Juki Event Pattern(JEP)

The JIK Event pattern is an extension of Adobe Flash Player's Native Event mechanism. All normal rules still apply however the JIK Event pattern allows for API restrictions, event interfaces, forwarding, silent events and controlled capture phase.

I am a big fan of Signals/Slots or the Messaging Paradigm used by Cocoa. Since neither can really be implemented flawlessly in AS3(without some major potential performance issues down the road). I came up with a hybrid Event Pattern solution I call the "Juki Event Pattern(JEP)".

The major difference between JEP and the one adopted by Java/Flash is the fact that the events are exposed as public getter functions. This leads to whole new worlds of interface-required events. However, the pattern maintains full backwards compatibility with the built-in event pattern. JEP is an extension of the basic event pattern and requires a IEventDispatcher for the purpose of carrying out the actual dispatching.

A simple interface is used to "connect" a listener function or event to another event. Yes, I said you can connect one event to another. Also because the events are exposed as a public getter you could essentially wrap another event from a asset class and expose it as your own.(think about that for a moment). I have provided three connection options: "connect", "connectAsSilent", and "connectAsPassThrough". Respectfully the operations are basic connect: where the event object is passed to the listener, where nothing is passed to the listener, where all arguments of the dispatch() method are passed to the listener.

Full disconnect. As requested, I have provided the ability to disconnect from all listeners. I also maintain a list of the listeners in a soft-linked Dictionary object. This was also another request that I ended up finding useful for internal uses.

Dispatch log. I made a decision early on that my primary problem with the default event pattern is that there is no way to tell if any specific event has already dispatched. The problem here is sometimes a event only fires once. Say, If it represents a "download complete". It fires once, if you fail to catch it, your screwed. Normally, I'd say, check your code try to catch it earlier. What if that wasn't possible tho. I maintain a array of the last arguments passed to the event and if it has dispatched yet. Sadly, for memory purposes I have chosen not to capture dispatches outside of the JEP. I may implement this as a "use as your own performance risk" feature later.

There are two main classes used by the JIK Event Pattern.

Basic Usage Rules:

1) High Priority is 100, Low is 0. If you need a larger range, you can set priority to < 998 as 998 and 999 are internal priorities and should be respected.
2) The public getter must start with 'ev' for easier identification during code-hinting and the Outline. Example: evDisplayUpdate, evDestroy. I know this looks weird, but seriously hitting 'ev' on your keyboard and getting a full list of the events in code-hinting is pretty sweet.

JKEvent

Is the base entry point for all event actions. This is extended for added public properties the same way Event is extended for IOEvent, ProgressEvent, etc. Note that the constructor argument 'type' is not compatible with addEventListener unless you set 'unique' to false, otherwise it will be mutated with a unique number to ensure that it cannot be emitted via addEventListener. This is generally the case when you may or may not have collision issues with native events of the same name. If you have multiple events with one listener you can type check the event type or use event.type.split(":")[0] to get the original type you used when creating the event. For example new JKEvent("progress") may return a type like "progress:10203". The second argument of the JKEvent constructor requires a IEventDispatcher. If this is set to 'null' then the JKEvent will be self-scoped with its own unique EventDispatcher.

JKNativeEvent

Is a compatibility entry point for events that are dispatched directly from the EventDispatcher using the same 'type' defined in the JKNativeEvent constructor. Note, this does not apply to 'type' used in JKEvent as that is simply a debugging helper. JKNativeEvent encapsulates JKEvent.

Basic Usage

JKEvent can connect to other JKEvents. JKNativeEvent can connect to JKEvents and EventDispatchers. JKNativeEvent can also 'watch' EventDispatchers. Connect is to Watch as push is to get.

```
public var costo:Sprite = new Sprite();
public var nombe:JKEvent = new JKEvent("random", this );

private var _ADDED_TO_STAGE:JKNativeEvent = new JKNativeEvent("addedToStage",
this );
public function get ADDED_TO_STAGE():JKNativeEvent
{
          return _ADDED_TO_STAGE;
}

// in Constructor or elsewhere
_ADDED_TO_STAGE.connect( costo );
_ADDED_TO_STAGE.connect( nombe );
```

That may not be the best example, but I hope you get the idea. When JKNativeEvent ADDED_TO_STAGE is dispatched natively it will automatically dispatch the JKEvent 'nombe' and 'costo'. The same event object sent to JKNativeEvent will be correctly forwarded without modification.

API Restrictions / Controlled Capture

The JIK Event Pattern requires that all Events must be accessed via a public getter function or as a public variable if you want to limit some subclassing flexibility.

When encapsulating or delegating a control, you may want to expose it's event as your own.

```
public function get COMPLETE():JKEvent
{
          return _delegate.COMPLETE;
}
```

If you wish to require all listeners of the delegate or encapsulated class to fire first you may forward the event to the one in your subclass. (Setting priority of the forward to a negative number will make it the last to fire.)

```
// This should be in the class constructor
_delegate.COMPLETE.connect( _COMPLETE, false, -1 );

private var _COMPLETE:JKEvent = new JKEvent('complete', this );
public function get COMPLETE():JKEvent
{
	return _COMPLETE;
}
```

If you need to perform a task before your exposed event fires then.

```
// in Constructor
_delegate.COMPLETE.connect( performTaskFirst, true );

protected function performTaskFirst():void
{
	// Perform Task
	_COMPLETE.dispatch();
}

private var _COMPLETE:JKEvent = new JKEvent('complete', this );
public function get COMPLETE():JKEvent
{
	return _COMPLETE;
}
```

Event Interfaces

Since all Events can now be exposed as getters, they can be required by interfaces.

```
public interface LoaderEvents
{
	function get COMPLETE():JKEvent;
	function get OPEN():JKEvent;
	function get PROGRESS():JKEvent;
	function get NET_STALL():JKEvent;
	function get NET_TIMEOUT():JKEvent;
	function get ERROR():JKEvent;
}
```

Application Programming Interface Overview

Class - JKEvent

```
+ function get arguments():Array
+ function get hasDispatched():Boolean
+ function get tag():uint
+ function setOwner( owner:IEventDispatcher ):void
```

```
+ function connect( listener:*, useCapture:Boolean=false, priority:int=0,
  useWeakReferences:Boolean=false ):void
+ function connectAsSilent( listener:*, useCapture:Boolean=false, priority:int=0,
  useWeakReferences:Boolean=false ):void
+ function connectAsPassThrough( listener:*, useCapture:Boolean=false,
  priority:int=0, useWeakReferences:Boolean=false ):void
+ function disconnect(method:*):void
+ function dispatch(...args):void
+ function dispose():void
+ function disconnectAllConnections():void
+ function get isDisposed():Boolean
+ function get listeners():Array
```

Class - JKNativeEvent

```
+ function get hasDispatched():Boolean
+ function get tag():uint
+ function setOwner( owner:IEventDispatcher ):void
+ function get lastDispatched():Event
+ function connect( listener:*, useCapture:Boolean=false, priority:int=0,
  useWeakReferences:Boolean=false ):void
+ function connectAsSilent( listener:*, useCapture:Boolean=false, priority:int=0,
  useWeakReferences:Boolean=false ):void
+ function connectAsPassThrough( listener:*, useCapture:Boolean=false,
  priority:int=0, useWeakReferences:Boolean=false ):void
+ function disconnect(method:*):void
+ function watch( source:IEventDispatcher, useCapture:Boolean=false,
  priority:int=0, useWeakReferences:Boolean=false ):void
+ function dispose():void
+ function get isDisposed():Boolean
```

Note about hasDispatched: *hasDispatched() is generally used for a event that will only fire once. If you know it will fire once you can check to see if you were too late to catch it before it fired. arguments() is the array of arguments passed in to dispatch(.. args). In JKNativeEvent, lastDispatched() is a copy of the last fired event. Watch() is the exact opposite of forward -- this JKNativeEvent watches a IEventDispatcher. All properties, Target, CurrentTarget, etc are properly populated and available from the inherited API.*