**Extensible Component Scanner**

**Version 0.2b**

**User Manual**

**Matthias Rothe (**[**mimarox@users.sourceforge.net**](mailto:mimarox@users.sourceforge.net)**)**

**2011-03-21**

# Before it all starts

It has been pointed out that the Extensible Component Scanner project lacks decent user documentation. In recognition of that fact this manual is put together with the best hopes of solving the issue.

As this project is currently run as a one man show and I wanted to go by releasing early, releasing often I wanted to get the code out first. The documentation took a while to come. But finally here it is! ;-) So keep reading and enjoy!

Table of Contents

[Before it all starts 2](#_Toc288506247)

[1. The purpose of Component Scanning 4](#_Toc288506248)

[1.1. What is a Component 4](#_Toc288506249)

[1.2. Why to scan for Components 5](#_Toc288506250)

[1.3. Is Component Scanning the same as Classpath Scanning 5](#_Toc288506251)

[2. The Extensible Component Scanner API 6](#_Toc288506252)

[2.1. The ComponentScanner class 6](#_Toc288506253)

[2.2. The ComponentQuery class 6](#_Toc288506254)

[2.3. A first code example 7](#_Toc288506255)

[3. The Component Query Language 8](#_Toc288506256)

[3.1. The select clause 8](#_Toc288506257)

[3.2. The from clause 8](#_Toc288506258)

[3.3. The andStore clause 9](#_Toc288506259)

[3.4. The returning clause 10](#_Toc288506260)

[3.5. Annotation matching clauses 11](#_Toc288506261)

[3.6. Annotation with specific arguments matching clauses 12](#_Toc288506262)

[3.7. Extending superclass matching clauses 14](#_Toc288506263)

[3.8. Implementing a single interface matching clause 15](#_Toc288506264)

[3.9. Implementing one or more interfaces matching clauses 15](#_Toc288506265)

[3.10. Combining matching clauses 16](#_Toc288506266)

[4. Afterword 19](#_Toc288506267)

# The purpose of Component Scanning

## What is a Component

For the matters of the Extensible Component Scanner a component is some artifact that can be loaded and used as a Java class and conforms to some restrictions. It could either be a precompiled Java class or any class defined in source code of a programming language targeting the Java Virtual Machine (JVM) for which there is an extension for the Extensible Component Scanner available.

Currently only precompiled Java classes can be used, an extension for Groovy is under development. To be considered a component a Java class needs to conform to the following restrictions:

1. It must be a **class**: Interfaces, annotations and enums cannot be components
2. It must be a **primary** class: Nested or inner classes of any kind cannot be components
3. It must be an **instantiable** class: Abstract classes cannot be components

Note:

A class does not need to be public to be considered a component. It could also be package private. However, to be able to instantiate a package private class it must have at least one declared public constructor. This constructor must then also be made accessible before calling it. Supposing you want to work with a default constructor without any parameters you would have to instantiate that class this way:

Constructor<?> constructor = clazz.getConstructor();

constructor.setAccessible(true);

constructor.newInstance();

instead of just using the standard approach of:

clazz.newInstance();

This last way only works with public classes, since they are accessible per se. Needless to say that public classes also need a defined public no-args constructor or no defined constructor at all for this to work.

Any further advice on instantiating the classes retrieved by the Extensible Component Scanner, or on using them in any other way, is beyond the scope of this manual.

Besides conforming to the restrictions just given, components should also be annotated with any kind of annotation, implement some interface or extend some class, other than java.lang.Object, or have a combination of these characteristics. This ensures that they have a special meaning and purpose in your application and can be found by matching them against criteria more narrow than extending java.lang.Object.

## Why to scan for Components

There are two primary cases in which you’d probably like to use component scanning. The first one is to replace configuration files e.g. telling your application which classes to use as plug-ins or your framework or container which classes to use for a particular application. For example in the Spring framework ([www.springframework.com](http://www.springframework.com)) component scanning is used as an alternative to defining all the different beans of an application in configuration XML files. As these can become quite large this reduces the amount of configuration code quite a lot.

So if you are developing a new application featuring plug-ins, a new framework or container and want to make the lives of the developers targeting your system as painless as possible use component scanning instead of configuration files.

Even though in the first case component scanning is quite useful, you could do without. Not so in the second case. Suppose you wanted to know all the classes in a particular package that are serializable. In this case you cannot configure anything, as you simply don’t know all those classes. There is nothing short of manually looking through all classes in that package to find those that match, except component scanning.

In short component scanning is good for two things: saving configuration code and saving time.

## Is Component Scanning the same as Classpath Scanning

As component scanning refers to **what** you want to find and classpath scanning refers to **where** you want to find something, those terms are strictly speaking not equal. Furthermore with the Extensible Component Scanner you are not restricted to scanning the classpath that is known to the JVM at the moment of scanning to find the components you are looking for.

The components are retrieved as resources from a class loader using the method

ClassLoader.getResources(String name)

Since it is possible to pass a custom class loader to the Extensible Component Scanner you could retrieve components from just about any place, depending on the implementation of the getResources method of the class loader you are using.

# The Extensible Component Scanner API

The API of the Extensible Component Scanner is split into two parts. The first one is comprised of the classes and methods you use to process a scanning run. This is described in this chapter. The second one is the Component Query Language, an embedded domain specific language (eDSL) you use to create a query defining which components you actually want to find. The Component Query Language is described in chapter 3.

## The ComponentScanner class

The main entry point to the Extensible Component Scanner is the class ComponentScanner in the package net.sf.extcos. It features two public methods, both of which are part of the API.

**ComponentScanner.**

**getClasses(componentQuery: ComponentQuery): Set<Class<?>>**

This method lets you specify a component query defining the criteria components you are interested in must match. It looks up the default class loader and uses it to fulfill the request. It returns a set of classes matching the defined criteria of the query. The default class loader is retrieved via

Thread.*currentThread*().getContextClassLoader();

**ComponentScanner.**

**getClasses(componentQuery: ComponentQuery, classLoader: ClassLoader): Set<Class<?>>**

Along with the component query defining the criteria components you are interested in must match this method lets you specify a custom class loader to be used to fulfill the request. This allows you for example to use the Extensible Component Scanner within a web container like Tomcat and use its WebappClassLoader. This method returns a set of classes matching the defined criteria of the query.

## The ComponentQuery class

This abstract class in the package net.sf.extcos is the base for all component query definitions to be passed to one of the ComponentScanner methods. It defines one abstract method which needs to be implemented by the subclass used to define a specific component query:

**ComponentQuery.query(): void**

This method acts as a container for the actual query. It must contain exactly one Component Query Language query.

Besides the query method this class features a number of eDSL methods. These are explained in chapter 3.

## A first code example

Now that we know about the ComponentScanner and the ComponentQuery classes it’s time for a first code example. You may use it as a template every time you use the Extensible Component Scanner.

ComponentScanner scanner = new ComponentScanner();

Set<Class<?>> classes = scanner.getClasses(new ComponentQuery() {

protected void query() {

// the actual query goes here

}

});

This example uses the default class loader. If you want to use a custom class loader, pass it to the getClasses method, like so:

ClassLoader customClassLoader = getCustomClassLoader();

ComponentScanner scanner = new ComponentScanner();

Set<Class<?>> classes = scanner.getClasses(new ComponentQuery() {

protected void query() {

// the actual query goes here

}

}, customClassLoader);

Implement the getCustomClassLoader method in whatever manner you like and you’re done.

# The Component Query Language

The Component Query Language (CQL) is an eDSL and the heart and soul of the Extensible Component Scanner. It defines what kind of artifacts to find from which packages matching which criteria and how to return or where to store the resulting components. The CQL is currently comprised of the four clauses select, from, andStore, and returning.

## The select clause

The select clause defines which kinds of artifacts to find components of. It’s defined as the two methods

**ComponentQuery.select(): BasePackageSelector**

and

**ComponentQuery.select(ResourceType... resourceTypes): BasePackageSelector**

The first acts as a default and calls the second with the resource type for precompiled Java classes. So every time you just want to find precompiled Java classes you can use that method without needing to specify any resource type.

In order to make the CQL easily readable resource types are required to comply with a convention. They need to be classes with a private constructor and a static method returning an instance, rather like singletons. That static method should be called something like javaClasses or groovySources and needs to be imported as a static import. In the case of resource type for precompiled Java classes this method is implemented as

JavaClassResourceType.javaClasses()

in the net.sf.extcos.internal package. The specifics for extensions for other kinds of artifacts are documented with each extension separately.

For precompiled Java classes the following two listings are equivalent:

select()

and

import static net.sf.extcos.internal.JavaClassResourceType.javaClasses;

select(javaClasses())

## The from clause

The from clause defines where to find the components. More precisely it defines the base package or base packages in which components will be found. It’s defined as the method

**BasePackageSelector.from(String... basePackages): ForwardingBuilder**

You may specify various base packages. Make sure that you pass only valid package names as defined by the Java Language Specification. Otherwise an IllegalArgumentException will be thrown. Although technically possible it is not permissible to pass no base package at all. Doing so will also result in an IllegalArgumentException.

A base package is a package that contains classes and / or other packages. If it contains other packages those subpackages will also be scanned and matching components will also be returned. Supposing you’ve got two packages foo.bar1 and foo.bar2 and you pass foo as the base package to the from clause, then all matching components from foo, foo.bar1 and foo.bar2 will be returned.

**Examples**

Now that we learned about the select clause and the from clause, it’s time for the first complete example. Suppose you want to retrieve all the precompiled Java class components from the foo package, this is the way to do it:

ComponentScanner scanner = new ComponentScanner();

Set<Class<?>> classes = scanner.getClasses(new ComponentQuery() {

protected void query() {

select().from(“foo”);

}

});

## The andStore clause

The andStore clause allows you to retrieve several different sets of components with just one query. It’s an optional clause, so if you just need to get one set of components you don’t have to use it. The andStore clause is defined as

**ForwardingBuilder.andStore(StoreBinding... bindings): ReturningSelector**

and takes one or more StoreBinding objects as its parameters. Although it is technically possible to pass no StoreBinding object at all into the andStore clause, this is not permissible and will result in an IllegalArgumentException.

The CQL way of obtaining the StoreBinding objects to pass into the andStore clause is to use matching clauses defining component filters. Only matching components will be stored. These matching clauses are defined in the ComponentQuery class and described in detail in sections 3.5 to 3.10. The following table gives an overview.

|  |  |
| --- | --- |
| **Matching clause** | **Described in section** |
| thoseAnnotatedWith(Class<? extends Annotation> annotation) | 3.5 |
| thoseAnnotatedWith(Class<? extends Annotation> annotation, ArgumentsDescriptor arguments) | 3.6 |
| thoseExtending(Class<T> clazz) | 3.7 |
| thoseImplementing(Class<T> interfaze) | 3.8 |
| thoseImplementing(Class<?>... interfaces) | 3.9 |
| thoseBeing(TypeFilterJunction filter) | 3.10 |

Each matching clause returns either a TypelessStoreBindingBuilder or a TypedStoreBindingBuilder<T>. Both classes define an into clause which specifies where the components matching the given matching clause will go to. The difference in the definition of the into clause is in the type of Set it takes. The into clause in the TypelessStoreBindingBuilder is defined as

**TypelessStoreBindingBuilder.into(Set<Class<?>> store): StoreBinding**

and takes a Set which can contain any kind of class object. In contrast the into clause in the TypedStoreBindingBuilder is defined as

**TypedStoreBindingBuilder<T>.into(Set<Class<? extends T>> store): StoreBinding**

and takes a Set which can only contain classes extending or implementing the type T which has originally been defined in the matching clause. That way the type information given via the matching clause is used to require the type of the Set to be as specific as possible. This gives you additional type safety. Obviously this is only possible if the matching clause defines one specific type to match components against. This applies only to the thoseExtending and thoseImplementing clauses, the thoseImplementing clause taking just one parameter.

As you can see the into clauses return the StoreBinding the andStore clause requires. So each StoreBinding object is created by using a matching clause followed by an into clause.

**Examples**

Suppose you want to get all components implementing the java.io.Serializable interface in one set and all components annotated with some SampleAnnotation in another set, all components being located in the foo package. Then all you need is

final Set<Class<? extends Serializable>> serializables = new ...;

final Set<Class<?>> samples = new HashSet<Class<?>>();

ComponentScanner scanner = new ComponentScanner();

Set<Class<?>> classes = scanner.getClasses(new ComponentQuery() {

protected void query() {

select().from(“foo”).andStore(

thoseImplementing(Serializable.class).into(serializables),

thoseAnnotatedWith(SampleAnnotation.class).into(samples));

}

});

Note that the classes set is empty in this case, as there is no returning clause given and the default behavior in this case is to return an empty set.

## The returning clause

The returning clause defines which kind of components will be returned in the set returned by the query. As the andStore clause the returning clause is an optional clause. What is returned if the returning clause isn’t specified depends on whether the andStore clause is used or not. In case the andStore clause is used the returned set is empty. If it’s not used all components found are returned. The returning clause has two definitions. The first one is used when the andStore clause isn’t used and it directly follows the from clause. In this case it’s defined as

**ForwardingBuilder.returning(DirectReturning returning): void**

and takes a definition fitting the case of directly returning, without storing any components into additional stores. These definitions are provided by matching clauses. A special case is the all matching clause, which causes each and every component found to be returned. This is equivalent to not specifying the returning clause at all. The other matching clauses are listed in the following table and explained in the given section.

|  |  |
| --- | --- |
| **Matching clause** | **Described in section** |
| allAnnotatedWith(Class<? extends Annotation> annotation) | 3.5 |
| allAnnotatedWith(Class<? extends Annotation> annotation, ArgumentsDescriptor arguments) | 3.6 |
| allExtending(Class<?> clazz) | 3.7 |
| allImplementing(Class<?>... interfaces) | 3.9 |
| allBeing(TypeFilterJunction filter) | 3.10 |

If the andStore clause is used the returning clause follows it and is defined as

**ReturningSelector.returning(StoreReturning returning): void**

In addition to the matching clauses already mentioned this case of the returning clause takes two more matching clauses. These are the allMerged clause, returning all the components stored in the different stores defined in the andStore clause merged into the returned set, and the none clause causing an empty set to be returned. This is equivalent to not specifying the returning clause at all. All matching clauses including all, allMerged and none are defined as methods in the ComponentQuery class.

**Examples**

Suppose you want to retrieve all components implementing the interface java.io.Serializable from the foo package. The code required to accomplish that task would be

ComponentScanner scanner = new ComponentScanner();

Set<Class<?>> classes = scanner.getClasses(new ComponentQuery() {

protected void query() {

select().from(“foo”).

returning(allImplementing(Serializable.class));

}

});

## Annotation matching clauses

Annotation matching clauses specify which annotation a component must be annotated with to match and be stored or returned. There is one annotation matching clause to be used within the andStore clause and one to be used within the returning clause. The former is thoseAnnotatedWith and the latter is allAnnotatedWith. They are defined as

**ComponentQuery.thoseAnnotatedWith(Class<? extends Annotation> annotation): TypelessStoreBindingBuilder**

and

**ComponentQuery.allAnnotatedWith(Class<? extends Annotation> annotation): DirectReturning**

respectively.

**Examples**

Suppose you want to store all components annotated with some SampleAnnotation from the foo package into a set. The code to accomplish this would be

final Set<Class<?>> samples = new HashSet<Class<?>>();

ComponentScanner scanner = new ComponentScanner();

scanner.getClasses(new ComponentQuery() {

protected void query() {

select().from(“foo”).andStore(

thoseAnnotatedWith(SampleAnnotation.class).into(samples));

}

});

In case you want to return the same set of components the code to get this done would be

ComponentScanner scanner = new ComponentScanner();

Set<Class<?>> classes = scanner.getClasses(new ComponentQuery() {

protected void query() {

select().from(“foo”).returning(

allAnnotatedWith(SampleAnnotation.class));

}

});

## Annotation with specific arguments matching clauses

In addition to the annotation matching clauses (see previous section) the annotation with specific arguments matching clauses allow you to not only find components annotated with a certain annotation, but to be matched the given arguments of the given annotation must also match for the component to be stored or returned. There is one annotation with specific arguments matching clause to be used within the andStore clause and one to be used within the returning clause. The former is thoseAnnotatedWith and the latter is allAnnotatedWith. They are defined as

**ComponentQuery.thoseAnnotatedWith(Class<? extends Annotation> annotation, ArgumentsDescriptor arguments): TypelessStoreBindingBuilder**

and

**ComponentQuery.allAnnotatedWith(Class<? extends Annotation> annotation, ArgumentsDescriptor arguments): DirectReturning**

respectively.

Apart from the ArgumentsDescriptor these two matching clauses work exactly in the same way as the ones described in the previous section. The CQL way of obtaining an ArgumentsDescriptor defining the annotation arguments’ matching rules is to use one of the clauses withArgument and withArguments. These are defined as

**ComponentQuery.withArgument(ArgumentKey key, ArgumentValue value): ArgumentsDescriptor**

**ComponentQuery.withArguments(ArgumentMappingConjunction arguments): ArgumentsDescriptor**

and

**ComponentQuery.withArguments(ArgumentMappingDisjunction arguments): ArgumentsDescriptor**

respectively.

The ArgumentKey and ArgumentValue objects are to be obtained using the key and value clauses defined as

**ComponentQuery.key(String key): ArgumentKey**

and

**ComponentQuery.value(Object value): ArgumentValue**

respectively. As of this version the keys and values are used exactly as they are given and there is no way to use pattern, range or other possible kinds of matching.

Using the withArguments clause gives you the option to group two or more ArgumentMappings by one of the logical functions “and” and “or” using their respective clauses defined as

**ComponentQuery.and(ArgumentMapping... mappings): ArgumentMappingConjunction**

and

**ComponentQuery.or(ArgumentMapping... mappings): ArgumentMappingDisjunction**.

The CQL way of obtaining the ArgumentMapping objects is by using the mapping clause defined as

**ComponentQuery.mapping(ArgumentKey key, ArgumentValue value): ArgumentMapping**.

As to the ArgumentKey and ArgumentValue objects the description above applies in the same way as it does to the withArgument clause. There is currently no way to define logical subgroups of ArgumentMappings.

**Examples**

Suppose you’d want to find all components from the foo package annotated with a SampleAnnotation that’s got a sampleKey argument the value of which must be “sampleValue” for the component to be returned. The code would have to be this:

ComponentScanner scanner = new ComponentScanner();

Set<Class<?>> classes = scanner.getClasses(new ComponentQuery() {

protected void query() {

select().from(“foo”).returning(

allAnnotatedWith(SampleAnnotation.class, withArgument(

key(“sampleKey”), value(“sampleValue”))));

}

});

Suppose the SampleAnnotation had also an anotherKey argument the value of which must be “anotherValue” while the value of the sampleKey must still be “sampleValue” for the component to be returned. The code would then change to:

ComponentScanner scanner = new ComponentScanner();

Set<Class<?>> classes = scanner.getClasses(new ComponentQuery() {

protected void query() {

select().from(“foo”).returning(

allAnnotatedWith(SampleAnnotation.class, withArguments(and(

mapping(key(“sampleKey”), value(“sampleValue”)),

mapping(key(“anotherKey”), value(“anotherValue”)))));

}

});

If only one of the two arguments must have the aforementioned respective value the logical function to be applied would be “or” and the code would change to:

ComponentScanner scanner = new ComponentScanner();

Set<Class<?>> classes = scanner.getClasses(new ComponentQuery() {

protected void query() {

select().from(“foo”).returning(

allAnnotatedWith(SampleAnnotation.class, withArguments(or(

mapping(key(“sampleKey”), value(“sampleValue”)),

mapping(key(“anotherKey”), value(“anotherValue”)))));

}

});

There could obviously be more than two mappings and the argument values could be any valid annotation argument values like some primitive or enum value.

## Extending superclass matching clauses

Extending superclass matching clauses specify which class a component must either directly or indirectly inherit from to match and be stored or returned. There is one extending superclass matching clause to be used within the andStore clause and one to be used within the returning clause. The former is thoseExtending and the latter is allExtending. They are defined as

**ComponentQuery.thoseExtending(Class<T> clazz): TypedStoreBindingBuilder<T>**

and

**ComponentQuery.allExtending(Class<?> clazz): DirectReturning**

respectively.

**Examples**

Suppose you want to store all components extending some SampleBaseClass from the foo package into a set. The code to accomplish this would be

final Set<Class<? extends SampleBaseClass>> samples = new HashSet<Class<? extends SampleBaseClass>>();

ComponentScanner scanner = new ComponentScanner();

scanner.getClasses(new ComponentQuery() {

protected void query() {

select().from(“foo”).andStore(

thoseExtending(SampleBaseClass.class).into(samples));

}

});

In case you want to return the same set of components the code to get this done would be

ComponentScanner scanner = new ComponentScanner();

Set<Class<?>> classes = scanner.getClasses(new ComponentQuery() {

protected void query() {

select().from(“foo”).returning(

allExtending(SampleBaseClass.class));

}

});

## Implementing a single interface matching clause

The clause matching components either directly or indirectly implementing the single interface given is special in that it only exists within the andStore clause and doesn’t exist within the returning clause. Its purpose is to retain the type information given with the specification of the interface components to be stored must implement. This information is then used to require the storing set to be as specific as possible. This matching clause is defined as

**ComponentQuery.thoseImplementing(Class<T> interfaze): TypedStoreBindingBuilder<T>**.

**Examples**

Suppose you want to store all components implementing some SampleInterface from the foo package into a set. The code to accomplish this would be

final Set<Class<? extends SampleInterface>> samples = new HashSet<Class<? extends SampleInterface>>();

ComponentScanner scanner = new ComponentScanner();

scanner.getClasses(new ComponentQuery() {

protected void query() {

select().from(“foo”).andStore(

thoseImplementing(SampleInterface.class).into(samples));

}

});

## Implementing one or more interfaces matching clauses

Implementing one or more interfaces matching clauses specify which interface or interfaces a component must either directly or indirectly implement to match and be stored or returned. There is one such matching clause to be used within the andStore clause and one to be used within the returning clause. The former is thoseImplementing and the latter is allImplementing. They are defined as

**ComponentQuery.thoseImplementing(Class<?>... interfaces): TypelessStoreBindingBuilder**

and

**ComponentQuery.allImplementing(Class<?>... interfaces): DirectReturning**

respectively.

**Examples**

Suppose you want to store all components implementing both interfaces SampleInterfaceA and SampleInterfaceB from the foo package into a set. The code to accomplish this would be

final Set<Class<?>> samples = new HashSet<Class<?>>();

ComponentScanner scanner = new ComponentScanner();

scanner.getClasses(new ComponentQuery() {

protected void query() {

select().from(“foo”).andStore(

thoseImplementing(SampleInterfaceA.class,

SampleInterfaceB.class).into(samples));

}

});

In case you want to return the same set of components the code to get this done would be

ComponentScanner scanner = new ComponentScanner();

Set<Class<?>> classes = scanner.getClasses(new ComponentQuery() {

protected void query() {

select().from(“foo”).returning(

allImplementing(SampleInterfaceA.class,

SampleInterfaceB.class));

}

});

As the interfaces parameter is a varargs parameter any amount of interfaces could be given. This includes the technically possible option of giving none at all. However this will result in an IllegalArgumentException at runtime.

All interfaces given must be implemented by any component to be matched. The implicit logical operator is therefore the “and” operator. Should you wish to use the “or” operator you need to state this explicitly using the appropriate combining matching clause.

## Combining matching clauses

To combine the matching clauses described in the previous sections one of the two combining matching clauses thoseBeing and allBeing can be used within the andStore and the returning clauses respectively. The combining matching clauses are defined as

**ComponentQuery.thoseBeing(TypeFilterJunction filter): TypelessStoreBindingBuilder**

and

**ComponentQuery.allBeing(TypeFilterJunction filter): DirectReturning**.

The TypeFilterJunction can either be a TypeFilterConjunction if the logical operator “and” is to be used or a TypeFilterDisjunction if the logical operator “or” is to be used. The clauses needed to obtain the corresponding objects are therefore

**ComponentQuery.and(ExtendingTypeFilter filter, MultipleTypeFilter... filters): TypeFilterJunction**,

**ComponentQuery.and(MultipleTypeFilter... filters): TypeFilterJunction**

and

**ComponentQuery.or(TypeFilter... filters): TypeFilterJunction**.

The first of the two “and” clauses is to be used if the components to match must extend a certain base class and match some other given criteria, except class inheritance. Since any class can only extend one class directly and only one line of classes indirectly up to the Object class the explicit “and” operator isn’t applicable to class inheritance.

The second “and” clause is to be used if the components to match don’t need to extend a certain base class, but must match several other criteria. These can be stated in any order and can include further “and” and “or” clauses. The “or” clause takes any kind of matching criteria in any order, again including further “and” and “or” clauses.

The varargs parameters in the clauses could again also take no parameters at all, which would yet again result in an IllegalArgumentException being thrown at runtime. It’s also technically possible and doesn’t result in a runtime exception to give just one parameter to each varargs parameter, but at least for the second “and” clause and the “or” clause doing so clearly wouldn’t make any sense.

The matching clauses to be used within the “and” and “or” clauses are slightly different than the ones used directly within the andStore and returning clauses. This way a more fluent language is achieved. The following table shows the matching clauses as used within the returning clause and the matching clause to be used within the “and” and “or” clauses corresponding to each.

|  |  |
| --- | --- |
| **Returning matching clause** | **Corresponding matching clause** |
| allAnnotatedWith(Class<? extends Annotation> annotation) | annotatedWith(Class<? extends Annotation> annotation) |
| allAnnotatedWith(Class<? extends Annotation> annotation, ArgumentsDescriptor arguments) | annotatedWith(Class<? extends Annotation> annotation, ArgumentsDescriptor arguments) |
| allExtending(Class<?> clazz) | subclassOf(Class<?> clazz) |
| allImplementing(Class<?>... interfaces) | implementorOf(Class<?>... interfaces) |

All corresponding matching clauses are defined as methods of the ComponentQuery class, as are all other matching clauses. All other syntaxes and semantics stay the same as described in the sections above.

**Examples**

Suppose you want to store all components extending SampleBaseClass and implementing SampleInterface from the foo package into a set. The code to accomplish this would be

final Set<Class<?>> samples = new HashSet<Class<?>>();

ComponentScanner scanner = new ComponentScanner();

scanner.getClasses(new ComponentQuery() {

protected void query() {

select().from(“foo”).andStore(

thoseBeing(and(

subclassOf(SampleBaseClass.class),

implementorOf(SampleInterface.class))

.into(samples));

}

});

In case you want to return the same set of components the code to get this done would be

ComponentScanner scanner = new ComponentScanner();

Set<Class<?>> classes = scanner.getClasses(new ComponentQuery() {

protected void query() {

select().from(“foo”).returning(

allBeing(and(

subclassOf(SampleBaseClass.class),

implementorOf(SampleInterface.class)));

}

});

As another example suppose you want to store all components implementing SampleInterface or annotated with SampleAnnotation from the foo package into a set. The code to accomplish this would be

final Set<Class<?>> samples = new HashSet<Class<?>>();

ComponentScanner scanner = new ComponentScanner();

scanner.getClasses(new ComponentQuery() {

protected void query() {

select().from(“foo”).andStore(

thoseBeing(or(

implementorOf(SampleInterface.class),

annotatedWith(SampleAnnotation.class))

.into(samples));

}

});

A lot more examples could be given, but this should suffice to clarify the issue.

# Afterword

In case this manual leaves any questions you have unanswered, please get in touch. You can either post a question in the support forums at <http://sf.net/projects/extcos/forums> or write an email to [mimarox@users.sourceforge.net](mailto:mimarox@users.sourceforge.net). Any question or remark is highly appreciated and will be promptly answered.