

# Develop HTML widgets with Dojo

## Get started with the Dojo toolkit 0.3

Skill Level: Intermediate

[Igor Kusakov \(igor@kusakov.com\)](mailto:igor@kusakov.com)

Web client lead developer

IBM

31 Oct 2006

Updated 14 Feb 2007

In this in-depth introductory tutorial by Web application developer Igor Kusakov, you learn the basics of developing HTML widgets using Dojo; including how to refer an image, how to add an event handler to an HTML page, and how to handle composite widgets. Also, discover some important differences between plain old JavaScript-style coding versus using Dojo, and get tips for handling complex issues inherent in Web application development.

Note that this tutorial does not address the overall architecture of a Dojo application or the various Dojo packages. Since the Dojo toolkit is rapidly evolving, many issues discussed in the tutorial are subject to change.

## Section 1. Before you start

Learn what to expect from this tutorial and how to get the most out of it.

### About this tutorial

The goal of this tutorial is to get you started with HTML widget development using

the Dojo toolkit. You'll learn how to develop Dojo HTML widgets, starting with one that is very simple and working up to one that is more complex. By the end of the tutorial you will have experience with composite widgets and various event-handling alternatives.

This tutorial is intended to be technical, sober, and unbiased. It will be most useful to working developers who need to get their job done.

## Prerequisites

This tutorial is written for JavaScript programmers whose skills and experience are at an intermediate to expert level. You should have a working knowledge of browser-side JavaScript development.

## System requirements

You can use any browser that is supported by the Dojo toolkit to run the examples (see the [Dojo FAQ](#)). All you need to do is [download and extract the Dojo toolkit](#) onto your hard drive and add a few files, and you'll be able to run the example HTML files in your browser.

---

## Section 2. What is the Dojo toolkit?

This section introduces the Dojo toolkit: what it is, what it is used for, and some of its strengths and current weaknesses.

### The Dojo toolkit defined

Dojo is a multi-platform JavaScript toolkit. Dojo for JavaScript is much like the Java™ runtime library for the Java language. At present the biggest domain of JavaScript usage, and, thus, Dojo usage, is browser-side Web development.

### Overall perspective

With the recent hype around Web 2.0 and Ajax technologies, it seems that big business is ready to give JavaScript-based applications another serious try. Doing so is a good idea given how much the technology has evolved since the late

nineties, when such projects were typically failing. Over the past decade, JavaScript virtual machines have matured, browsers are more compatible, important standards were defined, CPU power and memory on personal computers have exponentially increased, and a lot of knowledge has accumulated about developing JavaScript applications. The Dojo toolkit appears to be one place where this experience is reflected in code.

## Conceptual approach

Conceptually, Dojo looks amazing. One of the first things you notice about it is the high standard of quality it brings to JavaScript development, seemingly inspired by the Java language. The fact that Dojo has evolved from several other JavaScript frameworks ensures the maturity of its basic concepts. The *toolkit* approach (versus a framework or library approach) allows developers to use Dojo only when it's needed; in other cases it does not stand in your way. Dojo's business-friendly open-source license also makes it attractive for various projects.

## Current challenges

The hard thing about Dojo is that it is a work in progress. It is currently not positioned as release-quality software. Documentation is sparse, particularly with regard to the widgets infrastructure and particular widgets. The best way to understand Dojo is still to study its code, which isn't the easiest entry point for new developers. Hopefully these issues will be resolved over time.

Another noticeable problem is the perception of widgets bundled with Dojo. Many people (managers especially) tend to perceive Dojo widgets as a complete set of "Lego parts" that can easily be used in applications. This is where Dojo is going, but currently it is not even close. Several of the Dojo widgets have enough quality to be generic (like the Tree widget) but most Dojo widgets are still a work in progress. They can be qualified as "reference implementation" or "sample implementation" but they're not yet suitable to be used out-of-the-box in a commercial project, where you have too many requirements to fit in.

As with any evolving technology, the solution to most of the current limitations of Dojo is to contribute to its development and documentation. This tutorial is one such contribution.

---

## Section 3. Getting started

In this section, you will set up the development environment for the tutorial.

## Set up the tutorial environment

Setting up the development environment for the tutorial is relatively simple, and you'll need to do it to follow the examples. Assuming your operating system and browser are [supported by the Dojo toolkit](#), do the following:

- Download the Dojo toolkit zip file from <http://dojotoolkit.org/>.
- Create a tutorial folder, for example `C:\tutorial`.
- Unzip the Dojo zip file into your newly created tutorial folder (for example: `C:\tutorial\dojo-0.3.1-ajax`).
- Rename the version-specific Dojo folder to `dojoAjax`. (For example, rename `C:\tutorial\dojo-0.3.1-ajax` to `C:\tutorial\dojoAjax`.)

## Code examples

The tutorial code samples refer to three files that you will need to create and modify as necessary. You'll encounter the example code further in the tutorial. For now, simply create the empty files and place them as follows (Eclipse provides nice tabs per file, but a simple notepad application can do the job if necessary):

- `test.html`
  - `C:\tutorial\test.html`
- `dojoAjax/src/widget/HelloWorld.js`
  - `C:\tutorial\dojoAjax\src\widget\HelloWorld.js`
- `dojoAjax/src/widget/template/HtmlHelloWorld.html`
  - `C:\tutorial\dojoAjax\src\widget\template\HtmlHelloWorld.html`

### Placing files

Note that the tutorial files are placed in internal Dojo directories. This might not be a good practice for your projects in the long term. See the [Dojo FAQ](#) (in [Resources](#)) for information on placing code outside of the Dojo directory structure.

When you come to the code examples in the next sections, copy and paste the code into your files as appropriate. You will also need to open the `test.html` file in your

browser. After saving the files and refreshing your browser you should be able to see the resulting widget. (Remember to hold the Ctrl key while you click Refresh on your browser, to avoid caching. The caching problem is discussed [later in the tutorial.](#))

Some people reported a problem with running code samples from this tutorial - when the widget does not get rendered in a browser. The problem is solved if you add `dojo.require("dojo.widget.*");` to either `test.html` or `HelloWorld.js`. I personally cannot reproduce this problem, so this line is not added to the tutorial code samples.

---

## Section 4. Dojo HTML widgets

This section explains what a Dojo HTML widget is and the files that compose it, describes the widget directory structure, and presents the first example of how to use a widget.

### What is a Dojo widget?

A *Dojo widget* is like a custom tag in a JavaServer Pages (JSP) page, or a decorated `<span jwcid="...">` in Tapestry, but processed on the Web browser side. Dojo widgets aim to encapsulate visual Web components for easy reuse.

The Dojo widgets structure is independent from the DOM objects structure. Each Dojo widget has its own unique `widgetId` and can contain zero, one, or more DOM elements, as well as child Dojo widgets.

### Widget parts

A widget is constructed from several files. The JavaScript code file is required, while others are optional.

- A **JavaScript code file (\*.js)** contains widget logic in JavaScript. It manipulates visual elements using the JavaScript DOM objects API. (Dojo provides many useful libraries to simplify common functionality [here.](#))
- An **HTML template (\*.html)** provides basic widget representation in HTML. If a template is very simple, you can provide it in the code file as a

string.

- A **CSS style file (\*.css)** contains visual styles, which are used by the HTML template and/or JavaScript code.
- **Image files (\*.gif, \*.jpg, etc.)** hold images used by the widget.

**Figure 1. Widget directory structure**

/dojoAjax	dojo.js file is located here, you need to include it in your html/jsp file
/src	
/widget	Code file (*.js)
/templates	HTML template file (*.html) CSS style file (*.css)
/images	Images (*.gif, *.jpg)

## A Dojo widget in HTML code

Now look at how a Dojo widget is defined in HTML. You will use a standard Dojo widget called "Button" for the example. Start by copying the following code into the *test.html* file that you created earlier, then load it into your browser. The code will dynamically translate into HTML that displays a Dojo button widget with the caption "Submit."

**Listing 1. test.html: A Dojo button widget**

```
<html>
<head>
<script type="text/javascript" src="dojoAjax/dojo.js"></script>
<script type="text/javascript">
  dojo.require("dojo.widget.Button");
</script>
</head>
<body>
  <div dojoType="Button">Submit</div>
</body>
</html>
```

In [Listing 1](#), the widget type is defined as a *dojoType* attribute of an HTML *<div>* element. You can also define a Dojo widget with its own unique tag, as shown in [Listing 2](#).

**Listing 2. test.html: The button widget defined with a unique tag**

```
<html>
<head>
<script type="text/javascript" src="dojoAjax/dojo.js"></script>
<script type="text/javascript">
  dojo.require("dojo.widget.Button");
</script>
</head>
<body>
```

```
<dojo:button>Submit</dojo:button>
</body>
</html>
```

The first way (`<div dojoType="Button">...`) is typically used, as long as it works in most browsers. It also allows for the use of HTML editors that are unaware of Dojo-specific tags.

You will use the first approach throughout this tutorial.

---

## Section 5. The 'Hello World' widget

A classic "Hello World" widget will further familiarize you with the files and directory structure discussed in the [previous section](#).

### Hello World.js

[Listing 3](#) shows the required JavaScript file for the HelloWorld widget. Save this code in the appropriate file that you created during tutorial setup.

#### Listing 3 . dojoAjax/src/widget/HelloWorld.js

```
dojo.provide("dojo.widget.HelloWorld");
dojo.require("dojo.widget.HtmlWidget");

dojo.widget.defineWidget("dojo.widget.HelloWorld",
    dojo.widget.HtmlWidget, {
        templatePath:
        dojo.uri.dojoUri("src/widget/templates/HtmlHelloWorld.html")
    });
```

Note that `dojo.widget.HtmlWidget` is a superclass for the HelloWorld widget.

[Listing 4](#) shows the HTML template developed from the sample code shown in [Listing 3](#). Save this code in the appropriate file that you created during tutorial setup.

#### Listing 4. dojoAjax/src/widget/template/HtmlHelloWorld.html

```
<div>
    Hello World
</div>
```

After you save the following HTML into test.html file and then load it into a browser, you should see a message with the words "Hello World."

### Listing 5. test.html: Hello World test

```
<html>
<head>
<script type="text/javascript" src="dojoAjax/dojo.js"></script>
<script type="text/javascript">
  dojo.require("dojo.widget.HelloWorld");
</script>
</head>
<body>
  <!-- as an HTML element with dojoType attribute -->
  <div dojoType="HelloWorld"></div>
</body>
</html>
```

## Providing an HTML template as a string

If an HTML template is simple, it sometimes makes sense to keep it in a JavaScript file as a string, as shown in Listing 6.

### Listing 6. dojoAjax/src/widget/HelloWorld.js

```
dojo.provide("dojo.widget.HelloWorld");
dojo.require("dojo.widget.HtmlWidget");

dojo.widget.defineWidget("dojo.widget.HelloWorld", dojo.widget.HtmlWidget, {
  templateString: "<div>Hello World</div>"
});
```

---

## Section 6. Widget development basics

This section covers some of the basics of creating Dojo HTML widgets, such as using attributes to customize a widget; inserting code snippets in HTML templates; and referring to CSS and image files from various places.

### Using widget attributes

It is possible to customize a widget's view, as well as its behavior, using attributes. For example, you can modify the HelloWorld widget so it gets its display text from an attribute. Copy the code from [Listing 7](#), [Listing 8](#), and [Listing 9](#) into the appropriate

files and reload test.html in your browser. The widget will display the message: "This text is passed as a widget attribute."

### Listing 7. test.html

```
<html>
<head>
<script type="text/javascript"
src="dojoAjax/dojo.js"></script>
<script type="text/javascript">
    dojo.require("dojo.widget.HelloWorld");
</script>
</head>
<body>
    <div text="This text is passed as a widget
attribute"
        dojoType="HelloWorld"></div>
</body>
</html>
```

In [Listing 8](#), JavaScript code provides an attribute definition.

### Listing 8. dojoAjax/src/widget/HelloWorld.js

```
dojo.provide("dojo.widget.HelloWorld");
dojo.require("dojo.widget.HtmlWidget");
dojo.widget.defineWidget("dojo.widget.HelloWorld", dojo.widget.HtmlWidget, {
    templatePath: dojo.uri.dojoUri("src/widget/templates/HtmlHelloWorld.html"),
    // widget attributes
    text: "" // default value
});
```

Note that you **must** specify attributes with some default values (see the `text` attribute in [Listing 8](#)), otherwise they will be ignored. (Dojo will ignore attributes in the DOM node used to define the widget in *test.html*.)

In the HTML template in [Listing 9](#) the attribute value is transferred directly to where it needs to be using a code snippet (discussed in the next section).

### Listing 9. dojoAjax/src/widget/template/HtmlHelloWorld.html

```
<div>${this.text}</div>
```

## Processing attributes

Attributes are typically processed in widget code in the `postCreate()` method (discussed later in the tutorial). Another popular place is `fillInTemplate()` method. In many Dojo widgets you will notice that there is a widget attribute, which is

processed in `postCreate()`. But there is also something that seems like a setter for the same attribute (in the case of the above examples it would be `setText("...")`). These are two different things. The `postCreate()` method processes attributes on widget creation, while setters are used to set attributes dynamically at runtime. Attributes and setters are different things. It is good coding practice to keep them consistent (unlike in the above example).

## Code snippets in HTML templates

Having code in an HTML template sounds like a bad idea. But sometimes you really need it; such as to refer to a `dojoRoot` for images, to display an attribute directly, or to provide support for internationalization.

Dojo gives you a way to include JavaScript code in HTML templates using code snippets. Using the `${...}` notation ensures that code gets evaluated in the context of the widget object. So everything under *this* in the code, you can theoretically use in HTML templates. It's a good idea to restrict code snippets to the `dojoRoot` constant (discussed just below) and public widget attributes, however.

[Listing 10](#) shows how code snippets are used in HTML templates.

### Listing 10. `dojoAjax/src/widget/template/HtmlHelloWorld.html`

```
<div id="${this.widgetId}">
  
  ${this.text}
</div>
```

It is also possible to specify event handlers in HTML templates, as discussed later in the tutorial.

## Adding CSS

If you have global CSS files that are already loaded by HTML pages, you can refer to their CSS classes in your HTML templates and/or your JavaScript code. This might be okay for project-specific widgets.

But to make a widget truly self-contained and reusable across various projects, Dojo provides a way to define a per-widget CSS file. This file will load automatically with the related widget.

[Listing 11](#) shows how to define a per-widget CSS file in a JavaScript code file.

### Listing 11. `dojoAjax/src/widget/HelloWorld.js`

```
dojo.provide("dojo.widget.HelloWorld");
dojo.require("dojo.widget.HtmlWidget");
dojo.widget.defineWidget("dojo.widget.HelloWorld", dojo.widget.HtmlWidget, {
    templatePath: dojo.uri.dojoUri("src/widget/templates/HtmlHelloWorld.html"),
    templateCssPath: dojo.uri.dojoUri("src/widget/templates/HtmlHelloWorld.css")
});
```

[Listing 12](#) shows how to use a CSS class in an HTML template.

### Listing 12. dojoAjax/src/widget/template/HtmlHelloWorld.html

```
<div class="helloWorld_caption">
    Hello World
</div>
```

[Listing 13](#) shows a sample CSS file that defines a class that is referred to in [Listing 12](#).

### Listing 13. dojoAjax/src/widget/template/HtmlHelloWorld.css

```
.helloWorld_caption {
    font-family: Verdana;
    font-size: 10pt;
}
```

Remember that CSS class names have to be unique across the project. So it is a good idea to use, for example, a prefix in CSS class names. Note also that there are few problems with applying inherited CSS to programmatically created DOM elements.

## Referring images

In Dojo, the legitimate way to refer images from a JavaScript code file is to use the `dojo.uri.dojoUri` method, providing the URL relative to the Dojo root directory. If your files are outside of the Dojo file structure, use `..` to find them. For example:

```
img.src = dojo.uri.dojoUri("../images/yourimage.gif");
```

### Listing 14. Referring an image from a JavaScript code file (relative to the Dojo root directory)

```
var img = document.createElement("img");
// the dojo way
img.src = dojo.uri.dojoUri("src/widget/templates/images/yourimage.gif");
// relative to the page using this component (depends on page location,
// thus breaks encapsulation and is not recommended)
img.src = "dojoAjax/src/widget/templates/images/yourimage.gif";
```

The Dojo way to refer images from an HTML template is to use a `${dojoRoot}` constant, with the URL relative to the Dojo root directory, as shown in [Listing 15](#).

### Listing 15. Referring an image from an HTML template file (relative to the Dojo root directory)

```

```

To refer images from a CSS file, you need to rely on the native CSS feature. Then you can refer images relative to the directory where the CSS file itself is located, as shown in [Listing 16](#).

### Listing 16. Referring an image from a CSS file (relative to the CSS file location)

```
background-image: url("images/yourimage.jpg");

/* Internet Explorer-only hack, relative to dojo root directory, not recommended */
background-image: expression(dojouri.dojouri(
    "src/widget/templates/images/yourimage.gif"));
```

---

## Section 7. Accessing widgets programmatically

In this section you'll start to add some code to the HelloWorld widget. In the process, you'll look at some of the things that can go wrong when you apply traditional JavaScript programming techniques to the Dojo toolkit.

### Accessing widgets the old-style JavaScript way

Suppose you decide to add some code to manipulate your HelloWorld widget, for example to change the "Hello World" text dynamically. [Listing 17](#) shows the code that a seasoned JavaScript developer might first try to write to manipulate a Dojo widget.

### Listing 17. test.html: Failing to access a widget the traditional way

```
<html>
<head>
<script type="text/javascript"
src="dojoAjax/dojo.js"></script>
<script type="text/javascript">
    dojo.require("dojo.widget.HelloWorld");
</script>
```

```
</head>

<body
  onload="document.getElementById('someID').innerHTML='New
text';">
  <div id="someID" dojoType="HelloWorld"></div>
</body>
</html>
```

See [Resources](#) to learn more about the Dojo event system.

*Gotcha!* This code does not work at all. Let's consider what went wrong.

## 1. The `<body onload="...">` attribute failed

The JavaScript `<body onload="...">` attribute is not compatible with the Dojo toolkit. Instead, Dojo wraps the `onload` event into its own event system. Here's the same code written the "Dojo way":

### Listing 18. test.html: Adding onload code Dojo style

```
<html>
<head>
...
<script>
  dojo.addOnLoad(function(){
    alert("onload!");
  });
</script>
</head>
```

## 2. A Dojo widget is not a DOM element

That's it, period. A Dojo widget is a separate entity from the DOM that, itself, typically contains one or more DOM elements.

When Dojo creates a widget, it removes the DOM node used to define it (`<div id="someID" dojoType="HelloWorld"></div>`). Thus, all its attributes are lost, except for those Dojo knows how to handle. For example, the `id` attribute will become the Dojo `widgetId`. If you want these attributes to be assigned to a particular DOM node in your widget, you need to take some actions. For example, you can transfer them to DOM nodes using HTML template, as shown in [Listing 19](#).

### Listing 19. dojoAjax/src/widget/template/HtmlHelloWorld.html

```
<div id="${this.widgetId}">
```

```
Hello World
</div>
```

In the sample above, you used the `${this.widgetId}` code snippet to specify an `id` attribute of the DOM node. `widgetId` is a standard widget *attribute* that defines its unique ID in a Dojo widgets infrastructure. It is also possible to use it as a DOM ID.

Another way to transfer attributes is to override the `fillInTemplate()` method in your JavaScript code file. Transferring "style" attribute is the main reason this method typically gets overridden.

## Mixed solution, mixed results

Now you can programmatically access a Dojo component with some old-style JavaScript code, but with the major failure points removed. The working sample in [Listing 20](#) does the same thing you first tried in [Listing 17](#).

### Listing 20. test.html: Accessing a Dojo widget the traditional way

```
<html>
<head>
<script type="text/javascript" src="dojoAjax/dojo.js"></script>
<script type="text/javascript">
  dojo.require("dojo.widget.HelloWorld");
  dojo.addOnLoad(function(){
    document.getElementById("someID").innerHTML="New text";
  });
</script>
</head>
<body>
  <div id="someID" dojoType="HelloWorld"></div>
</body>
</html>
```

On the one hand, it works. On the other hand, it's not the way Dojo intends you to use widgets. The explained technique is actually bad practice, because it breaks widget encapsulation. But if you've had JavaScript experience, this is likely the way you would start out with Dojo widgets. So let's look at the alternatives the Dojo toolkit has to offer.

## Accessing widgets the Dojo way

Once again you want to be able to change the HelloWorld widget's text dynamically, but this time you'll do it *the Dojo way*. Start by modifying the widget's HTML template and JavaScript code, as shown in [Listing 21](#).

### Listing 21. dojoAjax/src/widget/template/HtmlHelloWorld.html

```
<div dojoAttachPoint="domNode">
  Hello World
</div>
```

## Attachment points

A Dojo widget uses *attachment points* to operate on DOM elements created from the HTML template. *Attachment points* are basically named DOM elements. When coding the Dojo way, you don't need `id="{this.widgetId}"` because you will not access the root DOM node by its id, but by using an attachment point.

### domNode

You don't actually need to assign a *domNode* attachment point to the root DOM element manually, since it is done by default. The attachment point is assigned manually for the purpose of the tutorial only, to ensure the underlying concepts are learned.

We can assign an attachment point to any HTML element in the template. The root-element attachment point is traditionally named "domNode"; other elements you can name according to your naming conventions. It is a good practice to append *Node* as a suffix to any attachment point name, so the code is easier to read. Attachment points are resolved automatically on widget creation as widget fields. To refer to an attachment point from widget code use `this.someAttachPointNode` notation.

[Listing 22](#) shows the widget code with a dynamic setter, which is used to modify its displayed text. Note how the `domNode` attachment point is accessed.

### Listing 22. dojoAjax/src/widget/HelloWorld.js

```
dojo.provide("dojo.widget.HelloWorld");
dojo.require("dojo.widget.HtmlWidget");

dojo.widget.defineWidget("dojo.widget.HelloWorld", dojo.widget.HtmlWidget, {
  templatePath: dojo.uri.dojoUri("src/widget/templates/HtmlHelloWorld.html"),

  // dynamic setters
  setHtml: function(htmlString) {
    this.domNode.innerHTML = htmlString;
  }
});
```

## Getting a widget instance by its widgetId

The Dojo toolkit also allows you to dynamically get a widget reference by its `widgetId`, as shown in [Listing 23](#).

## Listing 23. Getting a widget by its widgetId

```
var widget = dojo.widget.manager.getWidgetById('someID');

// or, a short version:
var widget = dojo.widget.byId('someID');

// Gotcha: the following means document.getElementById('someID')
var widget = dojo.byId('someID');
```

## The final code

Here is the final working example, done the Dojo way. Note that here you use an abstract setter without knowing where or how the text will be displayed by the widget code. Thus the widget display logic is isolated from its user -- which is great.

## Listing 24. test.html

```
<html>
<head>
<script type="text/javascript" src="dojoAjax/dojo.js"></script>
<script type="text/javascript"> dojo.require("dojo.widget.HelloWorld");
    dojo.addOnLoad(function(){
        dojo.widget.byId("someID").setHtml("New text");
    });
</script>
</head>
<body>
    <div id="someID" dojoType="HelloWorld"></div>
</body>
</html>
```

## Creating a Dojo widget dynamically

In the previous examples you have created a widget *statically*, by defining it in the test.html file (<div dojoType="HelloWorld"></div>). You can also create a widget dynamically in JavaScript code, using the `dojo.widget.createWidget()` method.

## Listing 25. test.html: Creating a Dojo widget dynamically

```
<html>
<head>
<script type="text/javascript" src="dojoAjax/dojo.js"></script>
<script type="text/javascript">
    dojo.require("dojo.widget.HelloWorld");

    dojo.addOnLoad(function(){
        var newWidget = dojo.widget.createWidget(
            "HelloWorld", // widgetType
            {}, // widget attributes, for example {title: "Some Title"}
        );
    });
</script>
</head>
<body>
    <div id="someID" dojoType="HelloWorld"></div>
</body>
</html>
```

```

        dojo.byId("widgetPlaceholder") // reference to a DOM node that
        // will be REPLACED by new widget
    );
});
</script>
</head>
<body>
    <div id="widgetPlaceholder"></div>
</body>
</html>

```

The `dojo.widget.createWidget()` method takes two to four arguments, and the last two are a bit tricky.

- The first argument is the widget type.
- The second argument is a hash with widget attributes.
- The third argument is the DOM node and is treated differently depending on the fourth argument.
  - If there is no fourth argument, then the DOM node is **replaced** with the new widget (starting from its `domNode`).
  - If you *do* provide the fourth argument, then it is treated as a position for the `dojo.dom.insertAtPosition(node, ref, position)` method, and the widget is inserted relative to the third argument (the reference node is not replaced or deleted).
- The *position* argument might be as follows:
  - **before**: the widget is inserted before the reference node (thus becomes its previous sibling).
  - **after**: the widget is inserted after the reference node (thus becomes its next sibling).
  - **first**: the widget is inserted as a first child of a reference node.

It is also possible to handle widget insertion manually: just don't specify the third and fourth argument and insert the widget `domNode` wherever necessary, as shown in [Listing 26](#).

### Listing 26. test.html

```

<html>
<head>
<script type="text/javascript" src="dojoAjax/dojo.js"></script>
<script type="text/javascript">
    dojo.require("dojo.widget.HelloWorld");

    dojo.addOnLoad(function(){
        var newWidget = dojo.widget.createWidget(
            "HelloWorld", // widgetType
            {} // widget attributes, for example {title: "Some Title"}
        );
    });

```

```
    );
    dojo.byId("widgetPlaceholder").appendChild(newWidget.domNode);
  });
</script>
</head>
<body>
  <div id="widgetPlaceholder"></div>
</body>
</html>
```

Note that `dojo.widget.createWidget()` does not establish a parent/child relationship. So if you create a hierarchy of widgets you need to establish this relationship manually. I'll discuss this in detail in the upcoming sections on nested widgets and composite widgets.

---

## Section 8. Nested HTML

In this section, I'll begin to consider some of the more complex issues involved in creating Dojo HTML widgets, starting with different techniques for dealing with HTML nested in a widget element.

### Static nested HTML

Suppose you need to deal with HTML that a user defines statically within a widget element, as shown in [Listing 27](#).

#### Listing 27. test.html: Nested HTML sample

```
<html>
<head>
<script type="text/javascript"
src="dojoAjax/dojo.js"></script>
<script type="text/javascript">
  dojo.require("dojo.widget.HelloWorld");
</script>
</head>
<body>
  <div dojoType="HelloWorld">Some user-specific
    <b>HTML</b></div>
</body>
</html>
```

When Dojo creates a widget, it removes the DOM element used to define it. So you need to transfer all the nested HTML DOM elements you care about before Dojo removes them. To do this, you need to override the Dojo widget method `fillInTemplate()`. This method gives you access to both the original HTML

DOM elements and the already generated Dojo widget. [Listing 28](#) shows how to do this. (Note that you have to call the original superclass `fillInTemplate()` method.)

### Listing 28. dojoAjax/src/widget/HelloWorld.js: Transferring static nested HTML.

```
dojo.provide("dojo.widget.HelloWorld");
dojo.require("dojo.widget.HtmlWidget");

dojo.widget.defineWidget("dojo.widget.HelloWorld", dojo.widget.HtmlWidget, {
    templateString: '<div dojoAttachPoint="domNode"></div>',

    // override
    fillInTemplate: function(args, frag) {
        // Getting original HTML element
        var source = this.getFragNodeRef(frag);

        // Moving all children of original element to
        // the desired node of the new component
        while(source.hasChildNodes()) {
            var node = dojo.dom.removeNode(source.firstChild);
            this.domNode.appendChild(node);
        }

        // Invoking original dojo fillInTemplate() method
        dojo.widget.HelloWorld.superclass.fillInTemplate.call(this, args, frag);
    }
});
```

So, in the above code sample you moved all the HTML DOM nodes provided by a user as nested HTML to the place where you need it. In this case as children of a `domNode` attachment point. The HTML template is provided as a string.

Another way to handle static nested HTML is to set widget attribute `isContainer` to `true`. This should move the HTML content the same way as the [Listing 28](#) does.

## Adding nested HTML dynamically

You can use a couple of ways to add nested HTML dynamically, depending on how the dynamic HTML is provided:

- If it is provided as a string, you can set the new HTML using the `innerHTML` property of a desired DOM node.
- If it is provided as a DOM node, you can add it using the desired DOM node's `appendChild()` DOM method.

Here is a sample that operates properly in both cases.

### Listing 29. dojoAjax/src/widget/HelloWorld.js: Adding nested HTML dynamically

```
dojo.provide("dojo.widget.HelloWorld");
dojo.require("dojo.widget.HtmlWidget");

dojo.widget.defineWidget("dojo.widget.HelloWorld", dojo.widget.HtmlWidget, {
    templatePath: '<div dojoAttachPoint="domNode"></div>',

    // dynamic setters
    setHtml: function(html) {
        if (dojo.lang.isString(html)) {
            this.domNode.innerHTML = html;
        } else if (dojo.dom.isNode(html)) {
            // Cleaning whatever was there before
            dojo.dom.removeChildren(this.domNode);
            // Adding new element
            this.domNode.appendChild(html);
        } else {
            dojo.lang.assert(false, "setHtml called with incorrect type: "
                + (typeof html));
        }
    }
});
```

---

## Section 9. Nested widgets

In this section, you'll look at techniques for dealing with user-specified nested widgets.

### Static nested widgets

In some cases a user can specify nested widgets for the widget. The Dojo *TabContainer* and *Tree* widgets are good examples of this.

[Listing 30](#) shows how a nested Button widget is specified for the HelloWorld widget by a programmer -- *widget user* in test.html

#### Listing 30. test.html: Sample static nested widget

```
<html>
<head>
<script type="text/javascript"
    src="dojoAjax/dojo.js"></script>
<script type="text/javascript">
    dojo.require("dojo.widget.HelloWorld");
    dojo.require("dojo.widget.Button");
</script>
</head>
<body>
    <div dojoType="HelloWorld">
        <div dojoType="Button"></div>
    </div>
</body>
</html>
```

```

    </div>
  </body>
</html>

```

In the simplest case, you need to define the `isContainer` widget parameter to be `true`, and define a `containerNode` attachment point in the HTML template. And that's it -- nested widgets will automatically render and add to your `containerNode`.

### Listing 31. dojoAjax/src/widget/HelloWorld.js

```

dojo.provide("dojo.widget.HelloWorld");
dojo.require("dojo.widget.HtmlWidget");

dojo.widget.defineWidget("dojo.widget.HelloWorld", dojo.widget.HtmlWidget, {
  isContainer: true,
  templateString: '<div><div dojoAttachPoint="containerNode">'
    + '</div></div>'
});

```

The solution above is simple because it relies on default Dojo behavior. As soon as you need to do something more complex, however, you have to handle the *children* array (a standard widget field which holds all static nested widgets) in the `postCreate()` method.

In [Listing 32](#), you not only add nested user-defined widgets to the `containerNode`, but you also add a list of nested widget types to the `titleNode` attachment point.

### Listing 32. dojoAjax/src/widget/HelloWorld.js

```

dojo.provide("dojo.widget.HelloWorld");
dojo.require("dojo.widget.HtmlWidget");

dojo.widget.defineWidget("dojo.widget.HelloWorld", dojo.widget.HtmlWidget, {
  isContainer: true,
  templateString: '<div><div dojoAttachPoint="titleNode"></div>'
    + '<br/><div dojoAttachPoint="containerNode">'
    + '</div></div>',

  postCreate: function() {
    for(var i in this.children) {
      this._addRow(this.children[i]);
    }
  },

  _addRow: function(component) {
    this.titleNode.appendChild(
      document.createTextNode(component.widgetType));
    this.titleNode.appendChild(
      document.createElement("br"));
  }
});

```

[Listing 33](#) defines a `test.html` that runs with the widget defined in [Listing 32](#).

### Listing 33. test.html: Multiple static nested widgets

```
<html>
<head>
<script type="text/javascript" src="dojoAjax/dojo.js"></script>
<script type="text/javascript">
  dojo.require("dojo.widget.HelloWorld");
  dojo.require("dojo.widget.Button");
  dojo.require("dojo.widget.ContentPane");
</script>
</head>
<body>
  <div dojoType="HelloWorld">
    <div dojoType="Button"></div>
    <div dojoType="ContentPane"></div>
  </div>
</body>
</html>
```

See the Dojo toolkit's *TabContainer* and *Tree* widgets for a more useful example of handling nested widgets.

## Adding nested widgets dynamically

Imagine a user wants to add Dojo widgets to the widget programmatically at runtime.

Remember the discussion of widget attributes and their dynamic setters? The fact that consistency between these two things is not architecturally enforced might be the biggest flaw of current Dojo widgets architecture. It is, of course, more flexible this way and not that critical; but it is annoying because you never know if there is a dynamic setter for a given attribute. Developers tend just to forget to add them, which greatly decreases the widget's usability.

With nested widgets you have the same issue -- they are added dynamically in a different way than statically. For dynamic addition, you need to use the `addChild()` method. [Listing 34](#) shows a widget that handles both static and dynamic widget adding. Note that the reusable part is extracted in the `_addRow()` method. When overriding the `addChild()` method, you need to call its superclass as well.

### Listing 34. dojoAjax/src/widget/HelloWorld.js

```
dojo.provide("dojo.widget.HelloWorld");
dojo.require("dojo.widget.HtmlWidget");
dojo.widget.defineWidget("dojo.widget.HelloWorld", dojo.widget.HtmlWidget, {
  isContainer: true,
  templateString: '<div><div dojoAttachPoint="titleNode"></div>'
    + '<br/><div dojoAttachPoint="containerNode">'
    + '</div></div>',
  postCreate: function() {
    // adding statically
    for(var i in this.children) {
```

```

        this._addRow(this.children[i]);
    }
},
addChild: function(child, overrideContainerNode,
    pos, ref, insertIndex) {
    this._addRow(child);
    dojo.widget.HelloWorld.superclass.addChild.call(
        this, child, overrideContainerNode);
},
_addRow: function(component) {
    this.titleNode.appendChild(
        document.createTextNode(component.widgetType));
    this.titleNode.appendChild(
        document.createElement("br"));
});
};

```

[Listing 35](#) shows how to create a Button widget and dynamically add it to the widget in [Listing 34](#).

### Listing 35. test.html: Adding nested widgets dynamically

```

<html>
<head>
<script type="text/javascript" src="dojoAjax/dojo.js"></script>
<script type="text/javascript">
    dojo.require("dojo.widget.HelloWorld");
    dojo.require("dojo.widget.Button");

    dojo.addOnLoad(function(){
        var button = dojo.widget.createWidget("Button",
            {caption: "Submit"});
        dojo.widget.byId('someID').addChild(button);
    });
</script>
</head>
<body>
    <div id="someID" dojoType="HelloWorld">
    </div>
</body>
</html>

```

---

## Section 10. Composite widgets

Imagine you need your widget always to contain another widget. This section defines two ways to handle this: programmatically and through an the HTML template.

### A note about terminology

There is as yet no perfectly defined terminology to describe nested and composite widgets. For the purpose of this tutorial, here is what

I mean when I use these terms:

**nested**

A widget that is added *externally* as a child to another widget, either as a nested HTML element or using `addChild()`. For example a `TabContainer` widget can have a `ContentPane` as a nested widget.

**composite**

A widget that is always created *internally* as a child widget, typically in the `postCreate()` method of a parent widget.

## Defining composite widgets programmatically

First, you'll take a programmatic approach to creating a composite widget. (See [Resources](#) to learn more about this approach.)

In the `HelloWorld` widget's HTML template, define an attachment point where you want a new widget to appear. Then, in the widget's `postCreate()` method, programmatically create a new widget and add it to the node.

Composite widgets need to be added to a main widget's child list using the `addChild()` method (the same one you used for nested widgets). This ensures they will be destroyed properly with the main widget and prevents a memory leak.

[Listing 36](#) shows how to programmatically create a `Button` widget and add it to the main widget.

### Listing 36. `dojoAjax/src/widget/HelloWorld.js`

```
dojo.provide("dojo.widget.HelloWorld");
dojo.require("dojo.widget.HtmlWidget");
dojo.require("dojo.widget.Button"); // we require nested widget definition

dojo.widget.defineWidget("dojo.widget.HelloWorld", dojo.widget.HtmlWidget, {
  isContainer: true,
  templateString: '<div>Hello World <br/> '
    + '<div dojoAttachPoint="compositeWidgetNode"></div></div>',

  // override
  postCreate: function(args, frag) {
    this.btnSubmit = dojo.widget.createWidget("Button", { caption: "Submit" });
    this.addChild(this.btnSubmit, this.compositeWidgetNode);
  }
});
```

Consider this code in detail:

- First, include a reference to the required widget definition using the `dojo.require("dojo.widget.Button");` call.

- Then, set the `isContainer` attribute to `true`. This attribute comes from the `HtmlWidget` superclass. It provides an infrastructure for managing widget children: `addChild()` and `removeChild()` methods, calling `onResized()` when a parent widget is resized, and so on.
- The HTML template (file or string) can define a node where new child widgets will be added. The default name is `containerNode`. If you name your attachment point differently (as done in this example), you need to specify it as a second argument in the `addChild()` call. For consistency, try to use `containerNode`, especially in cases where you have only one node for composite widgets.
- Next, use `dojo.widget.createWidget()` to create a composite widget and a `this.addChild()` to add it.

Instead of `addChild()` you could, of course, do `this.nestedWidgetNode.appendChild(this.btnSubmit.domNode);` In such case, however, the new widget is not added to the parent widget's child list, so it is not destroyed automatically when its parent is destroyed. The new widget would also miss out on other benefits of the parent-child relationship, like being automatically notified of resizing. In most cases it is better to stick with the provided infrastructure and leave hacks as a last resort.

[Listing 37](#) shows the `test.html` file that you can use to test the widget in [Listing 36](#).

### Listing 37. test.html: Composite widget test

```
<html>
<head>
<script type="text/javascript" src="dojoAjax/dojo.js"></script>
<script type="text/javascript">
  dojo.require("dojo.widget.HelloWorld");
</script>
</head>
<body>
  <div dojoType="HelloWorld">
    </div>
</body>
</html>
```

## Defining composite widgets in the HTML template

Another way to define composite widgets is to define them directly in the HTML template. This approach is legitimate, and Dojo 0.4 will supposedly have a more direct way of doing this. The following code sample looks bigger than the previous one, but in complex cases it looks cleaner and requires much less routine JavaScript code.

[Listing 38](#) shows a widget code that handles a composite widget defined in an HTML template. This approach currently requires a bit of Dojo magic to do. Please use the test.html from [Listing 37](#) to test this widget.

### Listing 38. dojoAjax/src/widget/HelloWorld.js

```

dojo.provide("dojo.widget.HelloWorld");
dojo.require("dojo.widget.HtmlWidget");
dojo.require("dojo.widget.Button"); // we require nested widget definition
dojo.require("dojo.widget.*"); // useful libraries
dojo.require("dojo.xml.Parse");
dojo.require("dojo.dom");

dojo.widget.defineWidget("dojo.widget.HelloWorld", dojo.widget.HtmlWidget, {
  widgetType: "HelloWorld",
  isContainer: true,
  templateString: '<div>Hello World <br/> '
    + '<div dojoType="Button" caption="Submit">'
    + '</div></div>',

  // override
  postCreate: function(args, frag) {
    // dojo mambo-jumbo that converts all dojo tags into widgets
    var xmlParser = new dojo.xml.Parse();
    var dojoParser = dojo.widget.getParser();
    var jsObj = xmlParser.parseElement(this.domNode, null, true);
    var components = dojoParser.createComponents(jsObj, this);

    // now, 'components' is an array with all
    // first-level composite dojo widgets
    this.btnSubmit = components[0];
  }
});

```

In the `postCreate()` method, when the widget is already created from the HTML template, you invoke the Dojo parser on it, so it converts all Dojo definitions into widgets. As a result you receive an array of first-level widgets it has created.

Widgets in the array appear in the same order as in the HTML template. Use `someWidget.children[0]` to access a child widget of a composite widget.

#### Dojo 0.4 and composite widgets

Dojo 0.4 introduces a new way to define composite widgets inside HTML templates with *widgetsInTemplate* attribute. Discussing this feature is outside of the boundaries of this tutorial.

Another alternative to retrieving created widgets is to give them unique IDs, and then get them by these ids (`dojo.widget.byId("...")`). Doing this means we should take care of keeping the ids unique (when there are multiple instances of a widget on a page). It also looks more bulky this way -- there are already many other ids in the HTML template. The benefit is that no matter how the HTML template is reordered it will still work.

In both cases composite widgets are added properly as children of the root widget

using the `createComponents()` method.

---

## Section 11. Handling events in Dojo

In this section you will look at some of the particulars of event handling done "the Dojo way." Before you read this section, you might want to read an introduction to the [Dojo event system](#) in Resources.

### Attaching event handlers in HTML templates

[Listing 39](#) provides an example of how you are supposed to handle events in a Dojo widget using the Dojo event system. In an HTML template we need to define a `dojoAttachEvent="eventName: eventHandlerMethod;"` attribute. Where *eventName* is a standard DOM event name (like "onClick"). *eventHandlerMethod* is a method that is defined in widget JavaScript code and which will be called when an event occurs. This method will receive the Dojo event object.

[Listing 39](#) shows how the `dojoAttachEvent` attribute is used in an HTML template.

#### Listing 39. `dojoAjax/src/widget/template/HtmlHelloWorld.html`

```
<div>
  <span dojoAttachEvent="onClick:
onClickHandler;">
    Click me
  </span>
</div>
```

[Listing 40](#) shows how the `onClickHandler()` event handler is defined in JavaScript code.

#### Listing 40. `dojoAjax/src/widget/HelloWorld.js`

```
dojo.provide("dojo.widget.HelloWorld");
dojo.require("dojo.widget.HtmlWidget");

dojo.widget.defineWidget("dojo.widget.HelloWorld", dojo.widget.HtmlWidget, {
  templatePath: dojo.uri.dojoUri("src/widget/templates/HtmlHelloWorld.html"),

  onClickHandler: function(evt) {
    alert("Node clicked");
  }
});
```

[Listing 41](#) shows the test.html file that you can use to test the widget defined in [Listing Listing 39](#) and [Listing 40](#).

### Listing 41. test.html: Event handling test

```
<html>
<head>
<script type="text/javascript" src="dojoAjax/dojo.js"></script>
<script type="text/javascript">
  dojo.require("dojo.widget.HelloWorld");
</script>
</head>
<body>
  <div dojoType="HelloWorld">
  </div>
</body>
</html>
```

Handling events the Dojo way is a bit different from using standard DOM event handlers (like `node.onclick = function(evt) { this.className='clicked' ; } ;`). For one thing, you cannot access a target node using the `this` keyword. Instead you use Dojo's `evt.currentTarget` attribute.

For another thing, what if you need to return *true/false* from the event handler? Again the solution is to operate on an event object you get from Dojo. Use `evt.returnValue = true/false;`

Another interesting feature is how Dojo lets you to stop an event from bubbling over to parent DOM nodes. Use the `dojo.event.browser.stopEvent(evt);` method.

Dojo also allows you to specify multiple event handlers separated with a semicolon (;), like this: `dojoAttachEvent="onClick: onClickHandler; onKeyPress: onKeyPressHandler; "`.

## Attaching event handlers dynamically

You can attach event handlers dynamically in JavaScript code wherever you need it. In [Listing 42](#) you attach an event handler to the HelloWorld widget's `domNode` DOM element in the `postCreate` method. Please use the test.html from [Listing 41](#) to test this widget.

### Listing 42. dojoAjax/src/widget/HelloWorld.js

```
dojo.provide("dojo.widget.HelloWorld");
dojo.require("dojo.widget.HtmlWidget");
dojo.require("dojo.event.*");
```

```
dojo.widget.defineWidget("dojo.widget.HelloWorld", dojo.widget.HtmlWidget, {
  templateString: '<div dojoAttachPoint="domNode">Click here</div>',

  postCreate: function() {
    var controller = this;
    dojo.event.kwConnect({

      srcObj: this.domNode,
      srcFunc: "onclick",
      targetObj: controller,
      targetFunc: "onClickHandler",
      once:true});
  },
  onClickHandler: function() {
    alert("Node clicked");
  }
});
```

## Old-style DOM JavaScript event handling

Dojo positions itself as a *toolkit*, not a *framework*, so it allows you to use its subsystems only when you need them. One example of this is that you can still define event handlers the good-old DOM-style JavaScript way, like `this.domNode.onclick = function() {dojo.debug("Clicked");};`. There is nothing wrong with this old-style event handling, as long as you know what you are doing.

In some cases, extensive use of Dojo subsystems, like widget infrastructure or event handling, can lead an application to memory overflow and slow responsiveness. This is especially true in cases where you have a lot of visual components loaded at once. Trying to make each little component a Dojo widget and each event a Dojo event might overload the Dojo subsystems and the underlying browser's virtual machine.

Old-style event handling techniques can be easier for the JavaScript virtual machine of a browser to handle. The Dojo event handling system can make the developer's life easier. To find a balance between the two, use your personal judgment and common sense.

## JavaScript in HTML templates

The biggest problem with using JavaScript in HTML templates is that it breaks the conceptual consistency of the HTML template. But if you know what you're doing, you can still make it work. Here is an example:

```
<div onclick="dojo.widget.byId('${this.widgetId}').onClick();">Click me</div>
```

## Section 12. Development and debugging: Miscellaneous

In this final section of the tutorial, I'll discuss how to set up two popular browsers for Dojo widget development and debugging, then consider some of the issues that come up when you debug a Dojo widget. The section will conclude with some of the *gotchas* I encountered in my first attempts at developing HTML widgets using the Dojo toolkit.

### Browsers and development tools

#### Mozilla Firefox

Mozilla Firefox is typically considered to be the best browser for Web application development. Firefox has a nice JavaScript console, an integrated DOM inspector, and a lot of useful plugins. Firebug and the Web Developer Toolbar are the most popular plugins for Web development.

When installing Mozilla Firefox always pick the "Custom" installation and mark "Developer tools" -- you'll get a nice JavaScript Console and DOM Inspector.

You can do an interesting trick with Firefox. If you select an area on an HTML page, right-click and select **View Selection Source**, in most cases you'll see actual HTML generated by a Dojo widget. This is a very useful feature. Firebug plugin can show rendered HTML as well.

To download Firefox-related tools, see [Resources](#).

#### Microsoft Internet Explorer

Microsoft™ Internet Explorer™ is still a popular browser, so you need tools to test your Web applications in Internet Explorer as well. Some people prefer the Microsoft Script debugger to those for Firefox.

To download Internet Explorer-related tools, see [Resources](#).

### Debugging a Dojo widget

When you run a page with Dojo widgets, errors appear in two places:

- The JavaScript console (or popup/script debugger in Internet Explorer)

- The Dojo debug panel

It's important to monitor both.

It is possible to do a lot without debugging, and just use logging. The `dojo.debug("...");` function, which adds messages to a Dojo debug panel, is extremely useful. There are also `dojo.debugShallow(object);` and `dojo.debugDeep(object);` functions to dump JavaScript objects.

If you think an error might be caused by an old cached file (\*.js or \*.css), clean the browser cache (use the Web Developer Toolbar to do it faster) and/or refresh the page holding down the Ctrl key (theoretically refreshing with Ctrl should do the job, but sometimes it does not). Some people prefer to set their browser cache size to 0 to avoid caching at all. Note that this prevents file caching but not in-memory caching. Ctrl+refresh should help in this case. In some cases, especially when there's a memory leaking problem (on Microsoft Windows, use Task Manager to track this), restarting the browser completely is a good idea.

To learn more about debugging Dojo widgets, see [Resources](#).

## Gotchas

Here is a list of troubles I personally had when I started to use the Dojo toolkit. My assumptions were based on previous experience with browser-based JavaScript development. Some of these gotchas might bug you as well.

### Common errors and their solutions

Gotcha	Solution
The attributes of a DOM node used to define a widget seems lost (id, style)	These attributes can be preserved if needed. See <a href="#">Accessing widgets the old-style JavaScript way</a> and <a href="#">Nested HTML</a> topics for workarounds.
What does "this.domNode" mean?	<code>this.domNode</code> is a default <a href="#">attachment point</a> of the root DOM element in an HTML template.
Are setter methods called on widget creation?	No. <code>postCreate()</code> operates on attributes directly. See the <a href="#">Widget attributes</a> topic for more details.
Why are my attributes not passed to the widget on creation?	Because they are not defined in the widget JavaScript code file with default values.
<code>&lt;body onload="..."</code> does not work anymore	Use the Dojo event system instead. See <a href="#">Accessing widgets the old-style JavaScript way</a> for more details.
My CSS styles are messed up!	CSS classes names must be unique across the whole project, and should not interfere with Dojo CSS class names.

What is the difference between <code>dojo.byId()</code> and <code>dojo.widget.byId()</code> ?	<code>dojo.byId()</code> is an alias for <code>document.getElementById()</code> <code>dojo.widget.byId()</code> is an alias for <code>dojo.widget.manager.getWidgetById()</code>
Commented-out <code>dojo.require()</code> statements still work!	Dojo extracts these using regular expression. It is a known issue but is not critical.

---

## Section 13. Summary

Developing robust, stable, and high-quality software in JavaScript stands on the same principles as doing so with other programming languages. Applying development best practices, proven design patterns, and common sense makes a project successful.

### Further learning

If you need more information on developing Dojo widgets, now is a good time to start studying existing Dojo widgets code. You are already familiar with most of the important concepts used there. Also, see [Resources](#) for a list of articles and documents about using the Dojo toolkit.

That said, the dynamic nature of JavaScript makes it easier to get messy with than, for example, the Java language. The Java language is strongly typed; it pushes you to use namespaces (packages); and its Java runtime library was seemingly driven by the Gang of Four patterns and similar reference points. Also, Javadoc documentation is very good and usable. These factors are all extremely helpful for new developers, as long as they have less wheels to reinvent, and are pushed in the right direction from the start.

JavaScript is different. It is very flexible, sometimes too much so. You can do so many things with JavaScript, and in so many different ways. But after a while, who can maintain the code? If you've ever been a little *too* creative and/or inaccurate with JavaScript code, you probably know what I mean: maintenance can be tough even for the author of the code, after a while. JavaScript is far less forgiving than the Java language.

What can you do, then? One idea is to enforce known good practices using coding standards. You could even try to stick to *the Java way* of coding. Some decisions can really simplify things and help newcomers:

- Take advantage of Dojo's infrastructure for using packages.

- Define only one class per file and keep class and file names consistent.
- Validate arguments for correct types and not-null conditions explicitly at the beginning of public methods.
- Throw exceptions as you would in Java and view a stack trace with the Firebug plugin.
- Declare classes with `dojo.declare()`, and use the Dojo infrastructure for inheritance.

From another perspective, avoiding JavaScript's powerful flexibility can prevent really beautiful and worthy designs from happening. You don't want that either. So, as typically happens, it is all about finding a perfect balance between the extremes.

# Resources

## Learn

- [Dojo widgets FAQ](#): Find answers to common questions about Dojo HTML widgets, based on Dojo newsgroups.
- [Fast Widget Authoring with Dojo](#): In the first Dojo widgets tutorial, by Dojo Project Lead Alex Russell, learn to build Dojo widgets and quickly move from idea to upgrade-able interface.
- [The Dojo Event System](#): Dig into Dojo's broad view of events and the tools in `dojo.event.*` in this very good tutorial on the Dojo event system, also by Alex Russell.
- [Dojo nightly builds](#): Try these visual tests for standard Dojo widgets.
- [The Dojo Project's JotSpot wiki](#): Visit the home of the Dojo FAQ and find answers to questions about [using JavaScript in a namespace outside Dojo](#) ; [creating composite widgets](#); and [debugging Dojo](#).
- [developerWorks' Web development zone](#): Improve your work with info that specializes in Web architecture and development, featuring downloads and products, open source projects, a technical library, training, and events notices.
- [developerWorks technical events and webcasts](#): Stay current with jam-packed technical sessions that shorten your learning curve, and improve the quality and results of your most difficult software projects.

## Get products and technologies

- [Dojo homepage](#): Download the Dojo toolkit.
- [Mozilla Firefox](#): Try the following useful plugins for Web development in Firefox:
  - [Firefox Firebug plugin](#): Download a JavaScript development plugin that provides exception stack tracing, debugging, and a DOM inspector.
  - [Firefox Web Developer toolbar](#): Download more useful tools for JavaScript development.
  - [Venkman, the JavaScript debugger](#): Get a debugging tool for JavaScript in Firefox.
- [Internet Explorer](#): Check out the following plugins that can be useful for Web development in Internet Explorer:
  - [Internet Explorer developers toolbar](#): Explore many useful tools for JavaScript development.
  - [Internet Explorer 5.0 Web Developer Accessories](#): Download and try the

View Partial Source feature discussed in the tutorial. (See "[Powering up with Internet Explorer Extensibility](#)" to learn about the fix for Internet Explorer 6.)

- [Internet Explorer Microsoft Script Debugger](#): Use Internet Explorer itself to download and install this tool.
- [IBM trial software](#): Build your next development project with software available for download directly from developerWorks.

## Discuss

- [developerWorks Community](#): Visit blogs, forums, podcasts, and wikis hosted by and for developers.
- [developerWorks blogs](#): Get involved in the developerWorks community!

## About the author

### Igor Kusakov



Igor Kusakov joined the IT world as a professional ten years ago, and has been involved in developing Web and distributed applications since the late nineties. Igor is currently a consultant and leads Ajax Dojo-based application development at IBM Rational. Igor's primary professional interests are Java development, Web project architectures, rich JavaScript-based browser clients, modern persistence frameworks, and tooling for efficient development. Along with having a bachelors degree in computer science, Igor is certified in Java, C++, and C programming.