ListenerList Transformation Discussion

This page exists so that we can discuss the on-going @ListenerList implementation.

Basic Usage

This example shows the most basic usage of the @ListenerList annotation. The easiest way to use this annotation is to annotate a field of type List and give the List a generic type. In this example we use a List of type MyListener. MyListener is a one method interface that takes a MyEvent as a parameter. The following code is some sample source code showing the simplest scenario.

```
interface MyListener {
    void eventOccurred(MyEvent event)
}
class MyEvent {    def source
    String message

    MyEvent(def source, String message) {
        this.source = source
        this.message = message
    }
}
class MyBeanClass {
    @ListenerList
    List<MyListener> listeners
}
```

- + addMyListener(MyListener) : void - This method is created based on the generic type of your annotated List field. The name and parameter type is are based on the List field's generic parameter.
- + removeMyListener(MyListener) : void- This method is created based on the generic type of your annotated List field. The name and parameter type is are based on the List field's generic parameter.
- + getMyListeners() : MyListener[] - This method is created based on the generic type of your annotated List field.The name is the plural form of the List field's generic parameter, and the return type is an array of the generic parameter.
- + fireEventOccurred(MyEvent) : void - This method is created based on the type that the List's generic type points to. In this case, MyListener is a one method interface with an eventOccurred(MyEvent) method. The method name is fire[MethodName of the interface] and the parameter is the parameter list from the interface. A fireX method will be generated for each public method in the target class, including overloaded methods.

Gets turned into:

```
public class MyBeanClass {

    @groovy.beans.ListenerList
    private List<MyListener> listeners

    public void addMyListener(MyListener listener) {
        if (listener == null) { return }
        if ( listeners == null) { listeners = [] }
        listeners.add(listener)
    }

    public void removeMyListener(MyListener listener) {
        if (listener == null) { return }
        if ( listeners == null) { listeners = [] }
        listeners.remove(listener)
    }

    public MyListener[] getMyListeners() {
        def __result = []
        if ( listeners != null) {
            __result.addAll(listeners)
        }
        __result as MyListener[]
    }

    public void fireEventOccurred(MyEvent p0) {
        if ( listeners != null) {
            def __list = new ArrayList(listeners)
            for (listener : __list ) {
                listener.eventOccurred(p0)
            }
        }
    }
}
```

ListenerLists for classes and wide interfaces

The ListenerList generates a fireX method for every public method on the target. For instance, this class:

```
interface TestTwoMethodListener {
  void eventOccurred1(TestEvent event)
  void eventOccurred2(TestEvent event)
}


class TestClass {
     @ListenerList
     List<TestTwoMethodListener> listeners
}
```

Has these two methods generated:

```
    public void fireEventOccurred1(TestEvent event) {
        if ( listeners != null) {
            java.util.ArrayList<E extends java.lang.Object> __list = new
java.util.ArrayList<E extends java.lang.Object>(listeners)
            for (java.lang.Object listener : __list ) {
                listener.eventOccurred1(event)
            }
        }
    }

    public void fireEventOccurred2(TestEvent event) {
        if ( listeners != null) {
            java.util.ArrayList<E extends java.lang.Object> __list = new
java.util.ArrayList<E extends java.lang.Object>(listeners)
            for (java.lang.Object listener : __list ) {
                listener.eventOccurred2(event)
            }
        }
    }
```

Rejected Alternatives

- We considered allowing @ListenerList to be a Class/Type annotation. The benefit was terseness. You don't even need to declare a field in that case. However, we decided it brings too much complexity. We are searching for ways to simplify this, and removing this option reduces the complexity with no real loss in functionality.
- We considered adding fire* methods based on the declared constructors of the Event class. This means that if the Event class has 5 constructors then there would be 6 fire events on the target class. This is too complex, too verbose, and not part of the bean spec. To simplify we removed it. We can add it back in later if we want.
- Considered have the event type be part of the annotation, but the simplest things to do is make it required on the field declaration.