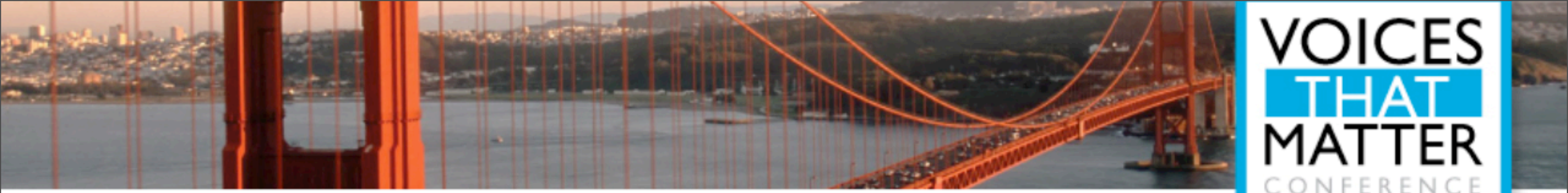
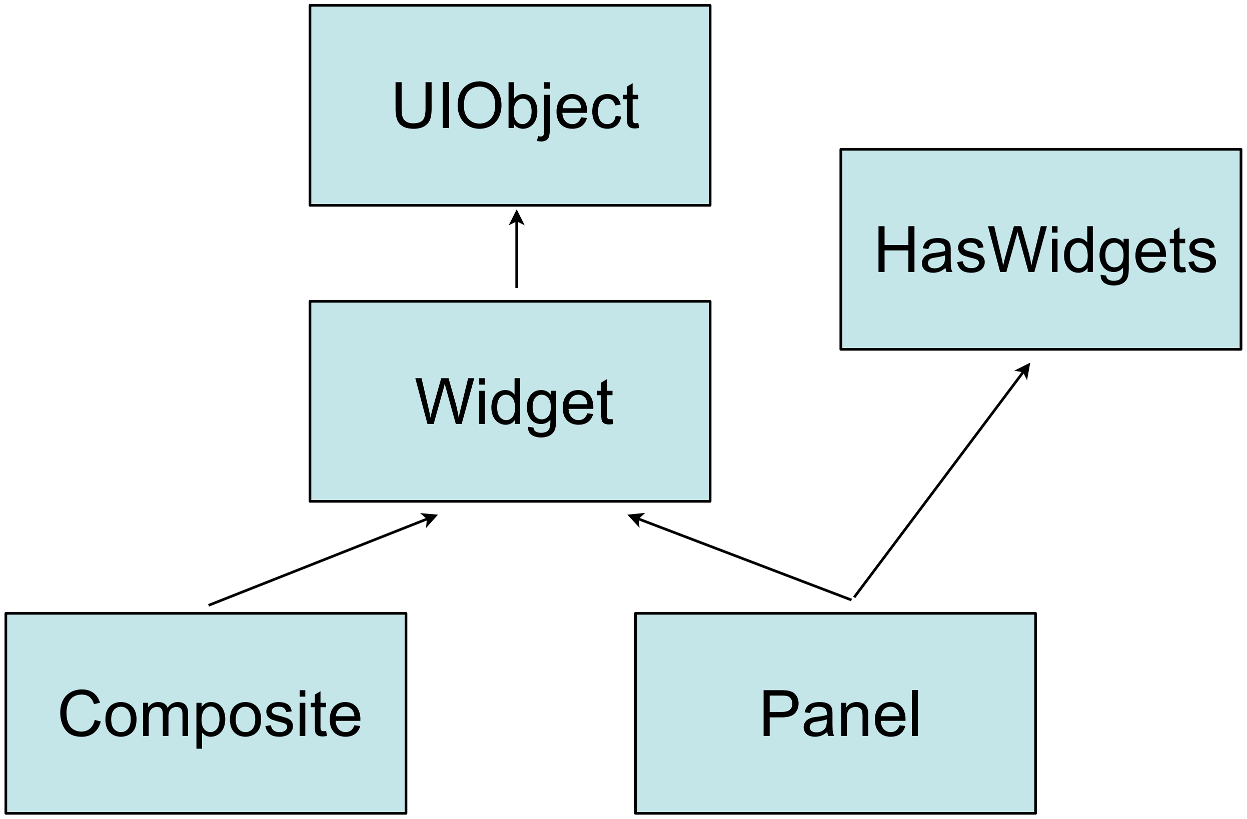


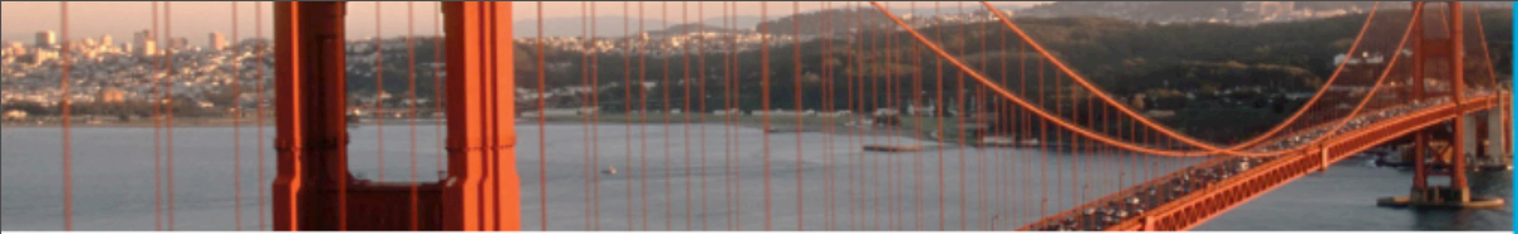
VOICES THAT MATTER CONFERENCE

GOOGLE WEB TOOLKIT

creating widgets

the usual suspects





why the distinction?

UIObject ≠ Widget ?

UIObject is a simple element wrapper.
Widget adds event handling and the ability to be added to panels.

composites

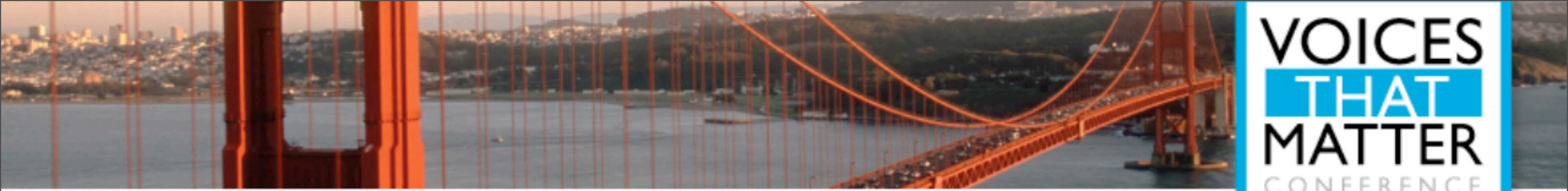
easy way to create widgets and panels

allows you to wrap existing widgets

Often better than extending an existing widget
- hides parts of the interface you don't want to expose
This allows you to control your invariants

```
class MyComposite extends Composite {
  private VerticalPanel p = new VerticalPanel();
  private Label l = new Label();
  private TextBox tb = new TextBox();

  public MyComposite() {
    p.add(l);
    p.add(tb);
    initWidget(p);
  }
}
```

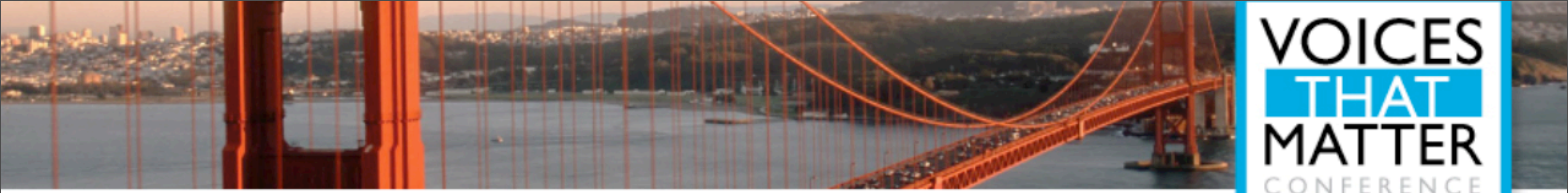


when to use composites

whenever your widget can be expressed in terms of existing widgets

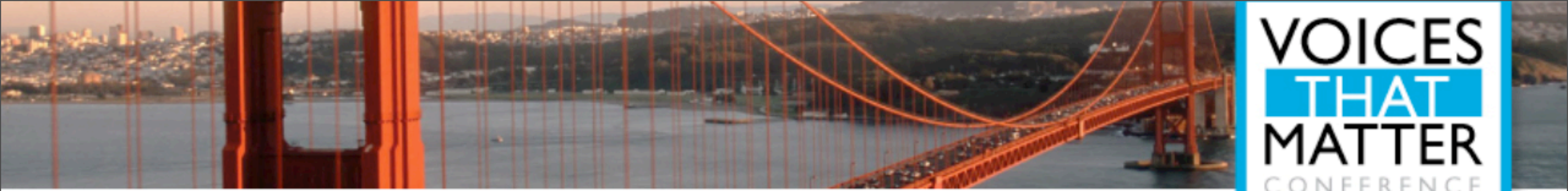
when you want to control the interface you expose

Great way to build application views.



styles and css

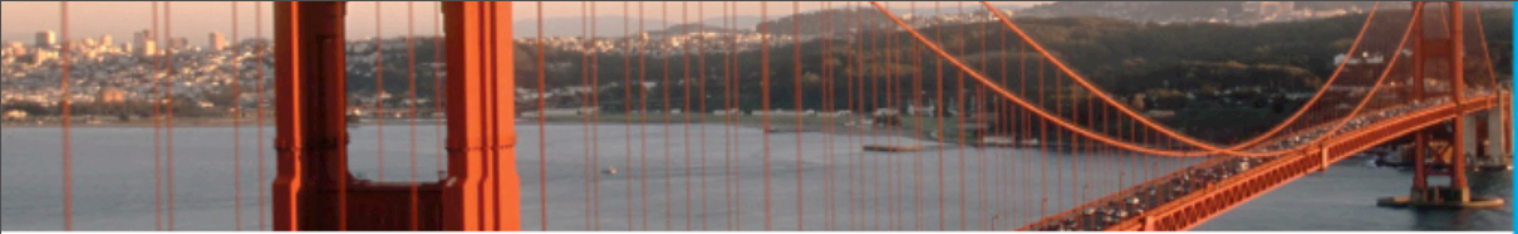
You should use style names and CSS for all widgets.
It's more efficient than setting style properties manually.



```
UIObject::
 setStyleName()
 addStyleName()
 removeStyleName()

 setStylePrimaryName()
 addStyleDependentName()
 removeStyleDependentName()
```

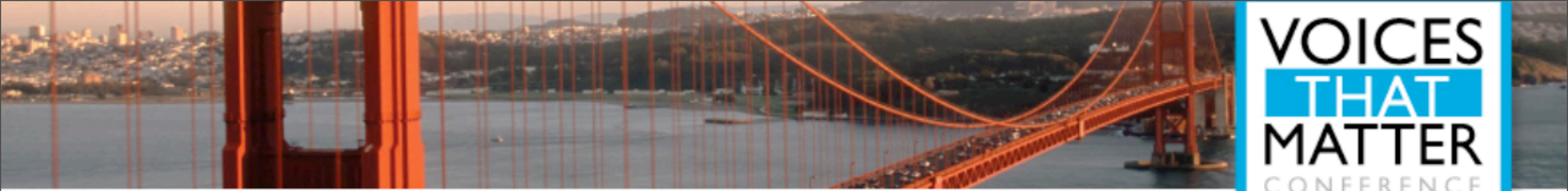
‘StyleName’ concept correlates to class names on the DOM.
set/add/remove pattern; regular and primary/dependent styles.



```
MyComposite w = new MyComposite();
w.setStylePrimaryName("myWidget");
w.addStyleDependentName("selected");
```

```
.myWidget { }
.myWidget .myWidget-selected { }
```

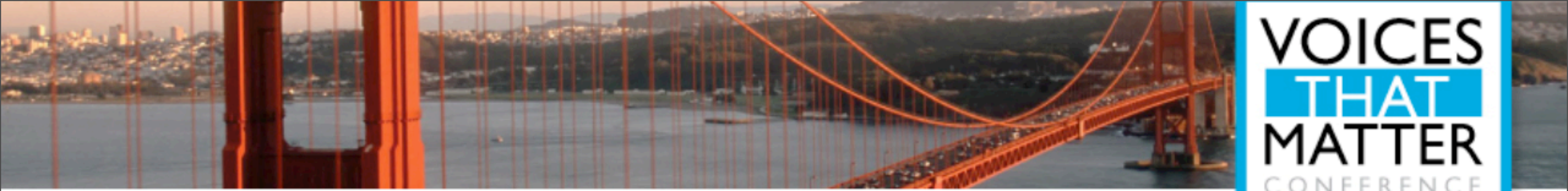
Primary and dependent style names are generally preferred.
Makes it possible to easily write CSS selectors for various states.
– IE doesn't support "union" selectors, thus the need for the repeated primary name.



when **not** to use css

```
.myWidget {
  position: absolute;
  width: 100%;
  /* etc */
}
```

Don't use CSS for properties necessary for your widget to function or layout correctly.

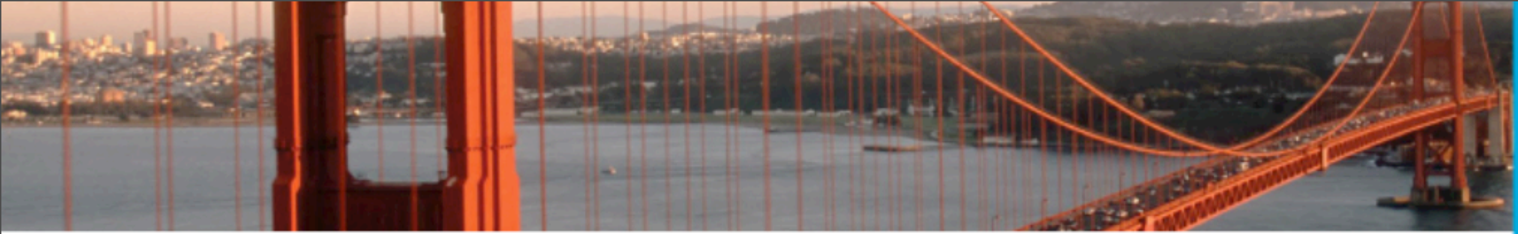


VOICES THAT MATTER CONFERENCE GOOGLE WEB TOOLKIT

fun with the DOM

```
DOM.createElement("button");
DOM.setElementAttribute(e, "tabIndex", "-1");
```

Most interaction with DOM elements and events goes through the DOM class.



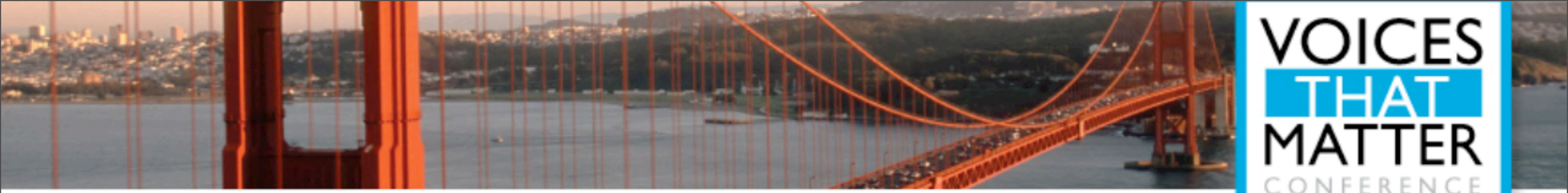
why is it so ugly?

why can’t I just type

```
element.setAttribute(“foo”, “bar”);
```

?

Serves as a browser abstraction layer.
Deferred binding is used to select browser-specific implementation.



VOICES THAT MATTER CONFERENCE GOOGLE WEB TOOLKIT

Interface:
```
public abstract boolean compare(Element e1, Element e2);
```

Standard:
```
public native boolean compare(Element e1, Element e2) /*-{
  return (e1 == e2);
}-*/;
```

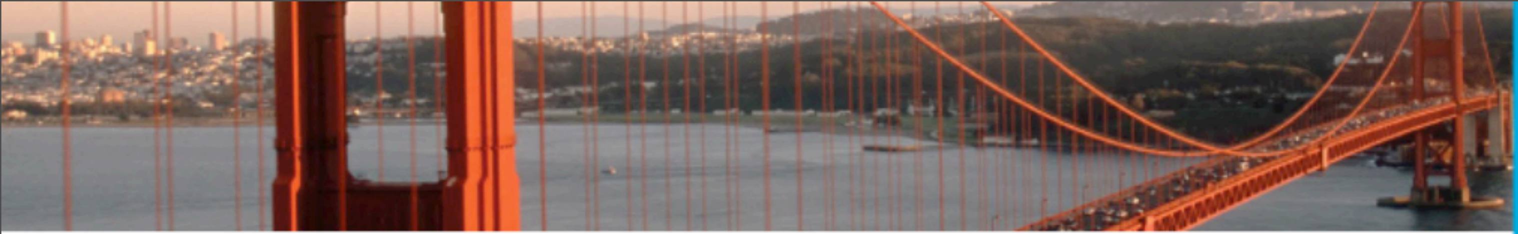
IE6:
```
public native boolean compare(Element e1, Element e2) /*-{
  if (!e1 && !e2)
    return true;
  else if (!e1 || !e2)
    return false;
  return (e1.uniqueID == e2.uniqueID);
}-*/;
```

Even something as simple as element comparison can get hairy across browsers.
Shocker: IE often differs drastically from others.

yeah, but can’t you make it **less** ugly?

```
class Element extends JavaScriptObject {
  public final void setAttributeString(
      String name, String value) {
    DOM.setElementAttribute(this, name, value);
  }
}
```

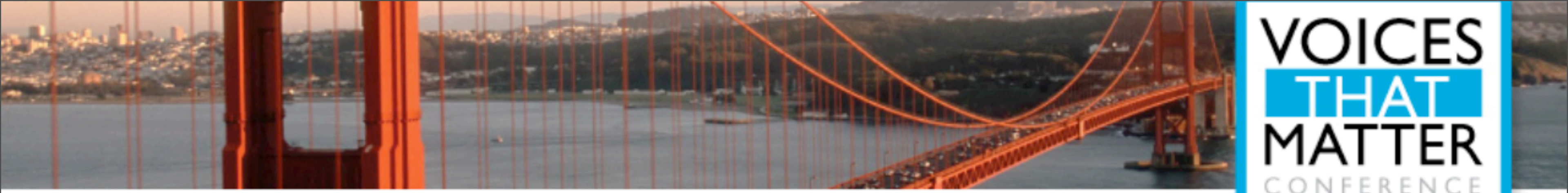
New compiler optimizations (inlining in particular) allow us to add convenience methods on JavaScriptObject, Element and Event.
Looks like an extra level of abstraction, but it gets inlined out in practice.



a bit of background

browser memory leaks

Change gears for a bit and talk about memory leaks.
Why? Everything after this point in the presentation is motivated by the desire to fix them.

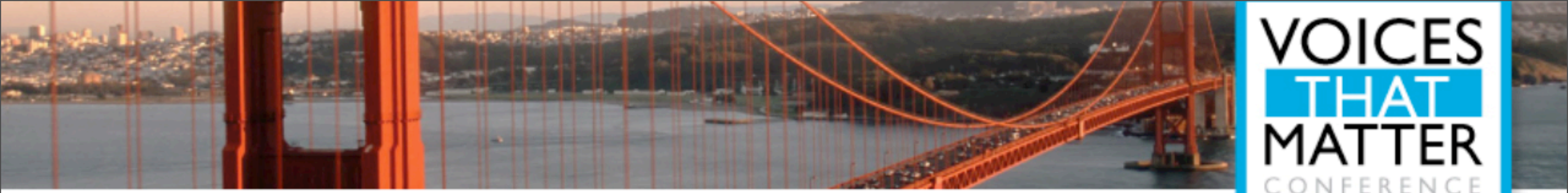


history

Garbage Collection
Reference Counting

LiveScript	COM
JavaScript	XPCOM
JScript	NSObject

Netscape LiveScript had a reference-counted memory allocator.
- This got fixed pretty quickly.
Internet Explorer implemented JScript with a garbage collector.
- Used COM (reference counted) for native interaction.
- This never got fixed, and probably can't without a major rearchitecture.
Mozilla (XPCOM) and WebKit (NSObject)
- Suffered from this to varying degrees, but are mostly fixed now.

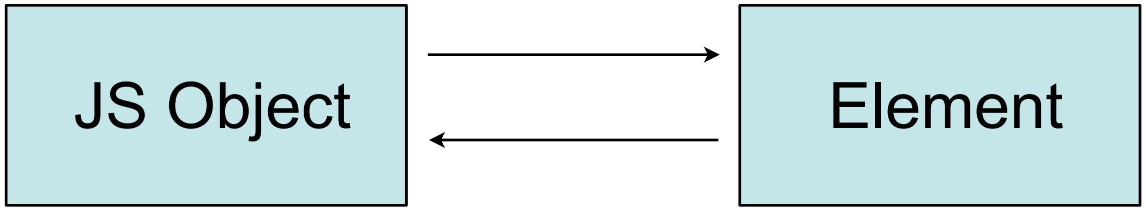


why does this matter?

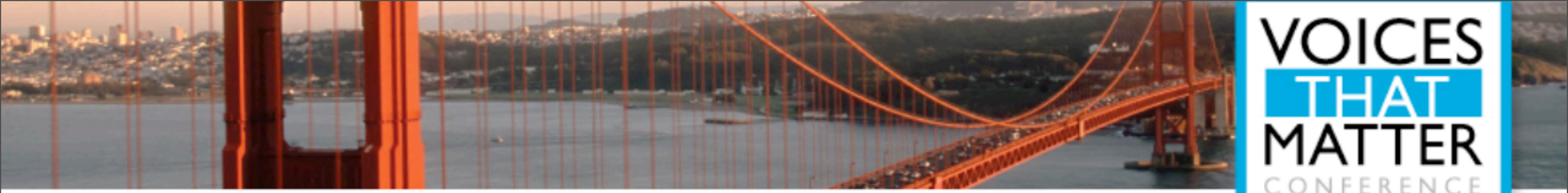
Memory leaks create some of the worst performance problems you can have.

"Transitive closure sickness":
- if you leak one object, you tend to leak the world
Browser performance tends to degrade with high memory usage
- even when not swapping
- IE in particular
This makes every app slower, since browsers tend to have a single process

circular references



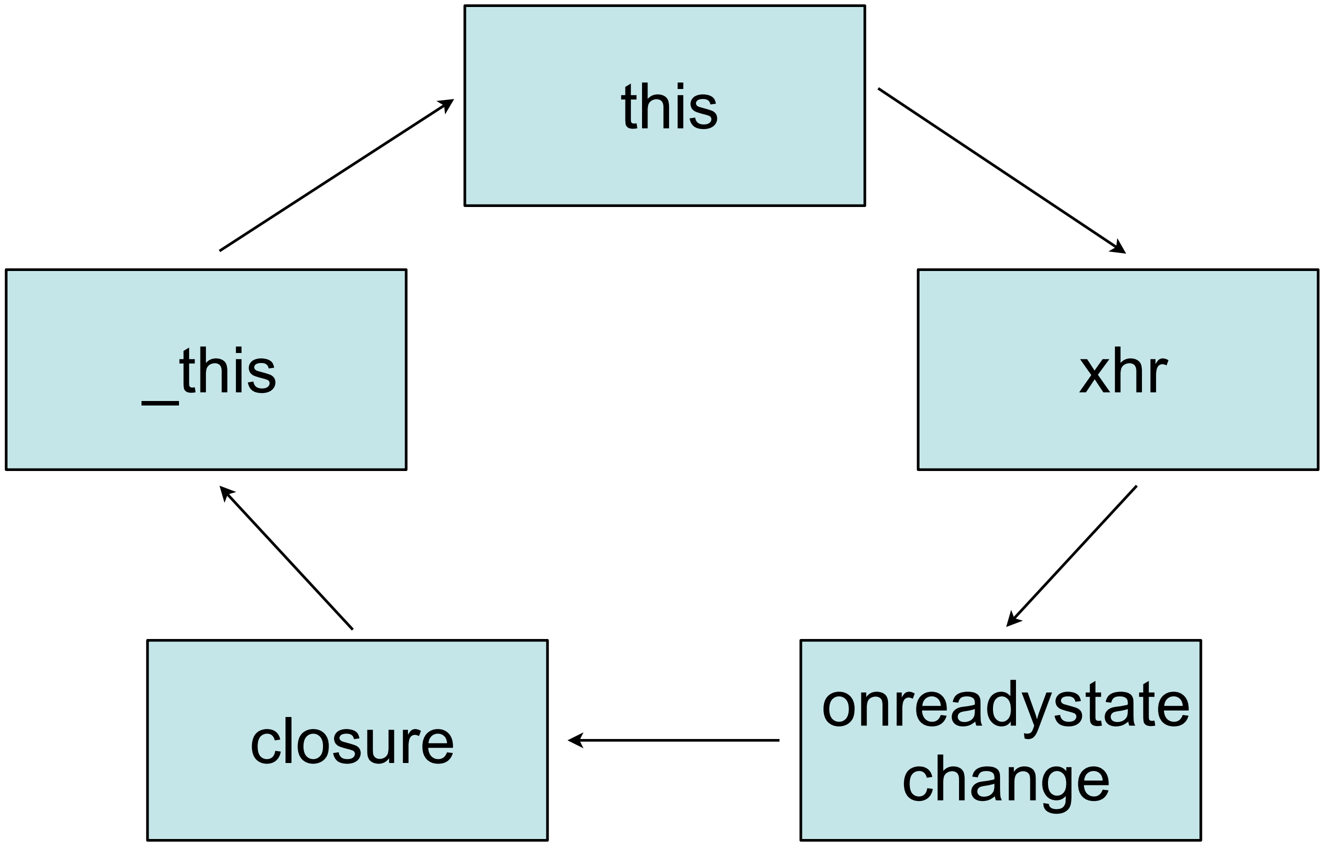
Leaks occur whenever you have a reference cycle involving native objects.
It's very easy to anger the circular reference gods by accident.



find the leak!

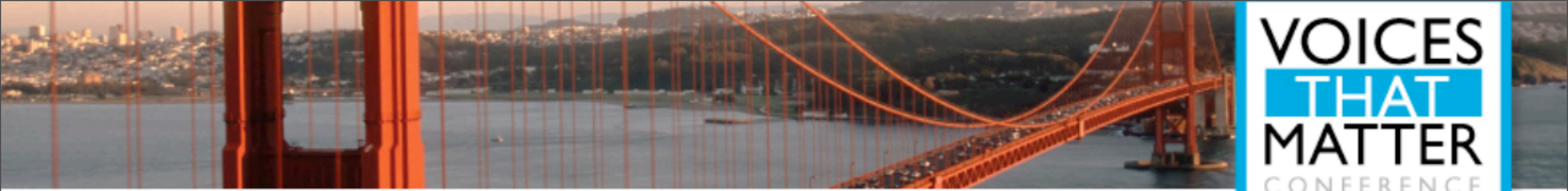
```
var _this = this;
this.xhr.onreadystatechange = function() {
  if (_this.xhr.readyState == 4) {
    _this.onCompletion(_this.xhr.responseText);
  }
};
```

find the leak!



$$\begin{tikzcd} & \text{this} \arrow[dr] & \\ \text{\_this} \arrow[ur] & & \text{xhr} \arrow[d] \\ \text{closure} \arrow[u] & & \text{onreadystate change} \arrow[ll] \end{tikzcd}$$

GWT libraries make it nearly impossible to leak.
But that forces some complexity on widget developers.



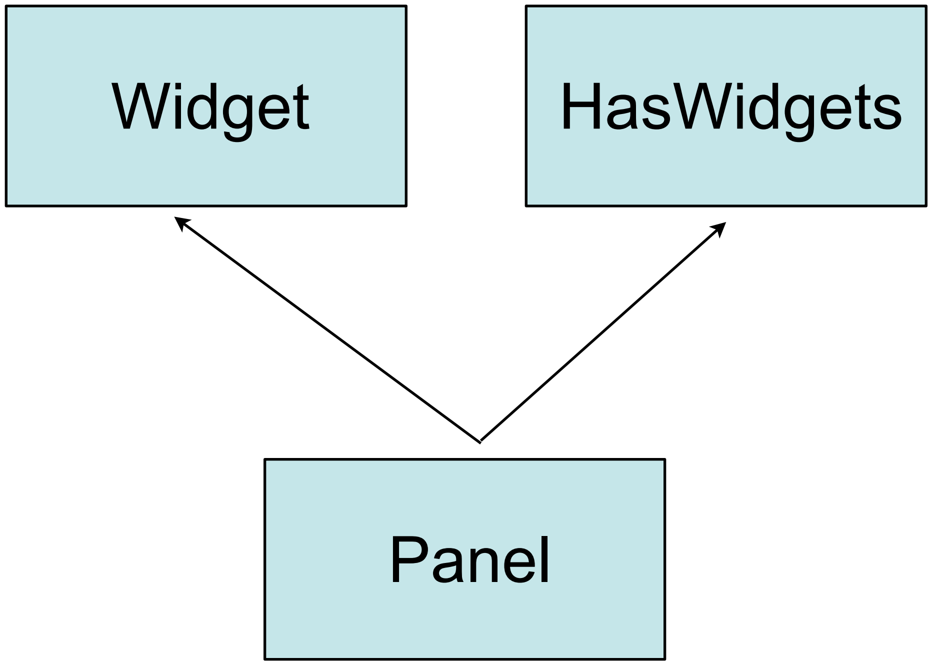
javascript events

```
sinkEvents(Event.ONCLICK);

public void onBrowserEvent(Event event) {
  switch (DOM.eventGetType(event)) {
    case Event.ONCLICK:
      // do something interesting.
      break;
  }
}
```

All events originate from native events at some point.
Bitfields? Only one listener? You've gotta be kidding me!
Event handlers are always reference cycles until they get cleaned up.
Detaching and re-attaching a widget requires that event handlers be removed.
If we want to re-add them, we need a concise way of representing them.
We also need to know who's listening. Having only one listener drastically simplifies this.

Panel and HasWidgets

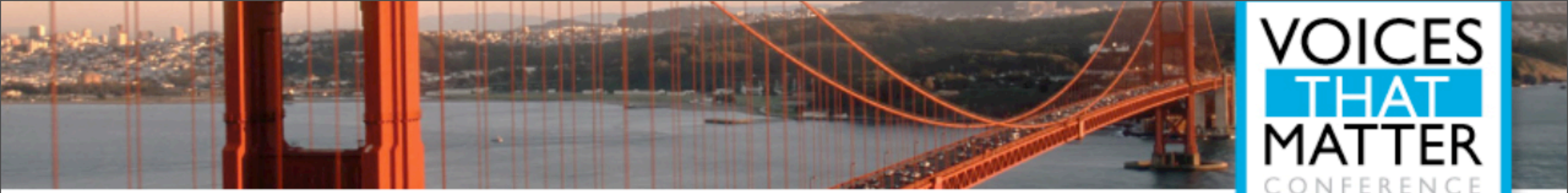


A bit of history:
- HasWidgets is really the interface we're concerned with.
- Panel is an inconveniently-named helper.

attachment lifecycle

```
Widget::
 onLoad()
 onUnload()
 onAttach()
 onDetach()
```

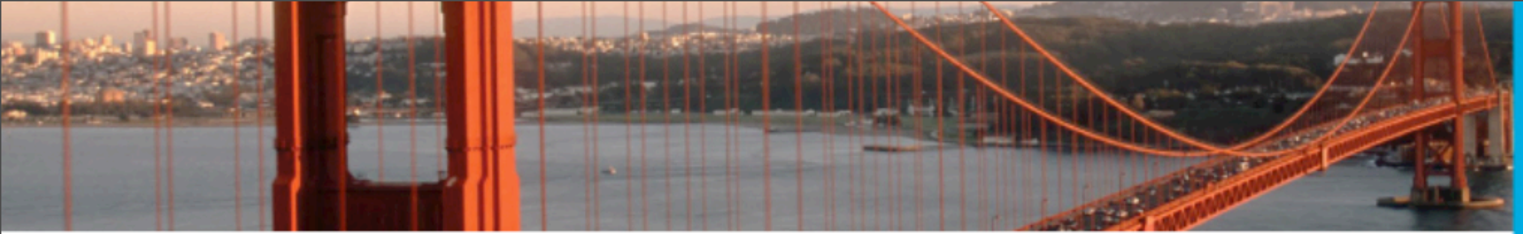
What do onLoad() and onUnload() mean?
- onLoad() is called after a widget becomes fully attached to the DOM.
- onUnload() is called just before it becomes detached.
This is really important for some widgets (e.g. RichTextArea).



a panel's children must be enumerable

```
protected void doAttachChildren() {
  for (Iterator it = iterator(); it.hasNext();) {
    Widget child = (Widget)it.next();
    child.onAttach();
  }
}
```

The default implementation of doAttachChildren() depends upon HasWidgets::iterator().
Back to memory leaks: if they're not enumerable, they can't be cleaned up.
Circular reference are created and broken in onAttach() and onDetach().

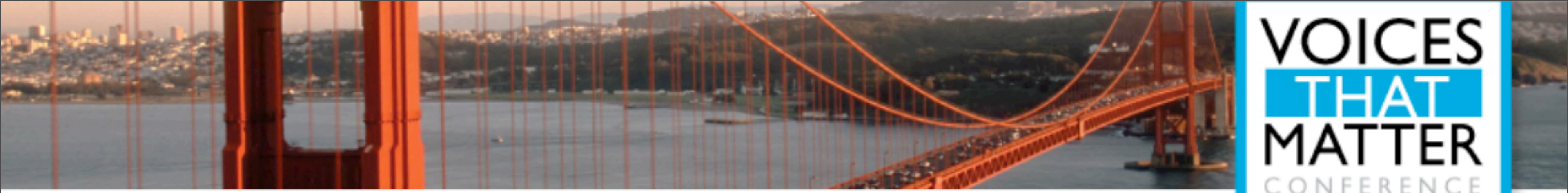


a final word on leaks

you don’t want your app leaking **while it’s running**

cleaning up in onunload() is not enough!

Why do we have to be so fastidious about removing event handlers on detachment?
If you want to have a long-running application, this is a strict requirement.
Most frameworks fail on this metric.
- They forcibly clean up event-handler references only on unload.
- This is simply insufficient.



Questions?