresado previamente.
- **year():** Devuelve el valor del año del atributo ingresado previamente.

Luego estos valores devueltos deben ser comparados utilizando alguno de los métodos de la Clase “Where”.

9.2.4) Realizando la Búsqueda

Una vez concatenado los métodos para formar la consulta, solo queda la “ejecución” de la misma, y existen 2 métodos para realizarlo, agregando como ultimo método en la concatenación:

- **find():** el cual completa con los datos resultantes el objeto pasado por parámetro a “*obtain*”.
- **findAll():** el cual retorna una colección con los objetos resultantes de ejecutar dicha consulta.

Ejemplos:

En base al siguiente Modelo de Datos:

```
public class UserQuery extends UserParent{

    private int id;
    private String name;

}

public class UserParent {

    private int id;
    private String description;
    private ReferenceQuery reference;

}

public class ReferenceQuery extends ReferenceParent{

    private int id;
    private String value;

}

public class ReferenceParent {

    private int id;
    private String valueParent;

}
```

```

public class CompleteQuery {

    private int id;
    private String name;
    private double salary;
    private int age;
    private Date birth;
    private boolean cond;

}

```

Consultas:

- Consulta Simple:

```

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

```

- Consulta Simple de Atributo Heredado:

```

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

```

- Consulta en base al valor del atributo de una Referencia:

```

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

```

- Consulta en base al valor del atributo heredado de una Referencia:

```

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

```

- Consulta utilizando “between”, “and” y “lower”

```

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

```

- Consulta utilizando “in”, “or” y “match”

```

ArrayList array = admin.obtain(query).where("age").in(22, 23, 24, 25).
    or("name").match("sarmentero").findAll();

```

- Consulta utilizando “group”, “whereGroup” y “greater”

```

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

```

- Consulta utilizando “differenceWidth”, “equal”

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

10 Vistas

Las Vistas en QuickDB se utilizan para crear representaciones mas simples de Estructuras de Datos mas complejas, como pueden ser Objetos compuestos por diversos Objetos, etc.

Una Vista brinda la posibilidad de realizar la obtención de los datos mas rápidamente, siendo que no se tiene que procesar jerárquicamente todos los Objetos involucrados, sino que se asignaran los valores especificados directamente a los atributos de la Vista disminuyendo de gran manera el tiempo de procesamiento.

Para armar una Vista es necesario crear una Clase que extienda de “View”, la cual especifica un metodo denominado “**query()**” el cual debe que ser implementado por la Clase Hija.

Este metodo “**query()**” retorna un dato de tipo “Object” el que bien puede ser un String conteniendo en una cadena la especificación de la consulta, o puede ser un Objeto del Tipo “QuickDB Query” permitiendo armar la consulta con las facilidades de este sistema de consultas.

Existen otros 3 métodos los cuales su implementación es opcional, dependiendo del tipo de Vista que se quiera generar:

- **renameColumns():** Se especifica en una cadena los nuevos nombres que tomara cada uno de los campos resultantes de la consulta. Estos nombres seran los que se utilizaran para obtener los valores y asignarlos a los atributos especificos dentro de la Vista. Si este método no se implementa, se toma por defecto como nuevos nombres los nombres de cada uno de los Atributos de la Vista en el orden en que figuran.
- **columns():** Se especifica en una Cadena cuales atributos se tendran en cuenta en la consulta, si no se implementa este método, se toma como los nombres de cada uno de los Atributos de la Vista en el orden en que figuran.
- **classes():** Especifica en un Array del tipo Class[] las clases a las que pertenecen cada uno de los atributos especificados en “columns()”. Si este método no se implementa, se asume que todos los atributos son de la Clase Base especificada en la consulta.

10.1) Ejemplos

En base al Modelo de Datos siguiente:

```
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;

        }
    }

```

Las Vistas creadas son las siguientes:

1. Haciendo uso del Sistema de consultas “QuickDB Query”, y reescribiendo todos los métodos de configuración de 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("accountTest");
        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};
    }

}

```


2. Haciendo uso de la especificación de la consulta en una Cadena de texto

```
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 = 'accountTest'";
    }
}
```

3. Haciendo uso del Sistema de consultas “QuickDB Query”, sin implementar el método de renombrado de las columnas resultantes.

```
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 dateView;

    @Override
    public Object query(){
        ObjectViewTest1 o1 = new ObjectViewTest1();
        Query query = Query.create(this.getAdminBase(), o1);
        query.where("account", ObjectViewTest1.class).equal("accountTest");
        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) Creando una Instancia de la Vista

Existen 2 formas de inicializar una Vista, la primera consiste en ejecutar el método “**initializeViews(...)**” de “*AdminBase*” el cual inicializara todas las instancias creadas o por crearse del Modelo de Datos. Y la otra forma es pasarle a la Vista una instancia activa de AdminBase:

```
AdminBase.initializeViews(AdminBase.DATABASE.MYSQL,
    "localhost", "3306", "testQuickDB", "root", "");

ViewObject view = new ViewObject();
view.initializeAdminBase(admin);
```

10.1.2) Obteniendo los Valores de la Vista

Una vez creada la instancia de la Vista, solo es necesario utilizar los métodos “**obtain()**” u “**obtainAll()**” para obtener la/s Vista/s resultantes en base a los objetos de la Base de Datos:

```
view.obtain();

ArrayList array = view.obtainAll();
```

10.1.3) Consultas Dinámicas en una Vista

Una vista es especificada en base a una consulta que la representa, pero si se quisiera en tiempo de ejecución alterar esa consulta para agregar alguna condición mas para que las Vistas obtenidas sean particulares a una situación, se puede recurrir a la utilización del método “**dynamicQuery**” de la Vista, el cual tanto como se vio en el método “**query()**” se puede trabajar mediante String o mediante la utilización del sistemas de consultas de QuickDB, y lo que hace es concatenar a la consulta que representa la Vista un nuevo conjunto de condiciones a evaluar.

Ejemplo:

- Utilizando String:

```
ViewObjectString view = new ViewObjectString();
view.initializeAdminBase(admin);
view.dynamicQuery("AND name LIKE '%Dynamic%'");
view.obtain();
```

- Utilizando QuickDB Query:

```
ViewObject view = new ViewObject();
view.initializeAdminBase(admin);
view.dynamicQuery( ((Query)view.query()).and("name").match("Dynamic") );
view.obtain();
```