# Runtime Verification

## Lecture 3 : Monitoring with AspectJ

http://www.runtime-verification.org/course

May 8, 2008

This third lecture illustrates through examples how to write monitors in AspectJ. This note in addition contains 2 new assignments in addition to the assignment presented in lecture 2. The plan for all assignments in the course is described below.

### Installation

In order to write monitors, we need various forms of sets and maps. the `java.util` library provides the basic `HashMap` and `HashSet`. In addition we need various combinations of these being *weak* (garbage collector removes entries when the monitored application no longer refers to them) and/or *identity* (entries are compared with == and not by the `equals()` method)[1]. Again, `java.util` offers `WeakhashMap` and `IdentityHashMap`. We further need `IdentityHashSet`, `WeakIdentityHashSet`, and `WeakIdentityHashMap`. These are downloadable from the website. They require the apache commons collections library. The instructions are below.

1. Install the apache commons collections package:

   `http://commons.apache.org/collections/`

   This package is used to implement the following classes.

2. Install the following (from the course website):

   (a) `IdentityHashSet`

   (b) `WeakIdentityHashSet`

   (c) `WeakIdentityHashMap`

### Reading

1. Same as lecture 2 (AspectJ documentation)

[1] There are 8 combinations, but we only need 7, ignoring `WeakHashSet`.

**Plan for Assignments During Course**

Below are 2 new assignments numbered 2-3 in addition to the assignment (number 1) presented in lecture 1. These are all the problems that will be presented during the course. Each of the 3 problems should ideally be specified in each of the following 5 specification formalisms: AspectJ (lectures 2+3), JavaMOP regular expressions (lectures 4+5), JavaMOP context free grammars (lecture 6), JavaMOP temporal logic (lecture 7), and in the rule-based system RuleR (lecture 8). Hence a total of max 15 specifications. The assignments are handed in after the end of the course on June 6, as a small report of 10-15 pages, explaining your experience, and a zip file containing the specifications. It is not required that you do all the 15 specifications, but try to do as many as you can. The important thing is to get an idea of the field.

**Assignment 1**

The assignment for lecture 2.

**Assignment 2**

Consider a stack implementing the following interface:

```
public interface StackInterface {
    public void push(Object t);
    public Object pop();
    public Object top();
    public boolean isEmpty();
    public int length();
}
```

Assume an implementation of this interface:

```
public class Stack implements StackInterface {
  ...
}
```

The stack may be used as follows:

```
public class Main {
  public static void main(String[] args) {
    Stack s = new Stack();
    s.push("1");
    s.push("2");
    System.out.println("pop: " + s.pop());
    System.out.println("pop: " + s.length());
    System.out.println("length: " + s.pop());
  }
}
```

Write an aspect that checks the correctness of the Java stack. That is, checks that:

1. *a pop is not invoked before an initial push.*
2. *a stack returns the last pushed object when pop is invoked.*
3. *a stack returns the last pushed object when top is invoked (and no pop happened between push and top).*
4. when something is pushed onto a stack, the stack size is increased by 1, and similarly that it is decreased by 1 by a pop.

**Assignment 3**

Consider the Java API `java.util.Collections` on the Java doc website:

`http://java.sun.com/j2se/1.5.0/docs/api/`

This package provides a selection of static methods for operating collections. Amongst these are a set of methods `synchronized...(..)` that each takes a collection and returns a thread-safe collection. For example:

```
public static Collection<T> synchronizedCollection(Collection<T> c)
```

The Java documentation states: "*Returns a synchronized (thread-safe) collection backed by the specified collection. In order to guarantee serial access, it is critical that all access to the backing collection is accomplished through the returned collection.*". An example of calling this method is the following:

```
public class Main {
  public static void main(String[] args) {
    Set set = new HashSet();
    Collection c = Collections.synchronizedCollection(set);
    c.add(1);
    System.out.println("c contains 1: " + c.contains(1));
  }
}
```

Write an aspect that checks the following policy: *If an object is synchronized via Collections.synch*(...), only the returned synchronized collection should be used in the remainder of the program.*.