



## Documenting Code

Level: **Beginner**  
Created: **November 8, 2007**  
Author: **Johan Lund**  
Update Date: **November 8, 2007**  
Revision No: **1.0**

## INDEX

<b>Purpose</b> .....	<b>3</b>
<b>Tools</b> .....	<b>3</b>
<b>Workflow</b> .....	<b>3</b>
<b>Documentation Structure</b> .....	<b>4</b>
Documenting Arguments.....	4
Documenting Properties.....	4
Documenting Classes.....	5
Documenting Functions.....	6
<b>Restrictions</b> .....	<b>6</b>

## Purpose

If you are creating your own Bindows class libraries it might be useful to generate API documentation similar to Javadoc that can be viewed in any browser. We include with Bindows a tool that can be used to generate such documentation from comments in Bindows JavaScript. In order for the tool to work you must follow certain rules when you document your code. This paper teaches you those rules.

## Tools

The JsDoc tool that ships with all versions of Bindows up and to including version 3.0 is based on cscript and is therefore only usable in Microsoft Windows environments. We have since then ported the tool to pure Java and it is now platform independent. The new Java based JsDoc improves on several limitations that the cscript version had and this document is dedicated to the Java version only. The Java version of the JsDoc is currently available in our BindowsPlugin for IntelliJ IDEA or available upon request. For more information about the BindowsPlugin please visit this URL:

[http://www.bindows.net/documentation/tools/intellij\\_plugin/](http://www.bindows.net/documentation/tools/intellij_plugin/)

## Workflow

The tool takes your Bindows JavaScript files and outputs XML files. With a good style sheet, also supplied, you can view these xml files in any browser as if it was an HTML file.

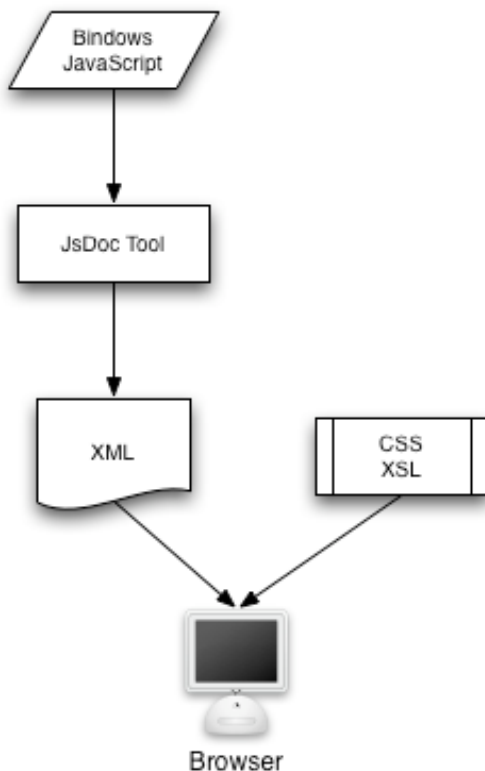


Figure 1. How JsDoc is generated.

## Documentation Structure

The Bindows API documentation is generated from comments in the code. The syntax is very much like Javadoc used to document Java code.

The documentation starts with a text which describes the documented object.

```
/** This is a JsDoc comment ... */  
/**  
 * ... this too ...  
 */
```

This text can be formatted using HTML tags. It can contain sample code and references (links) to external information. After that follows lines that start with keywords beginning with the character '@' that will be explained further.

### Documenting Arguments

Parameter/argument types are not declared in JavaScript. Instead we do it in the API documentation using the @param keyword. Declaring the type also gives hints for code completion to editors like IntelliJ IDEA.

```
* @param myParam : MyType      documentation
```

Parameters can be optional and have default values. They are written using the @optional and @default keywords respectively.

```
* @param myOptionalParam : String      documentation  
* @optional  
* @default "Hello World"
```

You also have the option to document the type parameter like this:

```
* @param {Aclass} p1 type before name
```

The reason we have this optional style is that some editors prefer this style when they parse documentation. We don't use it, IntelliJ IDEA understands both and it is ultimately your choice which style you want to use.

### Documenting Properties

If the tool finds an addProperty statement it will add property documentation. The value in the second parameter (e.g. Function.READ) determines if the values get and/or set will be checked.

If a method is identified as a getter or a setter, it will be documented as a property.

```
/** This is a get-only property which will be documented as someProperty */  
p.getSomeProperty = function () {/**...*/}
```

If a property is defined in several ways (e.g. both a getter and a setter method) only the documentation of the first occurrence will be used.

```
/** This is a get/set property which will be documented as someProperty */
p.getSomeProperty = function () {/**...*/};
/** This comment will be ignored */
p.setSomeProperty = function (value) {/**...*/};
```

Parameter values are not interesting for properties as they follow a strict structure. Instead we use the keyword `@type` to declare the return type of the getter/parameter type of the setter.

```
/**
 * This is a property
 * @type String
 */
p.getSomething = function () { return ""; }
```

It is also possible to combine `addMethod` statements with getter and setter methods. The following will create a property called “something” that has both read and write status in the API doc.

```
/**
 *This is a property
 *@type Number
 */
MyClass.addProperty("something", Function.READ);
p.setSomething = function(nSomething) {/**...*/};
```

## Documenting Classes

The documentation for a class is written in front of the constructor and method documentation is written just before the method.

```
/** This is documentation for the class MyClass */
function MyClass() {}
```

```
/**
 * This describes the MyClass class. See also
 * <link href="http://somewhere.com/stuff.html">this web resource</link>
 * for more information.
 * @param myParam : MyType documentation
 */
```

Events are documented in the constructor section only. If any of the member methods in the class is dispatching events, these should be documented in the class constructor in the following manner.

```
* @event eventname : MyType documentation
```

In the documentation of the constructor you also have the option to add remarks with `@remarks`.

```
* @remarks Some kind of remark...
```

## Documenting Functions

The way to document functions is similar to what we have seen so far. You can document both member methods and static methods in this way.

```
/** This is documentation for the instance method someMethod() */  
p.someMethod = function () {}
```

If we override a method from a superclass, we usually do not document it. To tell the tool to ignore the method (and not display an error message) we use `///  
@nodoc`.

```
///  
@nodoc  
p.overridingMethod = function () {}
```

```
/**  
 * This describes the myMethod method.  
 * @returns Boolean  
 */  
p.myMethod = function () { return true; }
```

## Restrictions

Methods should be written on the form:

```
Class.prototype.method = function () {};
```

instead of as closures in a constructor.

The Javascript syntax is not checked, but assumed to be correct.

Conditional inheritance is not supported. Use composition instead.

The order in which files are parsed is significant. If a constructor is parsed after one of its methods, that method will not be added to the resulting class XML file. However a class may be parsed before its superclass.

The HTML output includes hyperlinks to non-parsed content.

Superclasses that were not defined in the parsed Javascript code, will not appear in the class tree.