- Code Query Language -

Second Edition (01/01/2010)

Table of Contents

1 Introduction

The Code Query Language (CQL) is a domain-specific language used to query the Java source code and bytecode. It is also a declarative language. According to Wikipedia, declarative programming is a programming paradigm that expresses the logic of a computation without describing its control flow.

In the context of the Reflection Toolkit (REFLECTK) this means that CQL is only used to describe the attributes of a searched Java language entity and not how to iterate over the classpath or to parse each classpath component. Some concepts are borrowed from redicate logic others from the heory of sets.

Let's have a look at a few simple CQL statements to get a feeling for the language syntax. The first statement lists all names of classes that implement the interface java.io.Serializable.

```
select @name
from classes
where
   inherit('java.io.Serializable');
```

The second example prints all names of static fields that are part of a class implementing the interface java.io.Serializable.

```
select @name
from fields
where
   modifier('STATIC')
   && elementof class {
      inherit('java.io.Serializable')
   };
```

The third and last example prints all names of methods starting with “is” and having a boolean return type.

```
select @name
from methods
where
   returntype('boolean') && name('is.*');
```

2 Search Entities

The intention of each CQL-described search is to find objects that match a certain criteria. Throughout this document these objects are called search entities. Because the search is performed on Java source code or bytecode, search entities are Java language entities like classes, fields, methods or annotations.

2.1 Packages

Package are the top-level entities. They are used to group a set of classes. Considering a familiar Java class, the package is usually the declaration.

```
Synonyms:
   pkg, pkg, package, packages
```

2.2 Classes

Class entities and all its sub entities may come in two different flavors: source code and bytecode.

```
Synonyms:
   cls, class, classes
```

2.3 Methods

Methods are important class members. It is important to know that a method only exists in a class if it is actually declared in it. Even though a super class' method can be invoked (if visibility is appropriate) at a sub class, it is not declared within this class. In other words: in CQL methods are only listed when they are declared within a specific class. Special cases are overridden methods.

```
Synonyms:
   meth, method, methods
```

2.4 Fields

Fields are also imported class members. Basically everything that have been said about methods is also applicable for fields. The only difference is that overriding is not possible. Fields in a sub class are shadowing field with the same name within the super class.

```
Synonyms:
   fld, field, fields
```

2.5 Constructors

Constructors have to be declared within each class. If no constructor is explicitly specified a default constructor is implicitly generated (if possible). The

first statement within each constructors is a call of a super class' constructor. Additionally to the normal constructors there exists another type of constructor: the so called static constructor or class initializer. These constructors are used to initialize static class content. Both types of constructors can be searched.

```
Synonyms:
   con, constructor, constructors
```

2.6 Annotations

Basically all kind of search entity can be annotated. Even annotation can annotated. Note that there exists different kind of annotations: some are only available in source code, others are also available in bytecode, but not during run time (compare java.lang.annotation.RetentionPolicy for further information).

```
Synonyms:
   anno, annotation, annotations
```

2.7 Instructions

Some examples for Instructions are variable assignments or initializations, exception handling blocks or method invocations.

```
Synonyms:
   inst, instruction, instructions
```

2.8 Literals

Literals are simply primitive values.

```
Synonyms:
   lit, literal, literals
```

3 Syntax

Basically a CQL statements consists of three different parts: attribute selection, domain selection and conditional term. Here is the complete syntax specification in EBNF notation. Note that all terminals (CQL language parts) are bold and surrounded by double quotes. Non-terminals are italic and enclosed by lesser/greater sign.

```
["set" <CLASSPATH> [<CLASSPATH>]]

(<VARIABLE>"="<SIGNATURE>";")*

(["select"] <ATTRIBUTE>+ ["from"] <ENTITY TYPE> [["where"] <TERM>]";")+

<CLASSPATH>::="["<ARGUMENT>(<SEPARATOR><ARGUMENT>)*"]"

<SEPARATOR>::=(","|":")

<ARGUMENT>::="'" [^"'"]* "'"

<VARIABLE>::="$"("a"-"z"|"A"-"Z"|"_")
             ("0"-"9"|"a"-"z"|"A"-"Z"|"_")*

<SIGNATURE>::=(<ARGUMENT>|<VARIABLE>)
              (<SEPARATOR>(<ARGUMENT>|<VARIABLE>))*

<ATTRIBUTE>::="@"("a"-"z"|"A"-"Z"|"_")
               ("0"-"9"|"a"-"z"|"A"-"Z"|"_")*

<ENTITY TYPE>::=("pkg"|"package"["s"]|
                 "cls"|"class"["es"]|
                 "meth"|"method"["s"]|
                 "con"|"constructor"["s"]|
                 "fld"|"field"["s"]|
                 "anno"|"annotation"["s"])

<TERM>::=<FILTER>|("not"|"!")<TERM>|
         <TERM>(("and"|"&"|"&&"|"or"|"|"|"||"|"xor"|"^")<TERM>)*
         |<CONTEXT SWITCH>"{"<TERM>"}"

<CONTEXT SWITCH>::=("fa"|"forall"|"ex"|"exist"|
                    "eo"|"elementof"|"in")
                    ("inner"|"innerclass"["s"]|
                    "super"|"superclass"["es"]|
                    "sub"|"subclass"["es"]|
                    "call"["s"]|"type"["s"]|"return"["s"]|
                    "exception"["s"]|"parameter"["s"]|
                    <ENTITY TYPE>)

<FILTER>::="name""("<SIGNATURE>")"
```

4 Attribute Selectors

Attribute selectors are used to specify the columns of the result set. Let's take a look at a example that illustrates how attribute selectors are used. This CQL statement prints 3 attributes of a shown search entity: the entity type, the entity name and the last entity processed during the corresponding sub search. It finds all classes that call any class with a full-qualified name starting with “java”.

```
select @type @name @last
from classes
where {
    ex call {
        eo class { fqname('java.*')  }
    };
}
```

Note that the @last selector prints the called entity that causes the actual class to be part of the result set. The following figure shows the output of the given CQL statement:

INDEX	@type	@name	@last
134	CLASS	de.reflectk.filter.EntityTypeFilter	CLASS|java.lang.Object
135	CLASS	de.reflectk.filter.FullQualifiedNameFilter	CLASS|java.util.regex.Pattern
136	CLASS	de.reflectk.filter.InheritanceFilter	CLASS|java.lang.Object
137	CLASS	de.reflectk.filter.LiteralValueFilter	CLASS|java.util.regex.Pattern
138	CLASS	de.reflectk.filter.MagicNumberFilter	CLASS|java.util.EnumSet
139	CLASS	de.reflectk.filter.MethodFilter$Polymorphism	CLASS|java.lang.Enum
140	CLASS	de.reflectk.filter.MethodFilter	CLASS|java.lang.Object
141	CLASS	de.reflectk.filter.ModifierFilter	CLASS|java.lang.Object
142	CLASS	de.reflectk.filter.NameFilter	CLASS|java.util.regex.Pattern
143	CLASS	de.reflectk.filter.NotFilter	CLASS|java.lang.Object
144	CLASS	de.reflectk.filter.OrFilter	CLASS|java.util.List
145	CLASS	de.reflectk.filter.PackageFilter$PkgMode	CLASS|java.lang.Enum
146	CLASS	de.reflectk.filter.PackageFilter	CLASS|java.lang.String
147	CLASS	de.reflectk.filter.ParameterListFilter	CLASS|java.lang.Object
148	CLASS	de.reflectk.filter.PrimitiveFilter	CLASS|java.lang.Object
149	CLASS	de.reflectk.filter.PublicDefaultConstructorFilter	CLASS|java.lang.Object

4.1 @type

Selects the type of the search entity,

4.2 @name

Selects the name of the search entity,

4.3 @entity

Selects the search entity itself,

4.4 @last

Selects the last evaluated search entity. This selector is very useful in combination with CQL statements that have context switches, because you can output the reason a specific search entity is in the result set (compare the above example).

4.5 @location

This key shows the first classpath entry the current search entity is located at. Note that this is concordant to the class-loading behavior of the JVM.

4.6 @entries

This key shows all the classpath entries of the current search entity.

4.7 @card

This key shows amount of classpath entries of the current search entity.

4.8 @modifier

Determines the modifiers of the current search entity.

4.9 @magic_no

Determines the magic number of the current search entity. This key is only applicable if bytecode is available.

4.10 @metric_*

Prints the calculated metric.

5 Logical Operators

According to Wikipedia, a logical connective (also called a logical operator) is a symbol or word used to connect two or more sentences (of either a formal or a natural language) in a grammatically valid way, such that the compound sentence produced has a truth value dependent on the respective truth values of the original sentences.

In the CQL context, a logical operator is used to combine different filters to a logical condition a searched entity has to met.

5.1 Not

Negation is a unary operator. It negates the operand's value. If the return value of the negated filter is true, the resulting value is false.

```
Synonyms:
   not, !, ¬
```

5.2 And

Conjunction is a binary connection. It combines two or more logical expressions to return true if and only if all involved filter values are true. Additionally to the standard AND function there exists a so called conditional AND, which guarantees to evaluate as few filters as necessary to compute the return value.

```
Synonyms:
   and, &, && (conditional)
```

5.3 Or

Inclusive disjunction is a binary operator that created a truth-function that return true if at least one of the connected filters returns true. A conditional OR is also available.

```
Synonyms:
   or, |, || (conditional)
```

5.4 Xor

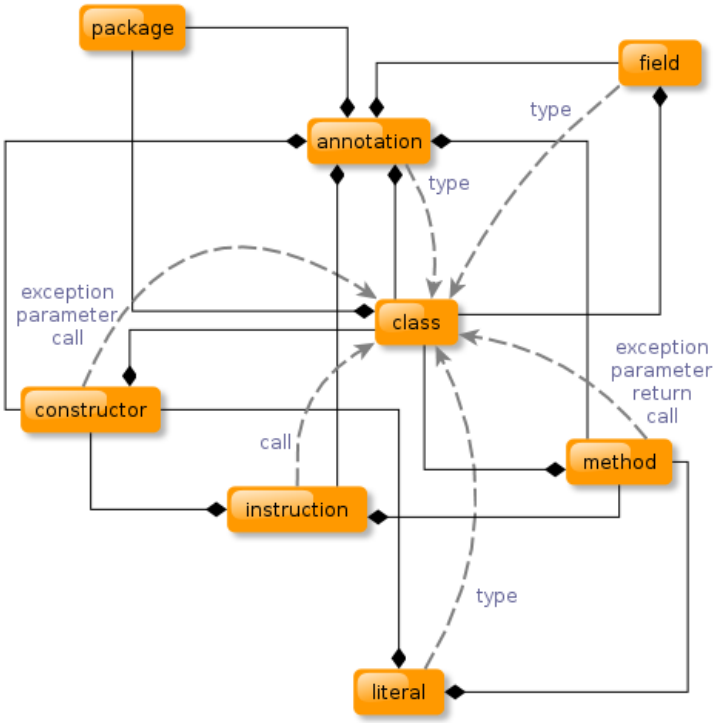
In contrast to the previous mentioned OR function exclusive disjunction is a binary connection that returns true if and only if exactly one of the filters returns true.

```
Synonyms:
   xor, ^
```

6 Context Switches

Context switches provide a mechanism either to reference the surrounding context of a search entity or to specify conditions for the search entities within another entity. In other words, context switches can be used to walk along the edges of containment relations of Java language entities. For example: classes consist of fields, methods and constructors. Methods have parameters, return types, exceptions or annotations. Additionally all context switches have to be specified with one quantifier.

The following graph displays all possible context switches:



6.1 For All

The universal quantifier is borrowed from predicate logic. It is used to express that a given criteria of predicate matches for all objects in a given set.

```
Synonyms:
   forall, fa
```

6.2 Exists

The existential quantifier is also introduced by predicate logic, but in contrast to the universal quantifier it is used to specify that there exists at least one object in the set that matches a given criteria.

```
Synonyms:
   exists, ex
```

6.3 Element Of

Set membership is used to specify that a specific element is part of a specific set. It was originally defined by the theory of sets.

```
Synonyms:
   elementof, eo, in
```

7 Search Filters

Filters (predicates) are a integral part of a CQL statement. They are used to check attributes of search entities and to formulate the search condition. Here is a list of all currently available filters. As a filter's name one of the string under the corresponding synonyms section can be used. The parameters section describes the filter's signature in EBNF notation. The type of search entity a filter can be applied to is listed under supported entities.

The following section lists some parameters that are commonly used by various search filters.

```
Operator:
   <=, le (lesser or equal)
   <, lt (lesser)
   =, ==, eq (equal)
   <>, !=, not, ne (not equal)
   >=, ge (greater or equal)
   >, gt (greater)

Regular Expression Flags:
   UNIX_LINES (only \n line terminator is recognized)
   CASE_INSENSITIVE (case-insensitive matching)
   COMMENTS (permits whitespace and comments with #)
   MULTILINE (in multiline mode the expressions ^ and $ match just
              after or just before a line terminator or the end of the
              input sequence)
   LITERAL (literal parsing of the pattern)
   DOTALL (the expression . matches any character, including a line
           terminator)
   UNICODE_CASE (Unicode-aware case folding)
   CANON_EQ (characters match if their full canonical decompositions
             match)
```

```
Entity Type:
   PACKAGE
   CLASS
   CONSTRUCTOR
   METHOD
   FIELD
   ANNOTATION
   INSTRUCTION
   RESOURCE
   LITERAL
```

7.1 Inheritance

A search filter which is able to filter sub classes. As parameters this filter accepts the class name of the super class or interface.

Additionally a sub class mode parameter can be specified that indicated which type of sub classes should be searched. The ALL mode is used, if none is specified.

```
Synonyms:
   inherit, inheritance, extend, extends, implement, implements

Parameters:
   <CLASSNAME> [, <SUB CLASS MODE>]

Sub Class Modes:
   ALL (accept the class/interface itself and all subclasses),
   REAL (accept all subclasses),
   DIRECT (accept direct (first level) subclasses)

Supported Entities:
   CLASS, FIELD
```

Search all classes that directly extends java.util.Comparator but that do not implement the java.io.Serializable.

```
select @name
from classes
where {
    inherit('java.util.Comparator', 'DIRECT')
    && !inherit('java.io.Serializable')
}
```

7.2 Reference

Instances of this class search for references the specified elements. Can be used for simple reference searches. Please take a look at the “call context switch” (see MagicNumberFilter example) if you need further functionality.

```
Synonyms:
   ref, reference

Parameters:
   <ENTITY>

Supported Entities:
   RESOURCE, PACKAGE, CLASS, CONSTRUCTOR, METHOD, FIELD, ANNOTATION
```

Search all classes that have one ore more references to the class java.lang.StringBuffer.

```
select @name
from classes
where {
    reference('java.lang.StringBuffer')
}
```

7.3 Magic Number

A search filter which is able to filter classes by their magic number. This is a bit misleading, since in OS terms all classfiles have the same magicnumber. Each classfile starts with the following section:

1. 4 bytes magic number (always 0xCAFEBABE)
2. 2 bytes minor version
3. 2 bytes major version

This filter actually uses the major version of the classfile. This filter will work on bytecode only.

```
Synonyms:
   magic, magicnumber

Parameters:
   <OPERATOR>, <MAGIC NUMBER>

Magic number:
   V1_1 (45, 0x2D)
   V1_2 (46, 0x2E)
   V1_3 (47, 0x2F)
   V1_4 (48, 0x30)
   V1_5 (49, 0x31)
   V1_6 (50, 0x32)

Supported Entities:
   CLASS
```

Search all classes compiled with JDK5 or later referencing java.lang.StringBuffer.

```
select @name
from classes
where {
    magicnumber('>=', 'V1_5')
    && exist call {
        entity('java.lang.StringBuffer')
    }
}
```

7.4 Singleton

Filter which finds classes that uses the singleton design pattern.

```
Synonyms:
   singleton

Supported Entities:
   CLASS
```

Search all references to singletons.

```
select @name
from classes
where {
    exist call {
        elementof class {
            singleton()
        }
    }
}
```

7.5 Bean

SearchFilter which can detect Java Beans (classes satisfying the Java Beans Specification). A class is detected as a valid Java Bean must fulfill the following criterias:

1. It must implement java.io.Serializable
2. It must declare a public constructor without arguments
3. It must declare at least one getter with a matching setter

The last criteria can be parameterized by passing a ThresholdStrategy so it's possible that only classes with more than n bean attributes are rated as beans. Without a parameter all beans with one ore more attributes are accepted.

```
Synonyms:
   bean

Parameters:
   [<OPERATOR>, <THRESHOLD>]

Supported Entities:
   CLASS
```

Search all classes hat fulfill the bean contract and that have more than 20 attributes.

```
select @name
from fields
where {
    elementof class {
        bean('>', '20')
    }
}
```

7.6 Parameter List

A search filter which is able to filter methods and constructors by their parameter lists.

```
Synonyms:
   paramlist

Parameters:
   (<CLASSNAME>)*

Supported Entities:
   CLASS
```

Search all classes that have private paramless constructors.

```
select @name
from class
where {
    exist constructor {
        modifier('PRIVATE') && paramlist()
    }
}
```

7.7 Primitive

Filter that accepts all primitive classes. Datatypes that are considered primitive are: boolean, int, short, float, double, void, byte and char.

```
Synonyms:
   primitive

Supported Entities:
   CLASS
```

Search all methods that have primitive returntypes or a primitive parameter.

```
select @name
from methods
where {
    exist return {
        primitive()
    }
    || exist parameter {
        primitive()
    }
}
```

7.8 Name

Search filter that accepts all entities that have a simple name matching the specified regular expression. The name is not the full qualified name, i.e. the parts that correspond to the package are omitted.

```
Synonyms:
   name

Parameters:
   <REGULAR EXPRESSION> (, <REGULAR EXPRESSION FLAGS>)*

Supported Entities:
   PACKAGE, CLASS, CONSTRUCTOR, METHOD, FIELD, ANNOTATION
```

Search all serializable classes that do not have a static field serialVersionUID.

```
select @name
from classes
where {
    inherit('java.io.Serializable')
    && !exist field {
        name('serialVersionUID')
        && modifier('STATIC')
    }
}
```

7.9 Unused

This filter tests weather the entity to be accepted is unused throughout the whole classpath.

Note: This operation is time-consuming. You may want to use it in combination with a caching mechanisms.

```
Synonyms:
   unused, notused

Supported Entities:
   ANNOTATION, CLASS, CONSTRUCTOR, FIELD, INSTRUCTION, LITERAL, METHOD,
   PACKAGE, RESOURCE
```

Search all non-private fields that are not used (not referenced).

```
select @name
from fields
where {
    !modifier('PRIVATE') && unused()
}
```

7.10 Entity

A search filter which that accepts exactly matching entities only.

```
Synonyms:
   entity

Parameters:
   entity

Supported Entities:
   ANNOTATION, CLASS, CONSTRUCTOR, FIELD, INSTRUCTION, LITERAL, METHOD,
   PACKAGE, RESOURCE
```

Search all calls to Object's wait method.

```
select @name
from methods
where {
    exist call {
        entity('java.lang.Object#wait()')
    }
}
```

7.11 Method

Search filter that accept methods that match the specified method.

```
Synonyms:
   declare, decl

Parameters:
   <POLYMORPHISM>, <METHOD ENTITY>
   <POLYMORPHISM>, <CLASSENTITY>, <METHODNAME>(, <CLASSENTITY>)*

Polymorphism:
   OVERRIDING (accept methods that override a method)
   OVERLOADING (accept methods that overload a method)
   DECLARING (accept methods that declare a method)

Supported Entities:
   METHOD
```

Search all classes that do override Object's equals but not hashCode.

```
select @name
from classes
where {
    exist method {
        decl('OVERRIDING','java.lang.Object#equals(java.lang.Object)')
    }
    && !exist method {
        decl('OVERRIDING', 'java.lang.Object#hashCode()')
    }
}
```

7.12 Annotation

A search filter which is able to filter classes by their annotation(s).

```
Synonyms:
   annotated

Parameters:
   <CLASSNAME> (, <KEY>"="<VALUE>)* [, <PARAMETER MATCHING>]

Parameter Matching:
   IGNORE (annotation's parameters are ignored completely)
   SUBSET (the specified annotation's parameters have to be a subset of
           the read annotation's parameters)
   SUPERSET (the specified annotation's parameters have to be a
             superset of the read annotation's parameters)
   EXACT_MATCH (the specified annotation's parameters have to match the
                read annotation's parameters exactly)

Supported Entities:
   ANNOTATION, CLASS, CONSTRUCTOR, FIELD, INSTRUCTION, LITERAL, METHOD,
   PACKAGE, RESOURCE
```

Search all methods in classes named Test.* oder .*Test that are annotated with JUnit's @Test annotation.

```
select @name
from methods
where {
    elementof class {
        name('.*Test') || name('Test.*')
    }
    && annotated('org.junit.Test')
}
```

7.13 Full-Qualified Name

Search filter that accepts all entities that have a full qualified name matching the specified regular expression. Contrary to the NameFilter this search filter compares the fullqualified names (e.g. classes "com.example.Foo" or package "com.example" where NameFilter would see "Foo" respectively "example")

```
Synonyms:
   fqname

Parameters:
   <REGULAR EXPRESSION> (, <REGULAR EXPRESSION FLAGS>)*

Supported Entities:
   PACKAGE, CLASS, CONSTRUCTOR, METHOD, FIELD, ANNOTATION
```

Search all classes that have references to Class#forName calls (Reflection API).

```
select @name
from classes
where {
    exist call {
        etype('METHOD') && fqname('java.lang.Class#forName.*')
    }
}
```

7.14 Entity Type

A search filter which is able to filter entities by their type.

```
Synonyms:
   etype

Parameters:
   <ENTITY TYPE>

Supported Entities:
   ANNOTATION, CLASS, CONSTRUCTOR, FIELD, INSTRUCTION, LITERAL, METHOD,
   PACKAGE, RESOURCE
```

Search all methods accessing a field of type int.

```
select @name
from methods
where {
    exist call {
        etype('FIELD') && fieldtype('int')
    }
}
```

7.15 Modifier

A search filter which is able to filter entities by their modifier.

```
Synonyms:
   mod, modifier

Parameters:
   <MODIFIER>

Modifier:
   PUBLIC, PRIVATE, PROTECTED
   STATIC, FINAL, STRICTFP
   SYNCHRONIZED, VOLATILE
   TRANSIENT, NATIVE
   INTERFACE, ABSTRACT, ENUM, ANNOTATION, VARARGS
   SYNTHETIC, BRIDGE

Supported Entities:
   CLASS, METHOD, FIELD, CONSTRUCTOR
```

Search all native methods

```
select @name
from methods
where {
    modifier('NATIVE')
}
```

7.16 Literal Value

Search filter that accept literals that match a specified value and optional a specified type.

```
Synonyms:
   litval

Parameters:
   <VALUE> [, <CLASSENTITY>]

Supported Entities:
   LITERAL
```

Search all classes that have references to a (String) literal containing the text "icon". This filter uses regular expression so ".*icon.*" is specified in the example.

```
select @name
from classes
where {
    ex call {
        etype('literal') && litval('.*icon.*')
    }
}
```

7.17 Return Type

A search filter which is able to filter methods by their return value.

```
Synonyms:
   returntype

Parameters:
   <CLASSENTITY>

Supported Entities:
   METHOD
```

Search all methods that do return booleans.

```
select @name
from methods
where {
    exist call {
        etype('METHOD') && returntype('boolean')
    }
}
```

7.18 Metric

A search filter that filters entities based on metrics. Which metric is used can be passed to the filter beside the threshold to use.

The corresponding metric types are discussed in detail later on. Please refer to the following chapter.

```
Synonyms:
   metric

Parameters:
   <METRIC TYPE>, <THRESHOLD OPERATOR>, <THRESHOLD>
```

Search all classes that have more than 15 fields.

```
select @name
from classes
where {
    metric('FC','>','15')
}
```

7.19 Class Level

Search filter that can search for top level or nested classes.

```
Synonyms:
   level

Parameters:
   <LEVEL TYPE>

Level Type:
   OUTER (top level class)
   INNER (nested (inner) class defined inside another class)

Supported Entities:
   CLASS
```

Search all non-private nested (inner) classes.

```
select @name
from classes
where {
    !modifier('PRIVATE') && level('INNER')
}
```

7.20 Empty Set

This filter tests weather the considered feature set is empty.

```
Synonyms:
   empty, emptyset

Parameters:
   <ENTITY TYPE>

Supported Entities:
   ANNOTATION, CLASS, CONSTRUCTOR, FIELD, INSTRUCTION, LITERAL, METHOD,
   PACKAGE, RESOURCE
```

Search all interfaces that have fields but no methods. This is known as "Constant Interface Antipattern".

```
select @name
from classes
where {
    modifier('INTERFACE')
    && emptyset('METHOD')
    && !emptyset('FIELD')
}
```

7.21 Source Comment

This filter tests checks for matches of the passed regular expression in each class' comments. This filter will work on source code only.

```
Synonyms:
   <comment>

Parameters:
   <REGULAR EXPRESSION> (, <REGULAR EXPRESSION FLAGS>)*

Supported Entities:
   CLASS
```

Search all classes that have do not have an source comment named “@author”.

```
select @name
from class
where {
    !comment('@author')
}
```

7.22 Fieldtype

A search filter which is able to filter fields by their type.

```
Synonyms:
   fieldtype

Parameters:
   <CLASSENTITY>

Supported Entities:
   FIELD
```

Search class that defines a fields named serialVersionUID that is not final nor static nor of type long.

```
select @name
from classes
where {
    inherit('java.io.Serializable')
    && exist field {
        name('serialVersionUID')
        && {
            !modifier('FINAL')
            || !modifier('STATIC')
            || !fieldtype('long')
        }
    }
}
```

7.23 Public Default-Constructor

Search filter that accepts all public default constructors.

```
Synonyms:
   publicdefault, pubdef

Supported Entities:
   CLASS, CONSTRUCTOR
```

Search all classes that are annotated with com.example.LoadedByReflection that indicates that the annotated class will be loaded via Reflection API that does not have a public zero arg constructor. Trying to load such a class would end in a RuntimeException.

```
select @name
from classes
where {
    annotated('com.example.LoadedByReflection')
    && !publicdefault()
}
```

7.24 Duplicated Class

A search filter which is capable to find classes that reside multiple times on the classpath. If no parameter is given “FULL_QUALIFIED” mode is used.

```
Synonyms:
   duplicate

Parameters:
   [<DUPLICATION MODE>]

Duplication Mode:
   SIMPLE_NAME (ignore the package name and compare simple names)
   SIMPLE_NAME_IGNORE_CASE (same as previous but case-insensitive)
   FULL_QUALIFIED (classes that have completely identical names)

Supported Entities:
   CLASS
```

Search for classes that have identical simple names.

```
select @name
from classes
where {
    duplicate('SIMPLE_NAME')
}
```

7.25 Package

Filter that accepts all classnames in the same package than the passed package name. If no parameter is given “SAME_OR_SUB” mode is used.

```
Synonyms:
   packagename

Parameters:
   <PACKAGEENTITY> [, <PACKAGE_MODE>]

Package Mode:
   SAME (accept entites that resists in the passed package)
   SAME_OR_SUB (accept entites that resists in the passed or in a
                sub package)
   SUB (accept entites that resists in a sub package)

Supported Entities:
   PACKAGE, CLASS, CONSTRUCTOR, METHOD, FIELD, ANNOTATION
```

Search all classes that have references to classes in the package javax.swing.

```
select @name
from classes
where exist call {
    elementof class {
        packagename('javax.swing','SAME_OR_SUB')
    }
}
```

7.26 Throws Exception

A search filter which is able to filter methods and constructors by their thrown exceptions.

```
Synonyms:
   exception, throws

Parameters:
   <CLASSENTITY>

Supported Entities:
   METHOD, CONSTRUCTOR
```

Search all methods that declares to throw IOExceptions.

```
select @name
from methods
where {
    throws('java.io.IOException')
}
```

7.27 Signature

A search filter which is able to filter methods by their signatures. This is achieved by combining a ReturnType Filter and a ParameterList Filter.

```
Synonyms:
   signature, sig

Parameters:
   <CLASSENTITY> (, <CLASSENTITY>)*

Supported Entities:
   METHOD
```

Search all classes that define a main method.

```
select @name
from classes
where exist method {
    name('main')
    && signature('void','java.lang.String[]')
    && modifier('PUBLIC') && modifier('STATIC')
}
```

7.28 Shadowing Field

A search filter which is able to filter fields that are shadowed. A shadowed field is a field that exists in one of the class's super classes (fields cannot be overridden).

```
Synonyms:
   shadow, shadows, shadowing, isshadowed

Supported Entities:
   FIELD
```

Search all classes that contains shadowed fields

```
select @name
from classes
where {
    exist field {
        isshadowed()
    }
}
```

8 Metrics

This section explains the different metric types that are supported by the corresponding metric filter.

8.1 Annotation Count

Number of annotations for an entity.

Synonyms:
AC

8.2 Constructor Count

Number of constructors declared by a specific class.

Synonyms:
CC

8.3 Field Count

Number of fields declared by a specific class.

Synonyms:
FC

8.4 Method Count

Number of methods declared by a specific class.

Synonyms:
MC

8.5 Parameter Count

Number of parameters of a method. Very long signatures are often seen as a sign of poor design.

Synonyms:
PC

8.6 Public Field Count

Number of public fields variables use by the class.

Synonyms:
PFC

8.7 Public Constructor Count

Number of public constructors in this class.

Synonyms:
PCC

8.8 Public Method Count

Number of public methods in this class.

Synonyms:
PMC

8.9 Static Field Count

Number of static fields.

Synonyms:
SFC

8.10 Static Method Count

Number of static methods.

Synonyms:
SMC

8.11 Single Lines of Code

Number of code lines in the entity. Comments and empty lines are not counted. Large classes/methods are usually a sign of poor design. LOC are interpreted as instructions on bytecode level.

Synonyms:
SLOC

8.12 Method Overriding Count

Number of times the method is overridden in the complete class hierarchy.

Synonyms:
MOC

8.13 Name Character Count

Number of characters in a name.

Synonyms:
NCC

8.14 Parent Count

Number of parents of the passed class.

Synonyms:
PARC

8.15 Inner Class Count

Number of inner classes in this class.

Synonyms:
ICC

8.16 Imports Count

Number of imports of a class.

Synonyms:
IC

Alphabetical Index