

## **Wikiprint Book**

**Title: Native Resolution and Opaque Links**

**Subject: Java Path Finder - projects/jpf-mango/opaque**

**Version: 8**

**Date: 02/21/2013 06:20:35 PM**

## Table of Contents

<b>Native Resolution and Opaque Links</b>	<b>3</b>
Static data and unsoundness	3
DiceRolls example	3
Nutshell	5
Conclusion	5

## Native Resolution and Opaque Links

Mango now has the ability to treat unspecified methods as "black boxes". A black box returns "opaque" output. If a black box is passed references, then the output heap has opaque heap values tied to these references. Rule-based reasoning about opaque return values and opaque heap values is still in its infancy, but this part of the "Mango brain" is expected to grow quickly. What is in place, Mango now is the ability to track opaque values via nested hyper-link display of state components.

### Static data and unsoundness

The following example illustrates how static data access can result in unsound modeling, or not, as the case may be. This is a fact of life which must be faced head on.

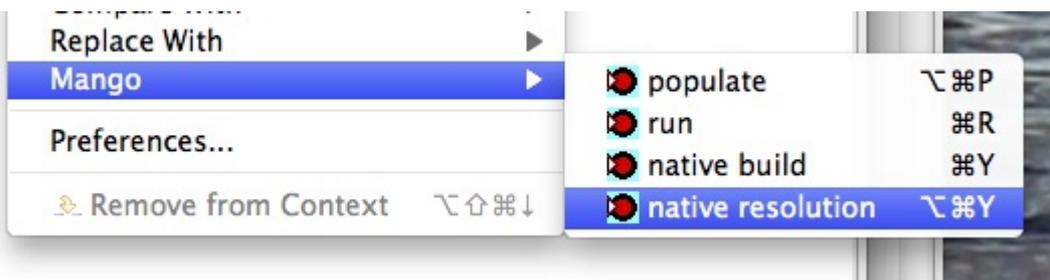
### DiceRolls example

```
public class DiceRolls
{
    public static void main(String args[])
    {
        int firstRoll;
        int secondRoll;
        int total;
        int loopnum;
        Random generator = new Random ();

        System.out.println ("Dice 1 Dice 2 Total ");
        for (loopnum=1;loopnum<=5;loopnum++)
        {
            //generating numbers
            firstRoll=generator.nextInt(6)+1;
            secondRoll=generator.nextInt (6)+1;
            total = firstRoll+secondRoll; //calculating sum
            System.out.println (firstRoll+" "+secondRoll+" "+total);
        }
    }
}
```

Consider the following source code.

Notice there are two unspecified methods, the Random constructor and Random.nextInt(). Instead of building the method population for main(), apply "native resolution" from the context menu focussed on main().



The native resolution command inventories all unspecified methods which the target depends upon, and builds a "native specification" for each such method. In our case, the upshot is that nextInt() will return an "opaque" number. To see how this works, consider the specification for the loop inside of main:

The opaque values returned by `nextInt()` appear as the links "opaque return value" and "opaque return value\_1". At this point, the alert reader must ask the question, "since the input for `nextInt` is 6 in both cases, and `nextInt` is modeled functionally, how can it be that the two output values are distinct". Excellent question! To see what is really going on here, let's trace the "history" of "opaque return value\_1" via hyper-linking. Here is the result of clicking on the link:

```
modules.firstYearCode.DiceRolls.main([Ljava/lang/String;)V.loops.loop at 43 for
(loopnum=1;loopnum<=5;loopnum++).parameters.opaque return value_2
<integralValue> opaque return value_1
opaque return value
o
( pstate 'identity )
```

Not much information here, just telling us that our link resulted from composing the previous opaque value with the identity state. So let's dig a little deeper by clicking on "opaque return value"

```
modules.firstYearCode.DiceRolls.main([Ljava/lang/String;)V.loops.loop at 43 for
(loopnum=1;loopnum<=5;loopnum++).parameters.opaque return value_1
<integralValue> opaque return value
opaque return value
o
( pstate ( <pathName> 'modules 'firstYearCode 'DiceRolls "main([Ljava/lang/
String;)V" 'loops "loop at 43 for (loopnum=1;loopnum<=5;loopnum++)" ) 'root 1 )
```

This is more interesting. We now know that our link resulting from some previous opaque value composed with (pstate ... 1). Ok, keeping that thought in mind, lets click one more time:

```
modules.java.util.Random.nextInt(I)I.parameters.opaque return value
<integralValue> opaque return value
opaque return value of java.util.Random.nextInt(I)I
```

Well, we have finally bottomed out. This is telling us that nextInt() returned an opaque value, end of story. So lets go back to that (pstate ... 1) expression. The key "pstate" just means "permanent state", which can be looked up somewhere if there is a really compelling reason to do so. For our purpose, it suffices to observe that if we repeat this entire experiment with the other opaque link of the loop specification, we end up with (pstate ... 0), as demonstrated below.

```
modules.firstYearCode.DiceRolls.main([Ljava/lang/String;)V.loops.loop at 43 for
(loopnum=1;loopnum<=5;loopnum++).parameters.opaque return value
<integralValue> opaque return value
opaque return value
o
( pstate ( <pathName> 'modules 'firstYearCode 'DiceRolls "main([Ljava/lang/
String;)V" 'loops "loop at 43 for (loopnum=1;loopnum<=5;loopnum++)" ) 'root 0 )
```

### Nutshell

In a nutshell, the links differ because the input state to the two different calls to nextInt() really were not the same after all. At the source code level, it looks like 6, but the actual input state for the second call has been tainted by the output of the first call, and so really is not the same. Now you might say this is an accident which could have easily come down against us if we were in fact modeling a genuine function with no side-effect resulting from a peek at the static area, and you would be right.

### Conclusion

In conclusion, rule-based reasoning about opaque values is open for business, and interesting!