# Canonical Filter Specification

Michiel Hendriks

# Canonical Filter Specification

Michiel Hendriks

Published 2009-03-11 14:49:47

# Reading Guide

This document discusses the canonical filter notation. It is constructed as a chapter in the Annotated Reference Manual (ARM). Everything from the past Compose* meetings concerning the canonical filters has been processed and is included in this document. The first section ( Chapter 1, *Canonical Filter Specification* ) should be complete and sound, anything that is missing or ambiguous should be considered a bug (please report this).

# Table of Contents

# List of Tables

# List of Examples

# Chapter 1. Canonical Filter Specification

The canonical filter specification allows a more fine grained definition for message matching and substitution actions than the classic notation. *This needs to be rewritten when it is made part of the ARM. Because at that point there is no "classic" notation, except for a section in the notes.*

## 1.1. Syntax

The basic syntax for the canonical filter elements is shown in Example 1.1, “Canonical Filter Definition Template” . The complete syntax is described by the grammar defined in Canonical Filter Grammar .

### Example 1.1. Canonical Filter Definition Template

```
FilterName : FilterType ( FilterArguments ) = ❶
 ( MatchingExpression ) ❷
 {
  Assignment; ❸
  Assignment;
 }
 cor❹ ( MatchingExpression )❺
```

❶ Declaration of a filter. It defines the filter types and optionally arguments to change the default behavior of the filter type.

❷ The expression that needs to evaluate to true for the message to be accepted by the current filter definition. The syntax for these expressions will be explained in Section 1.1.1, “Matching Expressions” .

❸ Assignments to be performed when the message matches. Assignments will be further discussed in Section 1.1.2, “Assignments” .

❹ The operators that links multiple filter elements in a filter definition. Currently there is only a single operator defined, namely the conditional *or* operator: `cor` .

❺ The assignment part is optional. When no assignments are needed this part can simply be omitted. An empty assignment part is also allowed: `( MatchingExpression ) {}` .

### Canonical Filter Grammar

| | | | |
|---|---|---|---|
| [1] | Filter | = | Identifier ':' FilterType '=' CanonFEs ; |
| [2] | CanonFEs | = | CanonFE { CanonFEOp CanonFE } ; |
| [3] | CanonFEOp | = | 'cor'; |
| [4] | CanonFE | = | '(' MatchExpr ')' [ '{' { Assignment } '}' ] ; |
| [5] | MatchExpr | = | AndExpr [ '\|' MatchExpr ] ; |
| [6] | AndExpr | = | UnaryExpr [ '&' AndExpr ] ; |
| [7] | UnaryExpr | = | ['!'] ( Operand \| '(' CompExpr ')' ) \| MatchExpr ; |
| [8] | Operand | = | 'True' \| 'False' \| Identifier; |
| [9] | CompExpr | = | BaseMessage CompOp CompRhs ; |
| [10] | CompOp | = | '==' \| '$=' \| '~=' \| '@='; |
| [11] | BaseMessage | = | 'target' \| 'selector' \| 'message.' ( 'target' \| 'selector' \| 'server' \| 'sender' \| 'self' ); |
| [12] | CompRhs | = | CompRhsLst \| CompRhsElm ; |
| [13] | CompRhsLst | = | '[' CompRhsElm { ',' CompRhsElm } ']' ; |
| [14] | CompRhsElm | = | FullyQualifiedName \| Literal \| 'inner' \| FilterModuleParameter \| BaseMessage ; |
| [15] | Literal | = | '"' ? string data ? '"' \| "'" ? string data ? "'"; |
| [16] | Assignment | = | AssignLhs '=' AssignRhs ';' ; |
| [17] | AssignLhs | = | 'target' \| 'selector' \| ( 'message.' Identifier ) \| ('filter.' Identifier); |
| [18] | AssignRhs | = | 'target' \| 'selector' \| ( 'message.' Identifier ) \| ('filter.' Identifier) \| FullyQualifiedName \| Literal \| 'inner' \| FilterModuleParameter ; |

## 1.1.1. Matching Expressions

A matching expression is an expression that evaluates to a boolean value. A matching expression defines if a message is accepted by the current filter. Only when the message is accepted the assignment part of the filter element will be processed.

The expression is a combination of condition references and compare statements combined with boolean operators like *or* ( `A | B` ) , *and* ( `A & B` ) and *not* ( `!A` ) . Sub-expressions can be created using parenthesis: `(A | !(B & C))` .

Condition references are the identifiers of the conditions defined in the current filter module. Besides condition references there are also the constants `true` and `false` .

The compare statements check if a message property matches with a given value. Compare statements consist of three parts: left hand side, matching operator, and the right hand side. The following matching operators have been defined:

Instance/name matching ( `==` )
: Returns true when both sides are the same instance (for objects) or have the same name (for selectors).

Signature matching ( `$=` )
: Returns true when the right hand side contains the signature given by the left hand side.

Compatibility matching ( `~=` )
: Returns true when both sides are compatible.

Annotation matching ( `@=` )
: Returns true if the program element on left hand side has the annotation given in the right hand side.

Information about the semantics of these operators in explained in Section 1.2.2.1, “Compare Statements” . To negate a compare statement it needs to be enclosed within parenthesis: `!(`**`target`** `==` **`inner`**`)` . The right hand side of the matching operator can contain a list of elements. The list is created by enclosing a comma separated list of elements with square brackets. With a list the compare statement will be `true` when the left hand side matches with at least one element from the list. For example, the following two matching expressions are identical:

```
(selector == 'foo' | selector == 'bar' | selector == ?quux)
(selector == ['foo', 'bar', ?quux])
```

### 1.1.2. Assignments

The assignment part contains entries where message and filter properties are assigned (new) values. The syntax is much like basic configuration files: `someVariable = newValue;` . The available variables and allowed values are discussed in Section 1.2.3, “Assignments” .

The assignment part of a filter element can contain zero or more assignments. The whole block can be omitted when no assignments are needed. Each assignment is terminated with a semicolon ( `;` ). The value only contains a new value, which is either a constant or an other value. Expressions are not possible as value.

## 1.2. Semantics

### 1.2.1. Types

There are a couple of different types used in both the matching expressions and assignment part. These types determine how certain elements work.
*Make a distinction between user input types (literal, FQN) and background types.*

Object
: Or rather, *instance* . Variables of the type object are: internals, externals, **`target`** , **`inner`** , **`message.target`** , **`message`**`.sender` , **`message`**`.self` , **`message`**`.server` New objects can be created through internals and externals in the current filter module, otherwise only the existing objects (target, inner, etc.) can be used.

Selector
: Only the variable **`selector`** (and therefore **`message.selector`** ) has this type. A selector is much like a method signature (it has a name, return type and argument types), except that the selector can contain wildcards for the return type and accepted arguments. *Is type information really useful here?*

Literal
: A literal is a string of characters enclosed by quotes (double quotes are allowed). For example: `'this is a literal'` or `"this is also a string"` . Literals are used *as is* , no standard (pre-)processing is performed on these values. In certain cases literals are converted to other types, for example in case of name matching with a selector.

Fully Qualified Name (FQN)
: A fully qualified name is resolved to a program element using the language model of the base program. A FQN is a combination of identifiers and periods, for example: `thisIsAFQN` or `this.is.also.a.fully.qualifiedName` . A FQN looks much like a reference to an internal/external/condition, they also use an identifier. Locally declared identifiers have precedence over fully qualified names. This means that if a identifier is encountered in either the matching expression or assignment part, it will first be looked up in the list of internals/externals/conditions of the current filter module and predefined variables (like: **`target`** , **`selector`** , **`message`**`.*` , etc.). If nothing was found the the identifier will be looked up in the language model This of course means that locally declared identifiers can hide certain program elements in the language model. No warning will be issued if hiding takes place. A warning or error will be issued when the identifier could not be resolved. If a error will be issued depends on where the identifier is used. For example in compare statements only a warning will be issued if an identifier

on the right hand side could not be resolved. But when it is on the left hand side, or used as a condition an error will be created.

Program Elements

Program elements cannot be entered directly in the source code. Program elements are entities in the language model of the base program which is created during the compilation process. Program elements are only indirectly available through a lookup, a lookup can be created through a FQN or through filter module parameters. A filter module parameter can contain a (set of) program elements when a selector is used in the filter module binding. Filter module conditions can also be used as a program element (in this case a method program elements), but only at places where program elements can be used. The following program element types are used in the canonical notations: Type (and therefore Class and Interface), Method, and Annotation.

**Example 1.2. Usage of different types**

```
filtermodule ExampleFM(?myParam) {
 internals
  foo : FQNExample.MyClass;
 conditions
  bar : foo.test();
 inputfilters
  f1 : Dispatch = ( true ) {
   filter.item1 = foo; ❶
   filter.item2 = bar; ❷
   filter.item3 = filters.item1; ❸
   filter.item4 = message.target; ❹
   filter.item5 = FQNExample.MyClass; ❺
   filter.item6 = 'hello world!'; ❻
   filter.item7 = ?myParam; ❼
  }
}
```

❶ `filter.item1` will contain a value of type *object* . `foo` refers to an internal, which is an instance of the class `FQNExample.MyClass` .

❷ `bar` is a condition registered in this filter module, a condition is a reference to a method program element, therefore `filter.item2` will contain a value of the type *program elements* . More specifically a method program element.

❸ `filters.item1` contains a typo, by mistake an *s* was added to the word `filter` . As a result `filters.item1` is regarded as a FQN and will be resolved in the language model of the base program. When no such element can be found a warning will be issued, and the resulting value is *void* . *Not regarded as FQN, just not resolved to a filter property. Why even include this example?*

❹ `message.target` is an object, therefore the value of `filter.item4` will be of type object.

❺ The current filter module does not contain a declaration with the name `FQNExample.MyClass` , so this entry is considered to be a fully qualified name and is resolved to a program element. This program element resolved to a type, not to an object. An object is an instance of a type, and instances are not present in the language model.

❻ `'hello world!'` is a literal, `filter.item6` will contain exactly that value (without the quotes).

❼ `?myParam` is a filter module parameter. Filter module parameters do not include a type definition. Therefore the value of `filter.item7` will not be known until this filter module has been instantiated. The value will most likely be a program element or a literal.

### 1.2.2. Matching Expression

The matching expression is constructed from a combination of boolean statements and boolean operators. There are three boolean operators. The & ( *and* ) and | ( *or* ) operators expect a boolean statement on each side. The third boolean operator is the *not* operator `!` which expects a single boolean statement on the right side. There are four types of boolean statements:

Constant
: Either `True` or `False` .

Condition
: These are the conditions defined in the filter module. These conditions can only be evaluated at runtime because they require the execution of the method to get its boolean value.

Compare statement
: A statement that compares two entities according to a given rule set. The result would be either true or false. More about the compare statements can be found in Section 1.2.2.1, “Compare Statements” . In order to avoid confusion compare statements must be placed in a compound statement to negate them.

Compound statement
: A compound statement is nothing more than a matching expression enclosed within parenthesis.

### 1.2.2.1. Compare Statements

Below is an overview of all compare operators and what arguments types they support. The left hand side of the operator can only contain a selector or object variable. The right hand side be a single element or a list of elements that are allowed for the used operator. This list may contain any combination of allowed right hand parameter types. For example: **`target`** `~= [myList, java.util.AbstractList]`

The allowed variables on the left hand side are restricted to the following set: **`target`** , **`selector`** , **`message.`** *property* . Only message properties that contain a selector or object type are legal. If the property contains any other data the compare will always result in false. The following message properties are always safe to be used: **`message.target`** , **`message.selector`** , **`message.`**`server` , **`message.`**`self` . The **`message.`**`sender` is theoretically safe, however but in practice this is not always the case (it could be unset).

**Note**

During static analysis no information about the sender is known, therefore it will always be unset. In the Compose* runtime component, information about the sender is known when the message was send from a non-static context.

All variables that can be used in the left hand side can also be used on the right hand side. Additionally the following variables are also allowed: **`inner`** , internals, externals, conditions, filter module parameters . Constants like literals and fully qualified names are also allowed.

**Table 1.1. Compare Operators**

| Operator | Left hand side | Right hand side | Comments |
|---|---|---|---|
| == | Object | Object | *Instance matching.* Returns true when the left hand side and the right hand side are the same instance. |
| | Selector | Literal | *Name matching.* The name part of the selector is matched with the literal. Possible values of the return and argument types in the selector are ignored. |
| | | Program Element (Method) | *Method instance matching.* The selector will be resolved to methods in the current target that match the selector (respecting wildcards for the return and arguments fields). This operator will return true if the method on the right hand side is in the list of resolved methods. This form is usually used in conjunction with filter module parameters that contain method program elements. These program elements are usually very specifically selected, |

| **Operator** | **Left hand side** | **Right hand side** | **Comments** |
|---|---|---|---|
| | | | and therefore this comparison is performed to the latter. To compare on only the signature of a method use the method signature matching listed below. |
| | | Selector | *Instance matching.* Will be true when both selectors have identical properties for name, return type and selector. This form is rarely used, it can only be used when a selector is stored in a message property by an earlier filter. |
| `$=` | Selector | Type Program Element | *Signature matching.* This operator returns true if the type program element on the right hand side contains a method that has a signature that matches the selector. The signature is matched against the name, return type and arguments set in the selector. If the return or argument fields of the selector contain wildcards they will be ignored. This form is usually used with filter module parameters. |
| | | Object | *Signature matching.* Just like the signature matching above, the type of the object is used as the type program element. This is the most commonly used form of signature matching. |
| | | Fully Qualified Name | *Signature matching.* The fully qualified name is resolved to a type program element. After that the earlier mentioned signature matching rules apply. |
| | | Program Element (Method) | *Method signature matching.* This will return true if the selector matches the signature of the given method. Wildcards in the selector are respected. This is a less strict method comparison than the method instance matching. |
| `~=` | Object | Type Program Element | *Compatibility matching.* Returns true if the object on the left hand size could be assigned to a variable of the type in the right hand side. This means that the type on the right hand side is in the parent list of the type of the object on the left hand side. Or, in case the type in the right hand side is an interface, the object's type implements the interface on the right hand side. This operator basically does the following: `matches if (lhs instance of rhs)` |
| | | Object | *Compatibility matching.* Just like the compatibility matching above, the type of the object on the right hand side is used as the type program element. |
| | | Fully Qualified Name | *Compatibility matching.* The fully qualified name is resolved to a type program element. After that the compatibility matching between an object and type applies. |
| | Selector | Literal | *Compatibility matching.* The literal is used to look up all methods in the current target that have a name matching the literal. The selector is matched against the signatures of these methods according to the signature matching rules (the name was already given). If any method signature matches with the selector this compare statement will be true. If the current target does not have a method with the given name, or no signature matches, it will result in false. |
| | | Method Program Element | *Compatibility matching.* Matches if the return type and arguments given in the selector match those of the method given in the right hand side. Wildcards in the selector are respected, in this case the selector would match any method. |
| | | Selector | *Compatibility matching.* Matches if the return and type arguments are the same (wildcards are respected). So bascially, only the name is ignored. |
| `@=` | Object | Annotation Program Element | *Annotation matching.* This operator will match when the type of the object on the left hand side contains the annotation given on the right hand side. |

| Operator | Left hand side | Right hand side | Comments |
|---|---|---|---|
| | | Fully Qualified Name | *Annotation matching.* The fully qualified name is resolved to an annotation program element after which it is compared along the same rules as listen above. |
| | Selector | Annotation Program Element | *Annotation matching.* Using the selector a selection is made of methods in the current target. If any of these methods contains the annotation given in the right hand side this operator will return true. In any other case it will return false. |
| | | Fully Qualified Name | *Annotation matching.* The fully qualified name is resolved to an annotation program element after which it is compared along the same rules as listen above. |

## 1.2.3. Assignments

In the assignment part certain variables can be given a new value. Not all variables can be assigned a new value. Below is an overview of all variables. The assignment part is only executed when the message was accepted according to the matching expression.

**Table 1.2. Variable assignments**

| Variable | Writable | Accepted Value Type | Comment |
|---|---|---|---|
| **`target`** / **`message.target`** | Yes | Object | A target always points to an object instance, therefore it can only accept objects as new value. So on the right hand side of the assignment operator can be a reference to an internal, external, **`inner`** , **`message`**`.self` , **`message`**`.sender` , **`message`**`.server` . |
| **`selector`** / **`message.selector`** | Yes | Literal | The literal is converted to a selector with the return value and arguments set to a wildcard. This is the most common assignment for selectors, and the only way to directly set a new value for a selector in the source code. |
| | | Program Element (Method) | The program element is converted to a selector. This selector is more specific than the selector created from the literal. This selector has complete information regarding the return type and accepted arguments. |
| | | Selector | The selector variable has the type selector, so it can always be assigned a different selector value. Because selectors can not be directly created assigning a new selector can only be done by first saving a selector in a message property. |
| **`inner`** | No | | |
| **`message`**`.sender` | No | | |
| **`message`**`.server` | Yes | | *Write this* |
| **`message`**`.self` | No | | |
| External / Internal / Condition | No | | |
| **`message.`** *property* | Yes | Any type | Where *property* is any identifier other than target, selector, sender, server, or self. Message properties will be carried over to the next filter. They can contain any type, how these properties are interpreted is up to the filter type using them. Information on what filter types make use of these message properties can be found in Chapter 2, *Filter Types* |

| Variable | Writable | Accepted Value Type | Comment |
|---|---|---|---|
| | | | . Assigning message properties never results in a warning or error. However, a filter type can produce an error or warning when an unexpected value is set. |
| **`filter.`** *property* | Yes | Any type | The filter arguments can contain any value, it is up to the filter type to interpret them. Chapter 2, *Filter Types* provides information about the filter arguments that can be set for certain filter types. Assigning values to filter arguments results in a warning that these argument are not used. Failing to set a required argument, or assigning an incompatible type, results in an error. *Last 2 lines are not correct/wrong* |

### 1.2.3.1. The Message and Filter Structures

In the assignment part there are two special structures: **`message.`** *property* and **`filter.`** *property* . Where property is an identifier specifying which property to set. The message structure contains some reserved names, but other than that all identifiers can be used.

The message structure can be used to pass certain information on to other filters. The other filters must ofcourse be aware of these properties and how to interpret these values.

The filter structure provides local access to the filter arguments. It can be used to override the filter arguments set in the filter definition for a specific filter element. Other than that it is identical to the filter arguments. The preferred form is to set the filter arguments after the filter type in the filter definition and not in the assignment part.

**Example 1.3. Setting Filter Arguments**

```
f1 : Exception(excetion='MyException') = (true)
// or, alternatively
f1 : Exception = (true) { filter.exception = 'MyException' }
```

How the filter arguments are used depends on the used filter type. See Chapter 2, *Filter Types* for more information. Filter arguments are not carried on to the next filter, they are specific for the current filter. To pass information to other filters you will need to use message properties.

## 1.2.4. Filter Module Parameters

Filter module parameters can be used in the matching expressions and assignment parts. This section will explain how the parameters are used. As a reminder, the values for the filter module parameters are provided by an external source during the binding of a filter module. The value types are not strictly defined, it could be a literal but also a program element. A single filter module parameter is prefixed with a single question mark `?singleFMP` , and a filter module parameter list is prefixed with two question marks `??fmpList` . Theoretically the filter module parameter list can contain different value types, but this should be avoided. *Allow this? Best to only allow a single type*

### 1.2.4.1. Matching Expression

Filter module parameters can only be used on the right hand side of compare statements. A filter module parameter list will be regarded just like a list construction. When a filter module parameter is used in a list all values of the filter module parameter are added to the list. For example: we have the following filter module parameters: `?fmp = 'foo'; ??lst = ['bar', 'quux'];` And the following matching expression:

```
(selector == ?fmp) |
(selector == ??lst) |
(selector == ['item', ?fmp]) |
(selector == ['item', ??lst]) |
(selector == ['item', ?fmp, ??lst])
```

This will internally be interpreted as:

```
(selector == 'foo') |
(selector == ['bar', 'quux']) |
(selector == ['item', 'foo']) |
(selector == ['item', 'bar', 'quux']) |
(selector == ['item', 'foo', 'bar', 'quux'])
```

The matching operators expect certain types to be on the right hand side. However, filter module parameters can contain any type. When the filter module parameters are processed only compatible values will be used, the other values are silently ignored. When the filter module does not contain a single compatible type a warning will be issued, in this case the compare statement will always evaluate to `false` . See Section 1.2.2.1, “Compare Statements” for information about what value types the compare operators expect.

### 1.2.4.2. Assignments

A variable can only be assigned a single value, not a list of values. When a parameter list is used as source for the new value, the first compatible element will be used. When the filter module parameter does not contain a compatible value a error or warning is issued and the assignment statement is discarded. In case of assignments to the target and selector values an error is created. For assignments to other message properties or filter arguments a warning is issued. This can subsequently result in an error when an assignment is required by the filter type.

**Warning**

The order in which elements are listed in parameter lists is nondeterministic. A second compile of the exact same code can result in a different value to be used in the assignment. A warning is issued when a parameter list contains more than one compatible value.

## 1.3. Examples

**Example 1.4. Example of canonical filters**

```
filtermodule FM1 (?type, ??methods)
{
 internals
  foo : ?type;
 conditions
  isActive : foo.isActive();
 inputfilters
  toFoo : Dispatch = (
    (!isActive & selector == "activate")
    | (isActive & selector == ??methods & selector $= foo)
   )
   {
    target = foo;
   };
  raiseError : Exception = (selector == ??methods)
   {
    filter.exception = "Example.Exception.InactiveException";
   }
}
```

*... explain*

## 1.4. Conversion

Below are some examples how to convert the classical notation to the canonical definition.

```
Dispatch = { [*.bar] *.quux }
```

```
Dispatch = (selector == 'bar') { selector = 'quux'; }
```

The classic notation does not force a matching on the target, so the target is not included in the expression. In the classic notation `bar` and `quux` are literals and not variables, in the canonical notation they should be written as literals.

```
Dispatch = { True ~> [*.bar] *.quux }
```

```
Dispatch = (!(selector == 'bar')) { selector = 'quux'; }
```

The disable operator is easily substituted with a not operator in the matching expression.

```
Dispatch = { [*.bar] foo.* }
```

```
Dispatch = (selector == 'bar') { target = foo; }
```

`foo` is declared as either and internal or external.

```
Dispatch = { <foo.*> foo.* }
```

```
Dispatch = (selector $= foo) { target = foo; }
```

Signature matching is performed between a selector and an object.

```
Dispatch = { c1 | c2 => [*.bar] }
```

```
Dispatch = ((c1 | c2) & selector == 'bar')
```

The assignment part can often be empty because the filter does not rely on assignments to be made. When the assignment part is empty is can also be omitted in the canonical notation.

```
Dispatch = { [*.bar] *.quux, [*.quux] *.bar }
```

```
Dispatch = (selector == 'bar') { selector = 'quux'; }
  cor (selector == "quux") { selector = "bar"; }
```

This example shows how to define multiple filter elements with a single filter.

```
Dispatch = { {[*.bar], [*.baz]} *.quux }
```

```
Dispatch = (selector == 'bar' | selector == 'baz') {
 selector = 'quux'; }
// or
Dispatch = (selector == ['bar', 'baz']) { selector = 'quux'; }
```

Matching different selectors can be done by using either a list of selectors or by writing the complete matching expression. Both notations have exactly the same effect.

```
Before = { [*.bar] foo.quux }
```

```
Before = ( selector == "bar" ) { filter.target = foo;
 filter.selector = "quux"; }
```

The Before filter executes a defined method, it does not substitute anything in the message. Previously the substitution part was used to provide what method should be called.

```
Exception(exception="java.lang.UnsupportedMethod") = { [*.bar] }
```

```
Exception = (selector == "bar") {
  filter.exception = "java.lang.UnsupportedMethod";
 }
// or
Exception(exception="java.lang.UnsupportedMethod") =
 ( selector == "bar" )
```

With canonical filters the filter arguments can be set in the assignment part. But they can also be given as arguments directly. In the latter case the argument will apply to all filter elements, while in the former case it will only apply to the current filter element.

## 1.5. Comments

### 1.5.1. Message Substitution

Assigning new values to the target and selector variables results in a direct message substitution. Substitution of the message used to be defined by the filter type in the old filter notation. This was mostly due to the lack of ways to provide information to the filter types like *Meta* or *After* .

With the canonical notation it will be possible to send a new message before the current message is dispatched and substitute the message's destination. For example in Example 1.5, “Before and substitute” the method `fooWasCalled` of the internal `someInternal` will be called. And the current message will be given the new selector `bar` . Which eventually might lead to the execution of the method `bar` .

**Example 1.5. Before and substitute**

```
flt : Before = ( selector == 'foo' ) {
 selector = 'bar';
 filter.target = someInternal;
 filter.selector = 'fooWasCalled';
}
```

## 1.6. FAQ

# Chapter 2. Filter Types

This section explains all default filter types which have been defined in Compose*. Not all filter types are available in all platforms.

Each filter type contains a behavior description. Below is explained what each behavior element means.

Filter activity
: Defines where in the message flow the filter has an activity. There are four options: Accept-call, Reject-call, Accept-return, Reject-return . When the matching expression matches the accept-* flow is taken. Otherwise the reject-* flow is used. The message is in the *-call flow when it has not be placed in the return flow. See the message flow behavior.

Message flow
: This defines how this filter (when executed) changes the message flow. There are three possible values. *Continue* means that the message will continue to the next filter element. *Return* means that the message will enter the return flow. And *exit* means that the message will not be processed by any filters.

Resource operations
: This defines what operations the filter type will perform and on what resources.

**Important**

The resource operation model as used here is subject to change. This is due to the introduction of the canonical filter notation.

## 2.1. Dispatch/Send

When the message is accepted it will dispatch the message to set target and selector. After the dispatch the message will enter the return flow. The dispatch and send filter are almost identical in their behavior, except that the send filter updates the *sender* value of the message to current object (which would be identical to *message.self* ). The send filter is commonly used as output filter, where the dispatch filter is commonly used as input filter.

### 2.1.1. Behavior

Filter activity
: Accept-call

Message flow
: Return

Resource operations
: args.read; message.dispatch; return.write; message.return;

## 2.2. Error

**Important**

The error filter has been deprecated by the exception filter.

The error filter throws an error when the message is not accepted.

### 2.2.1. Behavior

Filter activity
: Reject-call

Message flow
: Exit

Resource operations
: args.discard; return.discard; message.discard;

### 2.2.2. Arguments

| Name | Type | Required | Description |
|---|---|---|---|
| exception | fully qualified name | no | The exception type to throw. When it is not set, or could not be resolved to a valid exception type the default `ErrorFilterException` will be thrown. |
| | literal | | |

## 2.3. Before

This filter dispatches a new message to a defined target and selector. After the dispatch of the new message the filter processing will continue with the old message.

### 2.3.1. Behavior

Filter activity
: Accept-call

Message flow
: Continue

Resource operations
: ...

### 2.3.2. Arguments

| Name | Type | Required | Description |
|---|---|---|---|
| target | object | at least one must be set | This defines the message that should be dispatched. The method identified by this new message must have a signature that either takes a no arguments at all, or a single argument that is compatible with the join point context that is passed to it. |
| selector | literal | | |
| | selector | | |
| | method program element | | The method program element will be used to look up a method within the set target that is compatible with the given method program element. |

### 2.3.3. Examples

**Example 2.1. Log vault access**

```
filtermodule BeforeExample {
 internals
  logger : security.AccessLogger;
 inputfilters
  vaultAccess : Before = ( selector == ['open', 'close',
   'enter', 'leave'] )
  {
   filter.target = logger;
   filter.selector = 'log';
  }
}
```

This example will send a message to the access logger whenever a certain action on the vault is performed.

## 2.4. After

This filter behaves much like the before filter. Except that this filter dispatches a new message in the return flow of a message. This means that the new message will never be dispatched when the current message never enters the return flow.

### 2.4.1. Behavior

Filter activity
   Accept-return

Message flow
   Continue

Resource operations
   ...

### 2.4.2. Arguments

See Section 2.3.2, “Arguments” .

## 2.5. Meta

The meta filter is a very powerful, but also dangerous, filter type. It provides the functionality to modify the message in any way at runtime. The meta filter calls a user defined method with as argument the current message. Static analysis of the filters is not possible when meta filters are used, this is because the behavior of the meta filter is unknown (unless code analysis is performed on the called method).

**Warning**

Try to avoid using the meta filter. Try to do as much as possible with the other filter types. When the standard filter types do not offer the needed functionality, and when this functionality is needed at multiple locations it might be better to define a custom filter type. See ... for more information. *Add information for custom filter types*

### 2.5.1. Behavior

Filter activity
    Accept-return

Message flow
    Unknown, continue by default

Resource operations
    ...

### 2.5.2. Arguments

| Name | Type | Required | Description |
|---|---|---|---|
| target | object | at least one must be set | ... |
| selector | literal | | |
| | selector | | |
| | method program element | | ... |

## 2.6. Exception

The exception filter throws an error when the message is accepted. The exception filter is a complement of the error filter.

### 2.6.1. Behavior

Filter activity
    Accept-call

Message flow
    Exit

Resource operations
    args.discard; return.discard; message.discard;

### 2.6.2. Arguments

| Name | Type | Required | Description |
|---|---|---|---|
| exception | fully qualified name | no | The exception type to throw. When it is not set, or could not be resolved to a valid exception type the default `ErrorFilterException` will be thrown. |
| | literal | | |

## 2.6.3. Examples

**Example 2.2. Prevent removing of list elements**

```
filtermodule AddOnlyLists {
 inputfilters
  noremove : Exception =
   (selector == ['clear', 'remove', 'removeAll'])
   {
    filter.exception =
     'java.lang.UnsupportedOperationException';
   }
}
```

This example, when superimposed on a list type, removes the functionality to remove items from the list. Items can only be read and added.