Namespace.js

# Introduction

Javascript provide OOP features but no namespace support. **Namespace.js** brings namespaces to Javascript in a simple and intuitive way. It also features remote loading of scripts.

**Namespace.js** is a small javascript script which provide namespacing utilities. It is framework independent. It also allows you to remotely include files.

Features:

* Simple API
* Framework independent
* Remote file loading (synchronously or async)
* Tested against Firefox 3.x, Safari 3.x, IE 6/7 and Opera 9.5
* Highly configurable
* Events
* Optionally add methods to native objects

# Creating Namespaces

Namespace can be created using the **Namespace()** function.

Namespace('com.example');  
com.example.MyClass = function() {};

Rather than defining all elements of your namespace like in the previous example you can specify as the second argument an object. All its properties will be added to the namespace.

Namespace('com.example', {  
 MyClass: function() {}  
});  
var obj = new com.example.MyClass();

Defining multiple times the same namespace won’t erase previously defined objects but add them.

Namespace('com.example', {  
 MyClass: function() {}  
});

Namespace('com.example', {  
 MySuperClass: function() {}  
});  
var obj = new com.example.MyClass();  
var obj2 = new com.example.MySuperClass();

The namespace separator can be modified by setting **Namespace.separator**.

Namespace.separator = '/';  
Namespace('com/example');

You can test if a namespace already exists by using **N*amespace.exist(*)**.

if (Namespace.exist('com.example')) {  
}

# Importing Objects

When using namespaces, it is a common practice to import commonly used objects in the global namespace to avoid typing the full qualified name each time. This can be achieved using **Namespace.use()**.

Namespace.use('com.example.MyClass');  
var obj = new MyClass();

You can also use the ***with*** javascript instruction for temporary importation.

with(com.example) {  
 var obj = new MyClass();  
}

You can import all elements of a namespace using ***\\**** as the last namespace segment.

Namespace.use('com.example.\*');  
var obj = new MyClass();

Instead of a string you can use an array to specify multiple namespaces to import.

Namespace.use(['com.example.FirstClass', 'com.example.SecondClass']);

# Including Remote Scripts

Namespace.js can include remote scripts using a namespace identifier with the **Namespace.include()** function.

Namespace.include('com.example');  
var obj = new com.example.MyClass();

The uri is constructed by converting dots to slashes and adding .js as the end. Thus in the previous example, the file would have been *com/example.js*. The function will return true for success, false otherwise.

In the previous example, the file is loaded synchronously. Asynchronous loading is also supported if a callback is defined as the second argument.

Namespace.include('com.example', function() {  
 var obj = new com.example.MyClass();  
});

A second callback can be defined as the third argument. It will be called only if an error occurs.

Namespace.include('com.example', function() {  
 var obj = new com.example.MyClass();  
}, function() {  
 alert('an error occured loading com.example');  
});

A base uri can be specified in **Namespace.baseUri**. It will be prepended to all uris. It must end with a slash.

Namespace.baseUri = './assets/js/';

The mapping between namespaces and filenames can be overridden by overriding the **Namespace.mapIdentifierToUri** function. It takes as parameter a namespace string and must return an uri.

Namespace.mapIdentifierToUri = function(identifier) {  
 return 'foo.js';  
};

Packaging multiple scripts into a single one for production is a common practice but can lead to problem when using includes. This is resolved using **Namespace.provide()**. Call this function with a namespace identifier specifying an already loaded namespace.

Namespace.provide('com.example');  
Namespace.include('com.example'); // won't load any script

An array can also be used to specify multiple namespaces at once.

# Auto Include

When using **Namespace.use()**, if the targeted element is not found, it will be automatically included by default. This can be disabled by setting false to **Namespace.autoInclude**. It can also be disabled on a pair call basis setting false as the third argument of **Namespace.use()**.

Namespace.use('com.example', false, false);  
// or  
Namespace.autoInclude = false;

The file can be included asynchronously by specifying a callback as the second argument.

// include async com/example/MyClass.js  
Namespace.use('com.example.MyClass', function() {  
 var obj = new MyClass();  
});

You can see that using **include()** with **use()** means that there must be one file per namespace element. This can be avoided by using **Namespace.from().use()**

// include async com/example.js  
Namespace.from('com.example').use('com.example.MyClass', function() {  
 var obj = new MyClass();  
});

The namespace in **use()** can be specified relatively to the namespace in **from()** when starting with a dot.

// include sync com/example.js  
Namespace.from('com.example').use('.MyClass');  
var obj = new MyClass();

When auto including, no error callback can be specified. You must use events (see below) to catch the errors. If the **\*** character is used, auto loading won’t be used.

Only elements from already included scripts will be imported.

# Events

Events are fired when actions occurred. You can register listeners with **Namespace.addEventListener()** and remove them using **Namespace.removeEventListener()**. Both functions take as first argument the event name and as second a function. The function can take as parameter an event object.

Namespace.addEventListener('includeError', function(event) {  
 alert('failed loading ' + event.uri);  
});

You can see all available events on their dedicated page

# Native Extensions

Optionally you can add methods to javascript’s native objects using **Namespace.registerNativeExtensions()**.

Methods are:

* String.namespace()
* String.use()
* String.include()
* String.from()
* String.provide()
* Array.use()
* Array.provide()

There are the same as their **Namespace.\*** equivalent but do not take the namespace identifier as the first argument.

'com.example'.namespace({  
 MyClass: function() {}  
});

'com.example.MyClass'.use();  
var obj = new MyClass();

['com.example.FirstClass', 'com.example.SecondClass'].use()

'com.remote'.include();

# Examples

Here are some examples for using Namespace.js.

Namespace('foo.bar');  
foo.bar.myFunction = function() {};

Namespace('com.example', {  
 MyClass: function() { return {}; }  
});  
var obj = new com.example.MyClass();

Namespace.use('com.example.MyClass');  
var obj2 = new MyClass();

// include com/example/RemoteClass.js  
Namespace.include('com.example.RemoteClass');  
var obj3 = new com.example.RemoteClass();

Namespace.registerNativeExtensions();  
'com.foo.bar'.namespace();