

Java Properties and Events

Language Feature Suggestions (for Java.Next...)

Joe Nuxoll
Senior Staff Engineer
<http://blogs.sun.com/joe>
joe.nuxoll@sun.com

November 2005 (originally authored in 2004)

Following are a few basic language concepts that I believe should have been put into the Java language several years ago. These types of concepts raise the language up a level of abstraction so that APIs can be designed as component libraries rather than as class libraries. These patterns are prevalent in competing software platforms, and the lack of these have caused, in my opinion, the majority of the usability/developability gap that Java experiences today.

Having these concepts in the language itself (rather than a separate oft-ignored API: JavaBeans) promotes the design of new APIs with developer usability in mind. This will help **everything** in terms of developer usability a lot more than just the syntax sugar alone. We will start to see new technology areas addressed by libraries of components instead of collections of interfaces and classes. This allows a developer to get up to speed on new technologies much faster, as they apply the same usage patterns across the board on the different technologies (using the same model). Component-based development **really** helps IDEs visualize software development in a consistent manner.

1. Properties

The basic idea here is to elevate the notion of an encapsulated/dynamic field (with the capability of get/set side-affects) into the language itself. A language-level property is very similar to a public field, only it allows the class (component) author to associate programmatic side-affects with access to the field.

The primary push-back on this concept has been the notion of “preservation of transparency”. $C=A+B$ means only one thing in Java, well most of the time anyway (except for `java.lang.String`, etc). This language design rule of thumb works great if you approach the Java language from the point of a view of a language designer / compiler implementer, but does not provide any benefit for the vast majority of users of the language. Most developers experience Java as a language that they are trying to solve business problems with – not as a pure software science playground. $C=A+B$ **should** do the **right** thing according to the developer of the classes represented by A, B, and C. In the case of `java.lang.String` this is pretty straightforward, and extremely useful.

I do not believe that we should open the gates for operator overloading to the developer – as that does produce some very complex code that is not really necessary for most tasks.

A user of a software component does not need to know if a particular field is in fact a field – or if it is a property. Whether or not it causes side-effects when it is accessed or set is generally not a concern of the user of the software component. The software component author can make that decision as appropriate. Encapsulation is one of the most powerful features of object oriented development from a pure business standpoint.

1.1 Simple (singleton) Properties - Usage

```
JButton button1 = new JButton();
button1.backgroundColor = Color.RED;
button1.font = new Font("Arial", 12, Font.BOLD);
button1.foregroundColor = button1.backgroundColor.darker();
```

Note that this pattern would appear to be **exactly** the same in Java source, XML, JSP/JSF EL, or any place where the form `object.property` is referenced. It also represents a 1-to-1 relationship between what a development tool user sees in a property inspector in a visual designer and what they see in the source code when they have to produce some business logic in an event handler. It may seem simple to those of us to look for a `setBackground(...)` method to set a background property, but that is totally foreign (and absurd) to a new Java developer that is looking for the word “background” in the code because that's what they saw on the property inspector. This is especially true when they use the `object.property` notation in their JSP, JSF, or XML files today. This would make them all the same.

1.2 Simple (singleton) Properties - Declaration

```
public class JButton {

    // simple protected member field
    protected Color _bgColor;

    /**
     * Javadoc here is associated with the whole
     * property - not separately with a setter and
     * getter method
     */
    public Color bgColor get {
        return _bgColor;
    } set { // implied 'value' param of type Color
        _bgColor = value;
        repaint(100);
    };

    /**
     * This property is completely dynamic (no field)
     */
    public Color fgColor get {
```

```

        return _bgColor.darker();
    } set { // implied 'value' param of type Color
        // very weird example, yes...
        _bgColor = value.lighter();
        repaint(100);
    };
}

```

The general form of this declaration syntax is very similar to a member field declaration, but includes an optional `get` and `set` method with optional bodies. This introduces two new token location-specific keywords, “`get`” and “`set`” which may occur after a member field declaration, before the assignment and/or statement ending semicolon.

```

[public|protected|private|]
  TYPE propName
  [get [{ ... return TYPE; }]]
  [set [<implied:(TYPE value)>{ ... }]] [assignment op];

```

1.3 Simple (singleton) Properties – Variations (optional method bodies)

It might be nice to be able to declare a property and “inherit” some default method bodies if we just want to use them as a field. This may make things overly complicated, so it might make sense to just require the method bodies when declaring properties. This is here for discussion, as it may be convenient for the developer to be able to do this sort of thing...

```

// public field of type Color (same as today)
public Color fgColor;

// public property of type Color (default setter/getter)
// this appears to a user of this class as a
// functionally identical field or property to the
// above declaration but has a default field and
// default accessor methods instead just producing a
// field
public Color fgColor get set;
--> // produces this bytecode...
    private Color $fgColor;
    public Color $fgColor$get() {
        return this.$fgColor;
    }
    public void $fgColor$set(Color value) {
        this.$fgColor = value;
    }

// read-only (final) public field
// must be initialized via constructor codepath
public final Color fgColor;

```

```

// read-only public property (default getter)
// must be initialized via constructor codepath
// this appears to a user of this class as a
// functionally identical field or property to the
// above declaration but has a default field and a
// default get method instead just producing a field
public Color fgColor get;
--> // produces this bytecode...
    private Color $fgColor;
    public Color $fgColor$get() {
        return this.$fgColor;
    }

// write-only public field
// super weird but maybe used as a 'hit' counter?
public Color fgColor set;
--> // produces this bytecode...
    private Color $fgColor; // how can we block access?
    public void $fgColor$set(Color value) {
        this.$fgColor = value;
    }

```

1.4 Thoughts about bytecode generation – Dynamic Properties

```

// recursive reference to property inside of method
// body - assigns to implied storage field (if one is
// produced)
public Color fgColor get set {
    this.fgColor = value;
    repaint(100);
};
--> // produces this bytecode...
    private Color $fgColor;
    public Color $fgColor$get() {
        return this.$fgColor;
    }
    public void $fgColor$set(Color value) {
        // reference to 'this.fgColor' was replaced with
        // the default storage field inside this method
        this.$fgColor = value;
        repaint(100);
    }

// fully dynamic property (no default field created)
public int rows get {
    return _storage.size();
} set {
    _storage.setRowCount(value);
    fireStorageMuckedWithEvent();
}

```

```

        repaint(100);
    };
--> // produces this bytecode...
    public int $rows$get() {
        return _storage.size();
    }
    public void $rows$set(int value) {
        _storage.setRowCount(value);
        fireStorageMuckedWithEvent();
        repaint(100);
    }

```

1.5 Indexed Properties

What about indexed properties? Good question... First of all, indexed properties are not very highly used in Java. The defacto-standard IDE model for dealing with indexed properties has generally been to ignore the indexed version of the getter/setter methods and simply deal with the regular type accessor (same as a regular property). Indexed properties could easily be replaced by regular properties pointing to collection types, or better yet: generic collection types.

1.6 Property Change Events (bound/constrained properties)

In short, these are very rarely used. PropertyChangeEvents are fired for changes in different property values, yes – but the bound/constrained concepts are all but ignored. There is nothing in the above API that would prevent this from working as it does today.

On the other hand, with properties in the language itself, it might make sense to move property change events down into a reflection layer similar to JPDA. Perhaps something like:

```

// construct a ReflectionListener (operator class)
java.lang.reflect.ReflectionListener rl =
    new java.lang.reflect.ReflectionListener();

// set the object property to the object you wish to
// listen to...
rl.object = jButton1;

// add a listener method to the events of the RL
// (see below events section for syntax)
rl.propertyMutated += this.jButton1_propertyMutated;
rl.fieldMutated += this.jButton1_fieldMutated;
rl.fieldRead += this.jButton1_fieldRead;
rl.methodCalled += this.jButton1_methodCalled;

```

Or, this could be put into java.lang.Object itself:

```

// java.lang.Object includes a propertyChanged event
// that is automatically called when a property is
// mutated
jButton1.propertyChanged +=
    this.jButton1_propertyChanged;

```

2. Events

The basic idea here is to elevate the notion of an encapsulated/dynamic event into the language itself. This technique leverages the above property technique using a special type for the events themselves.

Events, which are widely adopted in Java APIs (phew!) allow a developer to reuse components and associate custom behavior with them on an instance by instance basis. There is no need to subclass a component in order to specialize its behavior. A developer only needs to register an event listener, and execute whatever code they wish when an event condition is met, and thus their listener is notified.

2.1 Events – Usage

```

public void init() {
    JButton jButton1 = new JButton();
    // this example is a direct field/property assign
    jButton1.onMouseClicked =
        this.jButton1_mouseClicked;

    // multicasting might use an incremter
    jButton1.onMouseClicked +=
        this.jButton1_mouseClicked;

    // retaining event sets (r/o property on the button)
    jButton1.mouseEvents.mousePressed =
        this.jButton1_mousePressed;

    jButton1.mouseEvents.mousePressed +=
        this.jButton1_mousePressed;
}

// 'this.jButton1_mouseClicked' is this method on the
// same form with a "matching" method signature...
public boolean jButton1_mouseClicked(
    Component comp, Point clickPoint, int clickCount) {
    ...
    return true;
}

```

The above implies that a method name could be used as a pointer to a method on an instance... better known as a “closure”. This also implies the potential operator

overloading of += and -= to register/unregister an event listener. I think this would aide in code readability, but should not be interpreted as a request for operator overloading! Just as we do with java.lang.String today, we could special case these event closure objects.

2.2 Events – Declaration

```
// this Closure object is the "magic" type representing
// an event property - this could live in java.lang or
// java.lang.reflect?
public class java.lang.Closure {
    public Method method;
    public Object instance;
    public Object invoke(Object[] args)
        throws InvocationException {
        ...
    }
}
```

Certain keywords could allow the component developer to declare events without needing to explicitly create new sub-types of closure. Following are several brain-storm syntax ideas about how this might be declared. Note that there is a general onXXX pattern for event naming.

```
// using an event keyword - method declaration same
// as everywhere else in Java (nice - my favorite)
public event boolean onMouseClicked(
    JButton button, Point clickPoint, int clickCount);

// maybe specify if its multicast?
// or should they all just be considered multicast?
public multicast event boolean onMouseClicked(
    JButton button, Point clickPoint, int clickCount);

// or a new [] syntax to mean "method pointer"
// note that param names are included for IDE code-
// insight information - this is more like a field
// declaration
public [boolean: JButton button, Point clickPoint,
    int clickCount] onMouseClicked;

// a no-arg, void return method pointer
// (syntax looks silly?)
public [void:] somethingHappened;

// looks a little better using an event keyword
public event void somethingHappened();

public [Object: int feet, int inches] onJump
```

```

        throws VaultException;

// read-only (final) 'mouseEvents' field
public final MouseEvents mouseEvents =
    new MouseEvents();

// read-only 'mouseEvents' property (same thing)
public MouseEvents mouseEvents get = new MouseEvents();

```

2.3 Invoking Events

Invoking an event is as simple as calling the method pointer in the class itself. If there are listeners attached, they will each be invoked in succession. If there are no listeners, this does nothing.

```

protected void internalMethodThatFiresEvent() {
    ... stuff happens ...
    Point point = new Point(...);
    int count = ...
    // directly invoking a method pointer member
    // variable - calls all registered listeners
    this.onMouseClicked(this, point, count);

    MouseEvent me = new MouseEvent(...);
    // calls all listeners
    this.mouseEvents.mousePressed(me);
}

```

2.4 Event Bundles (instead of listener interfaces)

Current events require a set of method in a listener interface (callback indirection). This, though confusing overall, does have the useful advantage of logical grouping of events into sets. This could be achieved using event bundles instead of listener interfaces:

```

// instead of listener interfaces - an event 'bundle'
// this one uses the [] syntax
public class MouseEvents {
    public [void: MouseEvent me] mousePressed;

    // cracked events are easier for developers
    public [void: Component source, Point point,
        int count] mousePressed;

    public [void: MouseEvent me] mouseReleased;

    public [void: MouseEvent me] mouseClicked;
}

// instead of listener interfaces - an event 'bundle'

```

```
// this one uses the event keyword syntax
public class MouseEvents {
    public event void mousePressed(MouseEvent me);

    // cracked events are easier for developers
    public event void mousePressed(Component source,
        Point point, int count);

    public event void mouseReleased(MouseEvent me);

    public event void mouseClicked(MouseEvent me);
}
```

I should point out that the main point of all of this is to simplify the model for events so that developers are more inclined to use them or more importantly new developers can understand them. This simplicity promotes the use of events in new API designs, and thus simplifies future APIs.