Detailed Argument and Return Info for FunctionName

Tracker:	http://jira.codehaus.org/browse/GEOT-3569
Tagline:	Modifying FunctionName to support detailed info about argument/return parameters.

- Description
- Status
- Tasks
- API Changes
  - FunctionName
  - FilterFactory2
- Existing Functions
- Variable Arguments
  - Sanity check - Existing Variable Length Functions

Children:

Description

Currently the FunctionName interface only represents some limited information about a function in terms of its arguments:

- number of arguments
- argument names

This proposal is to extend the list to include:

- argument types
- min/max argument occurrences
- return type

The latest version of the OGC filter specification (FES 2.0) requires that the above information be included in a filter capabilities document when describing supported functions.

The current patch implementing this proposal can be found here.

Status

Voting has not started yet:

- Andrea Aime: +1
- Ben Caradoc-Davies +0
- Christian Mueller +0
- Ian Turton +0
- Justin Deoliveira: +1
- Jody Garnett (current OSGeo representative): +1
- Simone Giannecchini +0

Tasks

This section is used to make sure your proposal is complete (did you remember documentation?) and has enough paid or volunteer time lined up to be a success

	no progress	✅	done	❌	impeded	⚠️	lack mandate /funds/time	❓	volunteer needed

1. ✅ API changed based on BEFORE / AFTER
2. ✅ Update default implementation
3. ✅ Update the user guide
4. ✅ Update the existing function implementations

- GEOT-3569 Add detailed information about Function arguments and return type to FunctionName

API Changes

FunctionName

BEFORE:

```
interace FunctionName {

  int getArgumentCount();

  List<String> getArgumentNames();
}
```

AFTER:

New interface:

```
package org.opengis.parameter;

interface Parameter {
    String getName();

    InternationalString getTitle();
    InternationalString getDescription();

    Class<T> getType();

    Boolean isRequired();
    int getMinOccurs();
    int getMaxOccurs();

    T getDefaultValue();
}
```

Existing Parameter class to extend new interface:

```
package org.geotools.data;
class Parameter implements
org.opengis.parmameter.Parameter {
  ...
}
```

Modify FunctionName:

```
interace FunctionName {

    /** derived from getArgumets() */
    int getArgumentCount();

    /** derived from getArgumets() */
    List<String> getArgumentNames();

    List<Parameter<?>> getArguments();

    Parameter<?> getReturn();
}
```

FilterFactory2

AFTER:

New method:

```
FunctionName functionName(String name,
List<Parameter<?>> args, Parameter<?> ret);
```

Existing Functions

In order to take advantage of the new api function implementations will have to be migrated in order to declare the required information. However since there are numerous function implementations out there a gradual migration path has to be put in place.

Since all functions today already declare either an argument count, or a set of argument names all the required information can be derived from that alone. For instance of a function simply declares that it takes 2 arguments we derive the following:

- arg1(type=Object.class,min=1,max=1)
- arg2(type=Object.class,min=1,max=1)
- return(type=Object.class)

If a function declares two arguments named "foo" and "bar" we derive the following:

- foo(type=Object.class,min=1,max=1)
- bar(type=Object.class,min=1,max=1)
- return(type=Object.class)

Moving forward it is expected that new function implementations will declare arguments explicitly. And that existing function implementations will do so as the need arises. The following is an example of how to do so. Consider a function named "strIn" that determines if a "String" is "in" a list of specified strings. The function arguments are the string being tested for existance, a boolean determining whether case should be taken into account, and finally a variable list of candidate string values.

```
public class StringInFunction extends
FunctionImpl {

    static FunctionName NAME =
functionName("strIn", "result:Boolean",
"string:String", "matchCase:Boolean",
"values:String:1,");

    public StringInFunction() {
        setName("strIn");
        functionName = NAME;
    }

    @Override
    public Object evaluate(Object object) {
    ....
}
```

The `functionName` method is static and avaialble to all subclasses of `FunctionImpl` and `FunctionExpressionImpl`.

Variable Arguments

Since arguments declare min/max occurrences it is possible to have variable arguments (ie. arguments that take an arbitrary number of values) and optional arguments.

Obviously this presents challenges when attempting to dispatch a function call using the current mechanism in which filter functions are invoked. So to maintain backward compatibility and prevent requiring a different

mechanism for invoking filter functions we adopt the same convention as the Java language in that only the last argument to a function may be variable or optional. This convention is enforced with the help of a convenience method on the function base class FunctionImpl.

Sanity check - Existing Variable Length Functions

Note **existing** variable length functions from app-schema and Symbology Encoding (concatenate, interpolate) has been defined without regards to the above convention.

```
public static final FunctionName NAME =
functionName("Concatenate",
"text:String,2,");
```

Interpolate represents a worst case senario where repeating pairs are required as shown below:

```
/**
     * Make the instance of FunctionName
available in
     * a consistent spot.
     */
    public static final FunctionName NAME;
    static {
        Parameter<Object> lookup = new
Parameter<Object>("lookup",Object.class,1,1)
;
        Parameter<Object> table= new
Parameter<Object>("data value
pairs",Object.class,4,-1);
        Parameter<String> mode = new
Parameter<String>(
            "mode", String.class,
            Text.text("mode"),
            Text.text("linear, cosine or
cubic"),
```

```
                true,1,1,
                MODE_LINEAR,
                new
KVP(Parameter.OPTIONS,Arrays.asList(new
String[]{MODE_LINEAR, MODE_COSINE,
MODE_CUBIC}))
        );
        Parameter<String> method = new
Parameter<String>(
                "method",String.class,
                Text.text("method"),
                Text.text("numeric or
color"),
                false,0,1,
                METHOD_NUMERIC,
                new
KVP(Parameter.OPTIONS,Arrays.asList(new
String[]{METHOD_NUMERIC, METHOD_COLOR}))
        );

        NAME = new
```

```
FunctionNameImpl("Interpolate", lookup, table, mode, method);
    }
```

✅ If you are interested the above interpolate function can be used as in this CQL example (to repalce a lot of code you often use Rules for):

```
Interpolate(POP_CNTRY,
0,'#ffffff',1000000,'#cccccc',10000000,'#9999aa',1000000000,'#55
55ff',''linear','color')
```

Note the "table" parameter must have a minimum of 4 values to form an interpolation range. The above example has 4 pairs forming three smooth gradients. Choosing a different MODE would result in a smoother shift between the gradients.

In addition note the use of Parameter.OPTIONS to indicate arguments that are restricted to a small set of allowable values.