Conditions

Table of contents

1. Overview

The package org.expasy.jpl.commons.base.cond provides powerful mechanisms to test conditions.

It provides a condition builder and objects to create expressions over conditions.

`Condition` is largely propagated in all JPL filters.

2. Conditions

A condition over a specific class of objects is to be tested against a specific `operand` and `operator` given at building time. Then, the satisfiability of the condition is tested against a valid object.

2.1. Generic Parameters of ConditionImpl.Builder

1. <T> the first type stands for the object class to *T*est.
2. <V> the second type stands for the *V*alue type really tested.

If data types are not compatible given the `operator`, you will have to define a stub that handles the access from <T> to <V> and give it to the `accessor` method.

For example, in the following condition `new ConditionImpl.Builder<List, Integer>(5).accessor(stub).build()` we have provided a stub that returns an Integer from the List parameter.

Here is a definition of a condition over `Double`s equivalent to the following equality 'x = 50.0'.

```
// the default operator is 'equals'
Condition<Double> condition =
        new ConditionImpl.Builder<Double, Double>(50.0).build();
```

The condition is then evaluated on x (here '50.01') that eventually return false.

```
Assert.assertFalse(condition.isTrue(50.01));
```

Here is a less strict version equivalent to 'x ~ 50.0':

```
condition =
        new ConditionImpl.Builder<Double, Double>(50.0).operator(
          OperatorApproxEquals.newInstance(0.01)).build();
```

```
Assert.assertTrue(condition.isTrue(50.09));
```

Sometimes the object to test is not of the same class type that the value to test with.

In a first case, it is a classical wishing situation when it imply specific set operators like `belongs` or `contains`:

```
// A simple set
final Set<Double> s1 = new HashSet<Double>();
s1.add(54.3);
s1.add(23.);

// Another set
final Set<Double> s2 = new HashSet<Double>();
s2.add(34.);
s2.add(23.);

final Condition<Set<Double>, Set<Double>> cond =
        new ConditionImpl.Builder<Set<Double>, Set<Double>>(s2).operator(
                OperatorContains.newInstance()).build();

Assert.assertFalse(cond.isTrue(s1));
```

In other cases, like noticed above, we have to provide an `accessor` method that makes the link to the object to test the predicate:

```
import import org.apache.commons.collections15.Transformer;

List<Integer> l = Arrays.asList(1, 2, 3);

// this stub reduces a List to its size (Integer)
Transformer<List, Integer> getListSize =
        new Transformer<List, Integer>() {

        public Integer transform(final List l) {
                return l.size();
        }
};

// the condition with the path correctly set
Condition<List, Integer> hasThreeElements =
        new JPLCondition.Builder<List, Integer>(3).accessor(
                getListSize).build();

Assert.assertTrue(hasThreeElements.isTrue(l));
```

3. Expression Tree over Conditions

Expression trees represent code in a tree-like data structure, where each node is an expression over `Condition`. It is then possible to create conditions over condition:

```
ConditionInterpreter<Double> engine =
        ConditionInterpreter.newInstance();

// the engine has an internal symbol table
// creation of a condition assigned to "c1"
engine.addCondition("c1", new ConditionImpl.Builder<Double,
        Double>(0.).operator(OperatorGreaterThan.newInstance()).build());

// creation of a condition assigned to "c2"
engine.addCondition("c2", new Condition.Builder<Double,
        Double>(10.).operator(OperatorLowerThan.newInstance()).build());

// creation of a complex condition over "c1" and "c2"
Condition<Double> condition = engine.translate("c1 & c2");

Assert.assertTrue(condition.isTrue(4.));
Assert.assertFalse(condition.isTrue(14.));
```

4. Caveat due to java reified-less generic

If object and value classes are not compatible because the user did not specify the `path` or did not set the correct operator, a `IllegalStateException` is thrown at testing time and not at building time :-(

We could have added a mandatory parameter to the builder to redundantly set the object class (the `T` in Condition<T, V>) and compare it with the value type at building time. Unfortunately, it is not possible to write some generic class like Collection. For example:

> `... you can't write class literals for generic types like List<String>.class, or test if an object is an instanceof List<String>, or create an array of List<String> ( in Reified Generic for Java).`

For all these reason, we had to test it at testing time and not at building time.