



# Language Integrated Queries For Object Relational Mapping

Version 1.0.0-rc1

Eric Bottard, Vincent Cornet

---

# **LIQUidFORM: *Language Integrated Queries For Object Relational Mapping***

by Eric Bottard and Vincent Cornet

1.0.0-rc1

Copies of this document may be made for your own use and for distribution to others, provided that you do not charge any fee for such copies and further provided that each copy contains this Copyright Notice, whether distributed in print or electronically.

---

---

---

<b>1. Introduction</b>	1
1.1. Background	1
1.1.1. Refactoring went mainstream	1
1.1.2. JPA still uses Strings	1
1.1.3. Enter LIQUidFORM	2
1.1.4. But wait ! There is more !	2
1.2. Goals and Principles	3
1.2.1. Stuck in Java Land	3
1.2.2. Early feedback at all costs	3
I. LIQUidFORM by Example	7
<b>2. Requirements</b>	9
2.1. Getting those dependencies	9
<b>3. Download using Maven</b>	11
3.1. Add a reference to the project repository	11
3.2. Add a dependency to the library	11
<b>4. Anatomy of a Query</b>	13
4.1. General Syntax of a Query	13
4.2. The Query Domain and Identification Variables	13
<b>5. Simple Queries</b>	17
5.1. Selecting Entities	17
5.2. Selecting Attributes and Associations	18
5.3. Going Further...	19
<b>6. Restricting the results</b>	21
6.1. Conditional Expressions	21
6.2. Using Parameters	21
6.3. Composition	22
6.4. Going Further...	23
<b>7. Grouping and Ordering</b>	25
7.1. Aggregating with the <code>GROUP BY</code> construct	25
7.1.1. Restrictions on groups	25
7.2. Ordering the Results	26
<b>8. LIQUidFORM in 4 easy steps</b>	27
II. LIQUidFORM Reference	29
<b>9. The <code>SELECT</code> clause</b>	31
<b>10. The <code>FROM</code> clause</b>	33
10.1. Aliasing	33
<b>11. The <code>WHERE</code> clause</b>	35
<b>12. The <code>GROUP BY</code> clause</b>	37
<b>13. The <code>HAVING</code> clause</b>	39
<b>14. The <code>ORDER BY</code> clause</b>	41

---

# Chapter 1. Introduction

LIQUidFORM stands for *Language Integrated QUeries For Object Relational Mapping* and is a Java library that provides a Domain Specific Language for building type-safe and refactoring proof JPA queries. While the project draws its inspiration from the LINQ project, it is not LINQ for Java : its aim is only to help in writting JPA queries against your domain model. Read on to understand what LIQUidFORM is and what it is *not*.

## 1.1. Background

The agility principles have made tremendous improvements in recent years thanks not only to good communication and marketing but also to evolved tooling. Agilists recommend that you refactor your code should you be unhappy with it. The pragmatic programmers recommend that you don't live with broken windows. We recommend you follow their advice !

### 1.1.1. Refactoring went mainstream

What was painful less than five years ago is now a piece of cake with modern IDEs. Using eclipse, refactoring a type can be done inline with a few keystrokes (**ALT+SHIFT+R** followed by **ENTER**) : if invoked on a java type, every occurrence of that type will be changed to the new type. If invoked on a method or field name, every occurrence will be changed accordingly. Note that this operation is not a trivial textual search/replace operation, as you could have performed with tools at the beginning of the century. No more excuses for not correcting that silly typo on `Customer.getCuontry()`. Time to rename `Client.getAddress2()` that no one understands to `Client.getBillingAddress()`. Isn't life beautiful ?

### 1.1.2. JPA still uses Strings

Well, it's not *that* beautiful. If you have ever written a JPA Query, like so

```
1 List people = em.createQuery(  
2     "SELECT FROM Person p WHERE p.surname LIKE 'Smith%'"  
3     .getResultList();
```

then you know where the problem lies. On line 2, when you specify the query to execute, you're using a simple `java.lang.String`. The problem with this is that your IDE doesn't have a clue about what the String contains, and is not aware of the class `Person`, nor is it of the `surname` property.

What this means is that if you happen to refactor `getSurname()` to let's say `getLastName()`, your query becomes invalid, but your IDE didn't help here.

Hopefully, you'll catch the error soon enough with unit tests, but isn't it sad that this isn't done automatically ?

### 1.1.3. Enter LIQUidFORM

LIQUidFORM attempts to solve just that (and a little more). Its aim is to capture your intent when creating the query, so that when you change `Person.getSurname()` to `Person.getLastName()`, the query string gets updated as well. Here is how it goes :

```
Person p = LiquidForm.use(Person.class, "p");
List people = em.createQuery(
    select(p).from(Person.class).as(p).where(
        like(p.getSurname(), "Smith%")
    ).toString()
).getResultList();
```

As you can see, the framework offers a syntax that is very similar to the JPA query syntax. As a matter of fact, LIQUidFORM does very little : it just constructs a String representation of your query, just like you would by hand, that is in turn passed to your JPA engine of choice.

But because in that construct `p.getSurname()` is an actual method call to the method named `getSurname` on the type `Person`, it *will* get refactored when you change `surname` to `lastName`. Of course, you also get other direct indications from your IDE when you perform other kinds of refactorings. For exemple, if you happen to delete the `Person` type, you'll be granted by a compile time error for sure. Something that would have gone unnoticed otherwise !

### 1.1.4. But wait ! There is more !

Using LIQUidFORM also yields additional benefits : because your queries are now built using actual java code, you get strong type safety. This manifests itself eg. when constructing `WHERE` clauses :

```
// Won't compile if Person.age is a number
select(p).from(Person.class).as(p).where(like(p.getAge(), "some string"))
```

This support is not perfect<sup>1</sup>, but that's still better than the plain string syntax that you're used to...

---

<sup>1</sup>because of the way generics work, there are some corner cases in which method signatures are laxer than they should be. See Part II, "LIQUidFORM Reference" for details.

## 1.2. Goals and Principles

### 1.2.1. Stuck in Java Land

As stated earlier, LIQUidFORM is an *internal Domain Specific Language (or DSL)* whose goal is to bring compile time safety and ease refactorings of code written in Java. There are new languages for the Java Virtual Machine (such as Groovy<sup>2</sup>) which are sexy and all, but for people stuck with plain old Java, LIQUidFORM tries to bring value added with the tools at hand (see the discussion about The benefits of Java 5 (p. 9)).

#### Internal vs. External DSLs

According to Martin Fowler <sup>3</sup>:

**Internal DSLs** are particular ways of using a host language to give the host language the feel of a particular language. This approach has recently been popularized by the Ruby community although it's had a long heritage in other languages - in particular Lisp. Although it's usually easier in low-ceremony languages like that, you can do effective internal DSLs in more mainstream languages like Java and C#. Internal DSLs are also referred to as embedded DSLs or FluentInterfaces

**External DSLs** have their own custom syntax and you write a full parser to process them. There is a very strong tradition of doing this in the Unix community. Many XML configurations have ended up as external DSLs, although XML's syntax is badly suited to this purpose.

### 1.2.2. Early feedback at all costs

Although things are slowly changing for the better, Java development (and especially server-side development) is plagued with the three phased *Compile-Deploy-Test* cycle. Indeed, there are three steps when doing server-side coding :

#### Compile phase

This early stage happens when you as a developer write Java code and, for the sake of discussion, JPA queries. You benefit from the help of your IDE of choice, be it incremental building, early red squiggles under errors or content-assist.

<sup>2</sup><http://groovy.codehaus.org>

<sup>3</sup><http://www.martinfowler.com/bliki/DomainSpecificLanguage.html>

### Deploy phase

This peculiar phase makes more sense for enterprise level development. It happens when you start or restart your application server, or simply redeploy your application. The different engines involved in your system (these could be the servlet container, or a Spring `ApplicationContext` or your JPA implementation of choice) are likely to read some configuration. At that time, you may be granted with errors because of eg. syntax errors in a XML file.

### Runtime phase

The runtime phase is identified as the instant when you actually exercise the code you just wrote. Errors may arise because the parameters you pass in are inconsistent, or the configuration may be wrong as in the previous case, but you *have to* trigger the use case (this may involve long set up phase for complex use cases) to diagnose it.

The philosophy of the LIQUidFORM library is to provide feedback as soon as possible *ie.* to help in the *compile-phase* rather than in the *runtime-phase*. Moreover, we believe that it is not LIQUidFORM's role to tell you about errors that your JPA engine will diagnose. A concrete example of this is that it is rather complex to know if a query is syntactically correct and the JPA engine knows better than us. So what LIQUidFORM provides is the best support possible at compile-time, but does not try to help you at runtime (because if your query is malformed, the JPA engine will tell you anyway).

So what does it mean to provide the best support at compile-time ? Well, that means that we tried to be as specific as was humanly possible in our API design, so that you can't use stuff that you're not supposed to. This is particularly true about type safety.

Take for example the various comparison tests available in the `WHERE` clause (see Chapter 11, *The `WHERE` clause* for a complete description of those conditions).

```
Person p = use(Person.class, "p");
Person p2 = use(Person.class, "p2");

// select p from Person as p where p.age in
//   (select p2.age from Person as p2 where p2.name like 'J%')
select(p).from(Person.class).as(p).where(
    in(p.getAge(), select(p2.getAge()).from(Person.class).as(p2)
        .where(like(p2.getName(), "J%"))))
).toString()
```

In this example, the subquery is strongly typed as returning `Integers` and thus, the `eq()` statement would only accept `Numbers` for the comparison (as a matter of fact `p.getAge()` is of the correct type).



This level of support sometimes involved hard work on our side, because of the way generic types work. There are for example tens of hand-written variations of the `eq()` method, so that the most precise bounds on argument types are accepted. We believe this was worth the burden.

---

# Part I. LIQUidFORM by Example

Writing LIQUidFORM was a fun adventure, yet we had to cope with all the subtleties of the JPA query syntax. This allowed us to (re-)discover that API under a new light, but if you're already familiar with JPA queries, or even SQL, then this first part will show you how to express your queries using LIQUidFORM constructs by walking you around everyday examples.

If on the contrary, you're looking for a complete reference, or don't understand why some constructs don't behave like you'd expect, then please refer to Part II, "LIQUidFORM Reference".

---

## Chapter 2. Requirements

This section highlights the software requirements for using LIQUidFORM. This is mainly an explanation of why the dependencies are required, since installation should really not be a bother, whatever the tools you use.

- LIQUidFORM/JPA as its name implies, helps in building queries that conform to the JPA Query Language syntax. As such, LIQUidFORM is meant to be used in a JAVA5+ environment and indeed uses many features of Java 5.

### The benefits of Java 5

LIQUidFORM leverages many features introduced with Java 5 :

- Generics are central to LIQUidFORM and allow strong type-safety when building queries against your model
  - Static imports allow you to use the library in a way that is readable, a key feature of a Domain Specific Language. See Improving Readability (p. 19) for more information about this.
  - Varargs are used here and there, eg. when building a `select(...)` construct
  - Covariant return types, along with generics, help in achieving type-safety
  - Annotations on your domain model tell LIQUidFORM about customizations or exceptions to naming conventions you've made
- LIQUidFORM also has a dependency on JPA annotations (package `javax.persistence`) which are expected to already be present on your development classpath since you're using JPA
  - Finally, LIQUidFORM works by creating so-called dynamic proxies of your domain classes. This functionality is provided by the CGLib library, which itself relies on ASM, a bytecode manipulation library. LIQUidFORM is compiled against `cglib-nopdep`, a version of CGLib that already includes ASM re-packaged as `net.sf.cglib` sub-packages, so you don't need to know about it.

### 2.1. Getting those dependencies

- If you're using Maven<sup>1</sup>, the aforementioned dependencies are already handled by Maven.

- If you're not using Maven, those libraries are available with the software distribution of LIQUidFORM, in the `lib/` directory.

Please note that those libraries are subject to their respective licenses. LIQUidFORM itself is licensed under the Apache Software License, v2.<sup>2</sup>

---

<sup>2</sup><http://www.apache.org/licenses/LICENSE-2.0>

---

# Chapter 3. Download using Maven

## 3.1. Add a reference to the project repository

If you're using maven, you can access releases from the following repository : <http://liquidform.google.com/svn/maven-repo/releases>. To use this repository, add the following to your project POM :

```
...
<repositories>
  <repository>
    <id>liquidform-repo-releases</id>
    <name>Maven Repository for LIQUidFORM (releases)</name>
    <url>http://liquidform.googlecode.com/svn/maven-repo/releases</url>
  </repository>
</repositories>
```

Snapshots may also be available from <http://liquidform.googlecode.com/svn/maven-repo/snapshots>.

```
...
<repositories>
  <repository>
    <id>liquidform-repo-snapshots</id>
    <name>Maven Repository for LIQUidFORM (snapshots)</name>
    <url>http://liquidform.googlecode.com/svn/maven-repo/snapshots</url>
  </repository>
</repositories>
```

## 3.2. Add a dependency to the library

Then, simply add a dependency, in your project's `pom.xml`, to `groupId=com.google.code.liquidform` and `artifactId=liquidform.jpa` using the appropriate version (1.0.0-rc1 at the time of writing) :

```
...
<dependencies>
...
  <dependency>
    <groupId>com.google.code.liquidform</groupId>
    <artifactId>liquidform.jpa</artifactId>
    <version>1.0.0-rc1</version>
  </dependency>
...

```

---

# Chapter 4. Anatomy of a Query

## 4.1. General Syntax of a Query

Before diving into the core of LIQUidFORM syntax, let's have a look at a random JPA query, so that we set common ground for the explanations that follow :

```
select o.quantity, a.zipcode (1)
from Customer as c (2)
inner join c.orders as o inner join c.address as a (3)
where a.state = 'CA' (4)
order by o.quantity asc, a.zipcode asc (5)
```

- (1) the `SELECT` clause lists the elements that you want returned by your query. Those can be scalar values, entities or aggregates over some quantities.
- (2) the `FROM` part of the query tells the engine about the entities that you *may* be interested in. It lists the candidate entities out of which you may select return values.
- (3) inside the `FROM` clause may appear *joins* which match entities on some properties. The behavior to adopt when the matching values are absent is given by the type of `JOIN`.
- (4) amongst all the matching entities, you may be interested by only a few. The `WHERE` clause allows you to specify conditions that entities must satisfy to be retained.
- (5) additionally, you may order the returned results according to some properties with the `ORDER BY` clause.

There are also the `GROUP BY` and the `HAVING` clauses which are discussed in Section 7.1, “Aggregating with the `GROUP BY` construct”. But an interesting fact appears here, which explains why many specifications and languages put the `FROM` clause ahead of the rest : you need to know the initial set of objects out of which you *may* select your query result. This initial set is called the *query domain* and is discussed in the following section.

## 4.2. The Query Domain and Identification Variables

The `FROM` clause defines the domain of the query by declaring *identification variables*. Those are the identifiers that appear after the optional `AS` keyword. Those identifiers will then be used throughout the query to specify conditions or to select only some properties.

The promise of LIQUidFORM is to use the full power of Java when declaring those identifiers so that

1. they bear strong type information, detecting incompatibilities as early as possible,

2. references to properties is made using actual method calls so that they get refactored should you decide to change their names.

Hence, you need to declare one local Java variable for each identification variable that will appear in the `FROM` clause, using the `LiquidForm.alias()` construct :

```
Person p = LiquidForm.alias(Person.class, "p");
```



### Note

This will allow you to reference the `p` variable later in your queries (more on this later) :

```
select(p.getName()).from(Person.class).as(p);
```

In the previous example, the identification variable `p` would thus designate every instance of the `Person` entity. This is because we used the variant of the `LiquidForm.alias()` method that accepts a `Class<T>` object as its first argument, and we chose to pass in the `Person.class` class. Note that the benefits of type safety appear this early in our usage of `LIQIdFORM`. The following code is invalid and will not compile :

```
Person p = (1)
    LiquidForm.alias(Address.class, "p"); (2)
```

- (1) here we declare a variable of type `Person`
- (2) this will actually return an object of type `Address`, which is incompatible with the `Person` type.

Apart from the `LiquidForm.alias()` method signature that accepts a `Class<T>` object, you may encounter other variants. These won't be described in detail here, but know that you can (and must) use an `alias()` method to declare other types of identification variables. An example with collections follows, but a complete reference can be found in Section 10.1, "Aliasing".

```
Order o = LiquidForm.alias(Order.class, "o"); (1)
LineItem l = LiquidForm.alias(o.getLineItems(), "l"); (2)
```

- (1) here a variable is declared over the `Order` entity, as seen earlier
- (2) the `l` variable is declared as being an element of the `o.lineItems` collection. Note that the actual type of `l` is `LineItem` and *not* `Collection<LineItem>` !

**What's with those two names ?**

You may have noticed in the previous examples that we give a name to an identification variable in our Java source. But there is also that second method parameter that always takes that same "value". How come ?

The thing is that the Java identifier name is known only to the Java compiler. The LIQUidFORM library thus needs a way to know about that variable name, and that second String parameter is the way to do it. Of course, it would be technically possible to use a different name, like such

```
Person person = LiquidFrom.alias(Person.class, "bogus");
```

but we strongly discourage it.

That `String` parameter given to the method is the one that will be used in the generated query text (again, LIQUidFORM has no idea of the Java identifier that you used.) So, to be 100% clear : if you were to use the example above to produce a query, you would see "bogus" appear and *not* "person".



---

# Chapter 5. Simple Queries

## 5.1. Selecting Entities

Now that we know how to declare Java identifiers for our identification variables, we are ready to write our first queries. Before we begin, let us assume the following domain model :

```
@Entity
public class Customer {

    private String firstname;

    private String lastname;

    private Set<Order> orders;

    public String getFirstname() {
        return firstname;
    }

    public String getLastname() {
        return lastname;
    }

    @OneToMany(mappedBy="customer")
    public Set<Order> getOrders() {
        return orders;
    }

    // setters omitted for brevity
}

@Entity
public class Order {

    private BigDecimal totalcost;

    private Customer customer;

    public BigDecimal getTotalcost() {
        return totalcost;
    }

    @ManyToOne
    public Customer getCustomer() {
        return customer;
    }

    // ..
}
```

Let's start with the simplest of all : selecting all entities of a certain type :

```
Order o = LiquidForm.alias(Order.class, "o"); (1)
String q = LiquidForm.select(o).from(Order.class).as(o).toString(); (2)
assertEquals("select o from Order as o", q); (3)
```

- (1) this is a simple *identification variable* declaration over the `Order` schema type, as seen in Section 4.2, “The Query Domain and Identification Variables”
- (2) we use `LIQUidFORM` to produce a query and assign its `String` representation to the `q` variable
- (3) we verify the produced query against the expected value



### About the examples

Throughout this tutorial, the generated queries are tested against their expected value using JUnit-like assertions (as on line 3 in the previous example). We assume that this syntax is comprehensible and has the benefits of not dictating how to actually use the framework. Indeed, you could decide to create a query everytime you want to run it, or you could decide to create them once and for all in a static initializer. The latter is our preferred way of doing, but your mileage may vary.

So we made it ! We built our first query using `LIQUidFORM`. As you can see, the grammar is really fluid and follows the native JPA syntax as much as it can. Let us review the actual query creation on line 2 of the previous example. We'll break it up in three parts and discuss them afterwards :

```
1 LiquidForm.select(o) (1)
2   .from(Order.class).as(o) (2)
3   .toString(); (3)
4
```

- (1) every query creation should begin with a call to `LiquidForm.select(Object...)`, which is a static method of the `LiquidForm` class.
- (2) the previous call returned some object<sup>1</sup> that is then used to chain calls to other methods. The set of possibilities is here restricted to some variant of the `from()` method.
- (3) this pattern goes on until you have completely built your desired query, at what time you should call the traditional `toString()` method to get a `String` representation of the query.

## 5.2. Selecting Attributes and Associations

Of course, you're not restricted to selecting the identification variable that is bound in your query. Nor are you required to select only one element. If you

want to select an attribute of your domain model, simply reference it using a call to the corresponding *getter* on the identification variable. Here is an example :

```
Customer c = LiquidForm.alias(Customer.class, "c");
String q = LiquidForm.select(c.getFirstname(), c.getLastname())
    .from(Customer.class).as(c).toString();
assertEquals("select c.firstname, c.lastname from Customer as c", q);
```

It is of course possible to navigate associations when selecting or binding to identification variables, as is demonstrated in the following example :

```
Order o = LiquidForm.alias(Order.class, "o");
String q =
    LiquidForm.select(o.getCustomer()).from(Order.class).as(o).toString();
assertEquals("select o.customer from Order as o", q);
```



## Improving Readability

You may have noticed that in all previous examples, we explicitly wrote calls to the static method `LiquidForm.select(Object...)` exactly just like that. But because it's a static method and we're using Java 5, we can of course use *static imports* (as explained in "The benefits of Java 5"). This will be particularly elegant as soon as we introduce "Conditional Expressions" :

```
import static com.google.code.liquidform.LiquidForm.*;
...
Order o = alias(Order.class, "o");
String q = select(o).from(Order.class).as(o).toString();
```

From now on, we will assume that you imported every static member of `com.google.code.liquidform.LiquidForm`. There are also other utility classes in that package (namely `Aggregates`, `Functions` and `Parameters`) but they will be discussed in greater length later.

## 5.3. Going Further...

There are *many* more possibilities when playing with the `SELECT` and `FROM` clauses, but those are discussed in Part II, "LIQUidFORM Reference".

For the time being, you may be interested to know that one can select aggregates (such as `max` or `avg`) by using other statics of the `LiquidForm` class :

```
Order o = alias(Order.class, "o");
String q = select(avg(o.getPrice())).from(Order.class).as(o);
assertEquals("select avg(o.price) from Order as o", q);
```

Regarding the `FROM` clause, all forms of joins are supported using `andFrom()`, `innerJoin()`, `leftOuterJoin()` and even the *fetch* variants. Here are some examples :

```
// Inner joins
Customer c = alias(Customer.class, "c");
Order o = alias(c.getOrders(), "o");
String q = select(c).from(Customer.class).as(c)
    .innerJoin(c.getOrders()).as(o).toString();
assertEquals("select c from Customer as c inner join c.orders as o", q);
```

```
// Theta joins
Customer c = alias(Customer.class, "c");
Order o = alias(Order.class, "o");
String q = select(c.getFirstname(), o.getTotalcost())
    .from(Customer.class).as(c)
    .andFrom(Order.class).as(o).toString();
assertEquals("select c.firstname, o.totalcost from Customer as c, Order as o", q);
```

---

# Chapter 6. Restricting the results

## 6.1. Conditional Expressions

So far, we only have covered the `SELECT` and `FROM` clauses, and their LIQUIFORM equivalent. But constructing queries without a `WHERE` clause is not very useful. This chapter explains how to do this with the library.

When writing a query with a `WHERE` clause, the expression that comes in that clause is a *condition* that will be evaluated for every tuple. Only the tuples that matched the criteria are actually selected. As such, LIQUIFORM supports a `WHERE` construct that follows exactly that paradigm : you provide it with an instance of `ConditionalExpression`. Hopefully, such objects are constructed with static methods of the `LiquidForm` class, so that the final "sentence" reads nicely :

```
Customer c = alias(Customer.class, "c");
String q = select(c).from(Customer.class).as(c)
    .where(eq(c.getFirstname(), "John"));    (1)
assertEquals("select c from Customer as c where c.firstname = 'John'", q);

Order o = alias(Order.class, "o");
String q2 = select(o).from(Order.class).as(o)
    .where(gte(o.getTotalcost(), 1000)).toString();    (2)
assertEquals("select o from Order as o where o.totalcost >= 1000", q2);
```

- (1) `eq` stands for "equals"
- (2) `gte` stands for "greater than or equal"

As you can see, every comparison operator is implemented as a static method available on the `LiquidForm` class. Think of them as a prefixed notation of the traditional binary operators (there are also equivalents of unary and ternary operators, such as `isEmpty(Collection)` OR `between(Number, Number, Number)` respectively).

An important aspect of those operators is that they are strongly typed. You can compare the `BigDecimal` property `Order.totalcost` with a literal `int` because they both resolve as `Numbers`. Similarly, you can compare `Customer.firstname` with a literal `String` because they're both `Strings` (obviously). Change one of them to an incompatible type and you'll get a *compilation* error.

## 6.2. Using Parameters

You may have noticed in the previous examples that the `String` literal `"John"` was properly escaped as a JPA Query literal (using single quotes) because

that's what it is to the engine's eyes : a *literal*. Of course, using a String Java variable won't change anything to that :

```
Customer c = alias(Customer.class, "c");
String n = "John";
// Will still produce [select c from Customer as c where c.lastname =
'John']
select(c).from(Customer.class).as(c)
    .where(eq(c.getLastname(), n)).toString();
```

If what you actually wanted to achieve is the use of a query parameter (that you will have to provide with an actual value by the time the query is *executed*), then you can use members of the `Parameters` class :

```
import static com.google.code.liquidform.LiquidForm.*;
import static com.google.code.liquidform.Parameters.*;

Customer c = alias(Customer.class, "c");
String q = select(c).from(Customer.class).as(c)
    .where(eq(c.getFirstname(), param(1, String.class))).toString();
assertEquals("select c from Customer as c where c.firstname = ?1", q);

// When you actually want to use your query, bind a value
// to that parameter. Note that this is plain JPA and has
// nothing to do with LIQUIDFORM :
List johns = em.createQuery(q).setParameter(1, "John").list();
```

As you can see, the type of the parameter has to be provided, which is a bit cumbersome. But again, if you later change the type of `firstname` to something other than `String`, you'll get a compile-time error.



### Note

There is also a variant for *named* parameters. See Chapter 11, *The WHERE clause*.

## 6.3. Composition

Specifying that one of the properties of your domain model should match some condition is nice, but often your conditions are more complex than a simple comparison.

Typically, you would want to *compose* atomic conditions together using boolean logic. In traditional JPA Query syntax, this is done with the `AND`, `OR`, `NOT` keywords and parentheses. In LIQUIDFORM land, you can simply chain conditions using `and(ConditionalExpression)` and `or(ConditionalExpression)` :

```
Order o = alias(Order.class, "o");
String q = select(o).from(Order.class).as(o)
    .where(gt(o.getTotalprice(), 100).and(lt(o.getTotalprice(), 1000)))
```

```
.toString();
assertEquals("select o from Order as o where o.totalprice > 100 and
o.totalprice < 1000", q);
```

Negating is done with the static `not(ConditionalExpression)` method, while grouping inside parentheses is accomplished with the static `paren(ConditionalExpression)` construct. Here is an example showing both :

```
Order o = alias(Order.class, "o");
String q = select(o).from(Order.class).as(o)
    .where(not(gt(o.getTotalprice(), 100))
        .and(paren(eq(o.getCustomer().getFirstname(), "John")
            .or(eq(o.getCustomer().getFirstname(), "Jane")))))
    .toString();
assertEquals("select o from Order as o where not o.totalprice > 100 and
(o.customer.firstname = 'John' or o.customer.firstname = 'Jane')", q);
```



### If expressions get too complex...

As you can see, expressions can quickly become complex with nested parentheses getting out of hand. If you need to break out your code, we recommend extracting the `ConditionalExpression`, even separating it into several boolean clauses if necessary. Thus, it will be easy to give them a meaningful name describing what they test (such as `nameCondition` for example).

Although the initial goal of LIQUidFORM is to read like a DSL, we acknowledge that parentheses are a necessary evil around method invocations in Java. You should always put readability first in your programs. In that respect, the "Extract local variable" refactoring offered by many IDEs (**ALT+SHIFT+L** in eclipse) is very useful, as is the "Expand current selection" command (**ALT+SHIFT+↑**).

## 6.4. Going Further...

In this chapter, we only scratched the surface about the `WHERE` clause. Be aware that there are so many more features : every conditional expression has a LIQUidFORM equivalent, be it about collections, nullity checks or string operations. See Chapter 11, *The WHERE clause* for further details.

There is even support for all the JPA standard functions in the `WHERE` clause, like `SQRT()` or `CURRENT_DATE`. An example is given below :

```
import static com.google.code.liquidform.Functions.*;
import static com.google.code.liquidform.LiquidForm.*;
```

```
Customer c = alias(Customer.class, "c");
String q = select(c).from(Customer.class).as(c)
    .where(gt(length(c.getFirstname()), 4))
    .toString();
assertEquals("select c from Customer as c where length(c.firstname) > 4",
    q);
```

Last but not least (as we are particularly proud of the typesafety feature of them), sub-queries are supported in LIQUIFORM :

```
import static com.google.code.liquidform.LiquidForm.*;
import static com.google.code.liquidform.Aggregates.*;

Customer c = alias(Customer.class, "c");
Order o = alias(c.getOrders(), "o");
String q = select(c).from(Customer.class).as(c)
    .where(gt(select(count(o)).from(c.getOrders()).as(o), 10))    (1)
    .toString();
assertEquals("select c from Customer as c where (select count(o) from
    c.orders as o) > 10", q);
```

- (1) the `select(count(o)).from(c.getOrder()).as(o)` bit is a query in its own right and LIQUIFORM knows that it returns a `Long`. Hence, it is eligible for a comparison to an `int`.



---

# Chapter 7. Grouping and Ordering

## 7.1. Aggregating with the `GROUP BY` construct

By now, you should have a fair understanding of how LIQUidFORM works. This last chapter deals with the grouping functionality offered by the `GROUP BY` clause.

You can group according to properties or whole entities (provided that the entity does not contain `Serializable` nor LOB-valued properties). Hence, the `groupBy()` method accepts multiple `Objects` as parameters :

```
import static com.google.code.liquidform.Aggregates.*;
import static com.google.code.liquidform.LiquidForm.*;

Order o = alias(Order.class, "o");
String q = select(o.getCustomer(), count(o))
    .from(Order.class).as(o)
    .groupBy(o.getCustomer()).toString();
assertEquals("select o.customer, count(o) from Order as o group by o.customer", q);
```

### 7.1.1. Restrictions on groups

You can also apply conditions on the *grouped* results using the `HAVING` clause. That clause is similar to the `WHERE` clause, but applies to the groups, as built by the `GROUP BY` construct.

Being very similar, it too accepts a `ConditionalExpression` as parameter. Indeed, if you read Chapter 6, *Restricting the results*, you already know how to use the `having()` method :

```
import static com.google.code.liquidform.Aggregates.*;
import static com.google.code.liquidform.LiquidForm.*;

Order o = alias(Order.class, "o");
String q = select(o.getCustomer(), sum(o.getTotalprice()))
    .from(Order.class).as(o)
    .groupBy(o.getCustomer())
    .having(gt(o.getTotalprice(), 1000)).toString();    (1)
assertEquals("select o.customer, sum(o.totalprice) from Order as o group by o.customer having o.totalprice > 1000", q);
```

- (1) as you can see, the same static methods used to construct conditions applicable to the `WHERE` clause can be used to construct a condition for the `HAVING` clause.

Of course, the same rules that applied to conditions for the `WHERE` clause still hold true : you can compose atomic expressions using boolean logic, use `Functions` or input parameters, *etc.*

## 7.2. Ordering the Results

Let us end this tutorial with support for results ordering using the `ORDER BY` clause. You can order the results by any property provided that it satisfies the following conditions :

1. it is naturally *comparable* (*ie.* `NumberS`, `DateS`, *etc.*)
2. it appears in the `SELECT` clause, or belongs to an entity that appears in the `SELECT` clause.<sup>1</sup>

Again, the syntax should be self explanatory, with the default sort direction being `ASC`. The `DESC` direction can be specified as the second parameter to the `orderBy()` call :

```
Customer c = alias(Customer.class, "c");
String q = select(c).from(Customer.class).as(c)
    .orderBy(c.getFirstname(), DESC).toString();
assertEquals("select c from Customer as c order by c.firstname desc", q);
```

It is of course possible to order the results according to multiple properties. To do so, chain the ordering properties using the `andThenBy()` method call which accepts an optional direction too :

```
Customer c = alias(Customer.class, "c");
String q = select(c).from(Customer.class).as(c)
    .orderBy()
    .andThenBy(c.getLastname(), DESC)
    .toString();
assertEquals("select c from Customer as c order by c.firstname asc, c.lastname desc", q);
```

---

## Chapter 8. LIQUidFORM in 4 easy steps

This final chapter sums up the key points you should remember when using LIQUidFORM. The basic ideas summarized here should help you in 80% of all queries you're likely to encounter in medium-size applications.

- I. Begin with `import static com.google.code.liquidform.LiquidForm.*` : that should give you access to all you need to build a query, be it the aliasing functionality, the `select()` bootstrap construct and most of the `ConditionalExpressions` builders.
- II. Declare *identification variables* using the various `alias(X x, String alias)` methods.
- III. Start your query with `select()` and build it from left to right, as you would if you were writing a plain JPA query.
- IV. The only gotcha is that conditions in the `WHERE` clause are written in prefixed form.

---

## Part II. LIQUidFORM Reference

If you just finished reading Part I, “LIQUidFORM by Example”, then we suggest you don't read further and go experiment. This second part of the manual serves more as a complete list of the syntactic constructs offered by LIQUidFORM than as a tutorial.

As a matter of fact, we secretly hope than you will never have to read this part ! Indeed, once you get the general idea of how the JPA Query syntax was transposed to Java, you should only need to browse this reference looking for some explanation about types, or maybe you will be curious about the method names used for aggregate functions. But that should be it.

---

## Chapter 9. The `SELECT` clause

---

# Chapter 10. The `FROM` clause

## 10.1. Aliasing

---

## Chapter 11. The `WHERE` clause

---

## Chapter 12. The `GROUP BY` clause



---

## Chapter 13. The `HAVING` clause

---

## Chapter 14. The `ORDER BY` clause