ZINCKSOFT

Give The Dream New Life.

JS Utils 1.0

The complete Core API Reference

Quick Introduction

JS Utils (also called JSU) is an extensible module-based object-oriented JavaScript framework aimed to bring developers new functionality at hand, and ease the development process by offering a wide range of time-saving methods for native objects, but also new objects that ease the coding process for everyday web applications, like timers and customizable password strength meters.

Even if it's the first release of this framework, it was put through several unit tests and succeded on all major browsers, basically because it is a framework that enhances coding experience and not UI design, thus working on almost[1] any browser that supports JavaScript.

Features and Compatibility

The main features of the JSU framwork are:

- *object-oriented* and *module-based design*;
- *extensible* (allows you to extend it with new modules);
- *integrates* with any JavaScript framework;
- native objects are not affected (the framework avoids using the **prototype** property on these objects);
- most methods have one or more *aliases* so you can call them the way it makes sense to you;
- supports *method chaining* for instantiable modules, allowing you to write less code;
- it was *successfully tested* on Safari 3+, Firefox 1.5+, Chrome 3+, Opera 9.5+, Camino 1+ and Internet Explorer 7+.

[1] The unit tests weren't successful on very old browsers due to old versions of JavaScript implemented on them.

Bundled Modules

The JSU framework comes with several modules along the core module to get you started coding smarter. Below is the list of these modules:

- **JSU Core** / **JSU** (*jsu.core.js*) - contains the core module that will be used by every derived module;
  - **JSU.String / JSUString** - this module comes with the core file because it offers string manipulation methods used by all other modules; the reason this object wasn't separated from the core file is that almost any module is using it, so it's better to minimize the number of JavaScript files the developer has to include in a web page;
  - **JSU.Array / JSUArray** - this module offers new functionality to JavaScript arrays;
  - **JSU.Object / JSUObject** - a basic module with just a few methods used to work with any object, like methods for comparing objects and verifying if they are empty or not;

- **JSU.Color / JSUColor** (*jsu.color.js*) - this modules is used for converting colors between different color spaces like RGB, CMYK, HSL, HEX, and so on (it supports conversions between 12 different color spaces);
- **JSU.Dictionary / JSUDictionary** (*jsu.dictionary.js*) - it is known that JavaScript can't handle associative arrays by default, and there were various implementations to offer the functionality provided by this kind of arrays, so this module is yet another implementation to associative arrays with two big differences: it has a lot of methods that let you do almost anything you want with them and is inspired by C#'s Dictionary object;

- **JSU.List / JSUList** (*jsu.list.js*) - this module offers an alternative to JavaScript arrays, and a pretty big one if we're looking at the impresive number of methods it has; just like the Dictionary module, this one is also inspired by C#'s List object and offers even more methods to work with them;
- **JSU.Misc / JSUMisc** (*jsu.misc.js*) - module with some miscellaneous methods for various things like unix permissions conversions, css unit parsing, and so on;
- **JSU.PassMeter / JSUPassMeter** (*jsu.password.js*) - this module is used to measure the strength of passwords with a custom set of rules or with your own;
- **JSU.Set / JSUSet** (*jsu.set.js*) - like the List module, this one is used for sets (lists of unique objects) that allow you to manipulate them using operations like subtraction / union / intersection with other sets;
- **JSU.Timer / JSUTimer** (*jsu.timer.js*) - based on the JavaScript methods setInterval and setTimeout, this object mimics the Timer objects from modern languages like Delphi, C#, Java, etc;
- **JSU.URL / JSUUrl** (*jsu.url.js*) - the module is used for URL handling and parsing;
- **JSU.Validation / JSUValidation** (*jsu.validation.js*) - the validation module allows you to validate user input with ease, offering you lots of methods for different needs.

The list isn't that impressive, but the framework does its job very well, and most of all it allows you to develop your own modules.

To find out how you can create your own modules, I've included a tutorial at the end of this book on developing your own module.

Including modules in your page

No matter what modules you use, first you'll always have to include the core module and then any other modules, like this:

```
<script type="text/javascript" src="jsu.core.js"></script>

<script type="text/javascript" src="jsu.timer.js"></script>
<script type="text/javascript" src="jsu.validation.js"></script>
```

Core API Reference

The core object (the JSU object) is the foundation of all modules, allowing you to extend it with new modules by using several of its methods. Also, it's a static class and cannot be instantiated.

A. Core Properties

1. nullRegExp

This property is the equivalent of **null** but for regular expressions (RegExp objects). Using it won't match only empty strings and nothing else.

Example:

```
return JSU.nullRegExp.test("a string");  // this will return false
return JSU.nullRegExp.test("");  // this will return true
```

2. __types

This is a private property (an array) which is used to define new objects so that the core module can correctly identify them using the **getType** method. See the last chapter of this book on how to define a new type of JSU object.

B. Core Methods

1. extend

The **extend** method is generally used when adding new functionality to the core module or when adding new types of JSU objects, but can be also used when you need to extend custom objects by adding new properties and/or methods to it. Basically, the core module defines the JSU object which is a regular JavaScript object, but it's used by all modules - thus naming it the *core* object of the framework.

Syntax:

```
JSU.extend(jsu_extension)
JSU.extend(target, extension1 [, extension2 [, extension3 ...]]);
```

Arguments:

- ***jsu_extension*** : represents an object containing new properties and/or methods that will be added to the core object; when the **extend** method is used with a single argument, it will consider the target object as being the core object and will accept only one extension at a time;
- ***target***: when this method is used with more than one argument, the first argument becomes the target object, and the following arguments will be treated as extensions to be added to the target object;
- ***extensionN***: if more than one argument is passed to this method, you can add as many extensions to the arguments list as you want.

Returns: Either the extended core object if one argument is provided, or the extended target object if there's more than one argument available.

Example:

```
var obj = JSU.extend( { myProperty: 2 } );
alert(obj.myProperty); // will output 2

JSU.extend( { myPropert: 2 } );
alert(JSU.myProperty); // will also output 2
```

Here you can see that you can use an alias to the core object by declaring it as a new variable (first example), but you can also use it directly (second example).

```
var obj = JSU.extend({ }, { prop1: true, prop2: 3 }, { prop3: false });
alert(obj.prop2); // will output 3
alert(obj.prop3); // will output false

JSU.extend(obj, { prop4: "string", prop1: false } );
alert(obj.prop4); // will output "string"
alert(obj.prop1); // will output false
```

In the first three lines of code, the first argument of the method is an empty object which will be extended by adding new properties to it. In the last three lines, we further extend the

previous object by adding a new string property, and then changing an existing property by adding an object containing a property with the same name as an existing one.

Be careful with this method because it can replace existing properties and methods, and you don't want to extend, for example, the core object by adding methods with the same name as the existing ones, because you'll end up with a broken framework and you won't know why everything stopped working and errors pop from everywhere.

2. __addExtensions__

This is a private method used internally by the **extend** method of the core object. It is used when adding extensions to an object, one extension at a time.

3. addType

Adds a new type of JSU object to the core module so that you can easily identify them globally. This method is used when extending the JSU framework with new modules that contain new JSU objects that need to be identified correctly (by default, the type will be *object* for new JSU objects).

Syntax:

```
addType(objectClass, type)
```

Arguments:

- ***objectClass***: the object's name (its class, not an instance of it) that will be used when getting the type of an object;
- ***type***: a string (preferably a lower case string) with the name of the type, like *string*, *number*, *myobject*, etc.

Returns: Nothing

Example:

```
function myObjectClass() {
    //initializations
};

JSU.addType(myObjectClass, "myobject");
alert(JSU.getType(new myObjectClass())); // will output "myobject"
```

I've declared a new JavaScript class for a custom JSU object, and after I've added a new type for it, the getType method of the core module will return the correct type of the object, that is *myobject*, and not *object* which is the default for new objects.

4. regExpEscape

Escapes a character if it's a regular expression reserved character, or if a string is passed as the argument it will escape all the reserved RegExp characters that it finds and will return a new string with escaped characters where needed.

Syntax:

```
regExpEscape(arg)
```

Arguments:

- ***arg***: a character or a string which will be parsed and escaped.

Returns: A character or a string, depending on the argument's value.

Example:

```
alert(JSU.regExpEscape("a")); // outputs  "a"
alert(JSU.regExpEscape(".")); // outputs  "\."
alert(JSU.regExpEscape("test[now]")); // outputs  "test\[now\]"
alert(JSU.regExpEscape("string")); // outputs  "string"
```

5. isFunction, areFunctions

The first one tests if a given argument is a JavaScript function, and the second tests if all the given arguments are JavaScript functions.

Syntax:

```
isFunction(arg)
areFunctions(arg1 [, arg2 [, arg3 ...]])
```

Arguments:

- ***arg***: an argument to test if it's a function;
- ***argN***: list of arguments to test if they are all functions.

Returns: Boolean

Example:

```
alert(JSU.isFunction( new Function())); // outputs true
alert(JSU.isFunction( function() { } )); // outputs true

var f = function(arg) { }, f2 = f;
alert(JSU.areFunctions(f, f2)); // outputs true

var o = new Object();
alert(JSU.areFunctions(o, JSU)); // outputs false
```

As you can see, various function declaration syntaxes are supported. The last call in this example outputs false because none of the two arguments are functions - o is an instance of Object, and JSU is also an object (so you cannot instantiate the core object).

6. other **is/are** methods

The core object also defines pairs of is/are methods for the following JavaScript objects:

- *Boolean*: **isBoolean / areBooleans**
- *String*: **isString / areStrings**
- *Number*: **isNumber / areNumbers**
- *Object*: **isObject / areObjects**
- *RegExp*: **isRegExp / areRegExps**
- *Date*: **isDate / areDates**

- *Predicate*: **isValidPredicate / areValidPredicates**
- *Undefined*: **isUndefined / areUndefined**

The first six pairs of methods have the same argument list and return type as the isFunction / areFunctions pair previously described. The last two were separated from the list because they don't represent any object.

The **isUndefined / areUndefined** pair of methods are used to check whether the arguments are defined or not (very useful when creating functions with optional arguments).

The **isPredicate / arePredicates** pair of methods are used to check whether the arguments are valid non-null functions (often used as callback methods in functions' arguments).

7. __getType

This is a private method used internally by the **getType** method of the core object. It handles native JavaScript types, while the **getType** method handles the user defined types.

8. getType

This method is used to get the type of an object, either a native object or a JSU object.

Syntax:

```
getType(obj)
```

Arguments:

- ***obj***: the object for which the type is needed

Returns: A lowercase string with the type's name.

Example:

```
alert(JSU.getType("a text")); // outputs "string"
alert(JSU.getType(2)); // outputs "number"
alert(JSU.getType(new Number(3))); // outputs "number"
alert(JSU.getType([])); // outputs "array"

alert(JSU.getType(obj)); // outputs "undefined" (obj is not declared)


function myObject() { };
var obj = new myObject();

alert(JSU.getType(obj)); // outputs "object"

JSU.addType(myObject, "myobject");
alert(JSU.getType(obj)); // outputs "myobject"
```

As you can see, it recognizes native objects no matter how you declare them, but to recognize a custom object, you need to tell JSU how to recognize it, and that is by adding the object's type to JSU's custom types using the **addType** method.

9. setDecimals

This method allows you to format numbers so they have a specified number of decimals.

Syntax:

```
setDecimals(nr [, decimals])
```

Arguments:

- ***nr***: the number which will be formatted - can be any integer / float / string containing a number, but remember that the integer won't get any formatting because it doesn't have any decimals being an integer; also, if the number has by default a lower number of decimals than the desired one, the method will ignore this argument, and if it has a larger number of decimals, the method will round the number to the desired number of decimals;
- ***decimals***: the number of decimals to show (it is optional, the default being 2 decimals).

Returns: A float number.

Example:

```
alert(JSU.setDecimals(10, 2)); // outputs 10
alert(JSU.setDecimals(10.0751, 2)); // outputs 10.08
alert(JSU.setDecimals(10.074, 2)); // outputs 10.07
alert(JSU.setDecimals("-.574")); // outputs -0.57
alert(JSU.setDecimals(-14.276, 4)); // outputs -14.276
```

JSU.Object API Reference

The JSU.Object is a static class (meaning it cannot be instantiated) allowing you to compare basic objects and verify if they're empty or not.

Methods

1. isEmpty

Checks whether the given argument is an empty object. If it's not an object or if it has at least one property/method, it will return false. Also, if the argument is null, the method will also return true.

Syntax:

```
isEmpty(obj)
```

Arguments:

- ***obj***: the object which will be tested.

Returns: Boolean

Example:

```
alert(JSU.Object.isEmpty(new Object())); // true
alert(JSU.Object.isEmpty(2)); // false
alert(JSU.Object.isEmpty(null)); // true
alert(JSU.Object.isEmpty( { } )); // true
alert(JSU.Object.isEmpty( { a: 2 } )); // false
```

2. equals

This method will check if two objects are equal. In JavaScript, there are two types of equality, the normal equality and the strict one. The normal equality (==) will return true for objects that have the same value but different types (eg. 2 == 2 and 2 == "2"), while the strict equality (===) will return true only for objects that have the same value and type (2 == 2 and 2 != "2"). The **equals** method is the equivalent of the normal equality check.

Syntax:

```
equals(obj1, obj2)
```

Arguments:

- ***obj1*** and ***obj2***: the objects which will be compared.

Returns: Boolean

Example:

```
var o1 = new Object(), o2 = new Object();

alert(JSU.Object.equals(o1, o2)); // true
alert(JSU.Object.equals(o1, { })); // true
alert(JSU.Object.equals(o1, { a: 2 })); // false
alert(JSU.Object.equals(2, 2)); // true
alert(JSU.Object.equals(2, "2")); // true
alert(JSU.Object.equals(new Number(2), "2")); // true
alert(JSU.Object.equals({ a: 2 }, { a: "2" })); // true
alert(JSU.Object.equals({ a: 2 }, { a: 3})); // false
alert(JSU.Object.equals({ a: 2 }, { a: 2, b: 3})); // false
alert(JSU.Object.equals(null, null)); // true
alert(JSU.Object.equals(null, "")); // false
alert(JSU.Object.equals("", "")); // true
```

3. same

Like the equals method, this method checks the equality between two objects but with one difference: it will use the strict equality test.

Syntax:

```
same(obj1, obj2)
```

Arguments:

- ***obj1*** and ***obj2***: the objects which will be compared.

Returns: Boolean

Example:

```
var o1 = new Object(), o2 = new Object();

alert(JSU.Object.same(o1, o2)); // true
alert(JSU.Object.same(o1, { })); // true
alert(JSU.Object.same(o1, { a: 2 })); // false
alert(JSU.Object.same(2, 2)); // true
alert(JSU.Object.same(2, "2")); // false
alert(JSU.Object.same(new Number(2), "2")); // false
alert(JSU.Object.same({ a: 2 }, { a: "2" })); // false
alert(JSU.Object.same({ a: 2 }, { a: 3})); // false
alert(JSU.Object.same({ a: 2 }, { a: 2, b: 3})); // false
alert(JSU.Object.same(null, null)); // true
alert(JSU.Object.same(null, "")); // false
alert(JSU.Object.same("", "")); // true
```

JSU.Array API Reference

This JSU Object (also a static class) offers you new methods for working with arrays, most of all being crucial for any developer.

Methods

1. isEmpty

Checks whether the given argument is an empty array. If it's not an arary or if it's null, it will return true for safety reasons (so that you won't start doing something with your non-array array, because you would get errors while using it). Also, if it's an empty array, it wil return true. The only time it returns false is when you pass a non-empty array object.

Syntax:

```
isEmpty(arr)
```

Arguments:

- ***arr***: the object which will be tested.

Returns: Boolean

Example:

```
alert(JSU.Array.isEmpty([])); // true
alert(JSU.Array.isEmpty([2, 3])); // false
alert(JSU.Array.isEmpty([null])); // false
alert(JSU.Array.isEmpty(null)); // true
alert(JSU.Array.isEmpty(new Array())); // true
alert(JSU.Array.isEmpty( { } )); // true
```

2. **equals** and **same**

Like the JSU.Object's methods with the same name (see the *JSU.Object API Reference* previously in this book for more details), these two methods will test for equality (normal or strict) between two arrays. The syntax and arguments are the same, and the return type is also a boolean.

Example:

```
alert(JSU.Array.equals([], [])); // true
alert(JSU.Array.equals([], [2, 3])); // false
alert(JSU.Array.equals([], [null])); // false
alert(JSU.Array.equals(null, [null])); // false
alert(JSU.Array.equals([null], [null])); // true
alert(JSU.Array.equals([""], [""])); // true
alert(JSU.Array.equals([""], [])); // false
alert(JSU.Array.equals([2, 3], [2, 3])); // true
alert(JSU.Array.equals([2, "3"], ["2", 3])); // true

alert(JSU.Array.same([], [])); // true
alert(JSU.Array.same([], [2, 3])); // false
alert(JSU.Array.same([], [null])); // false
alert(JSU.Array.same(null, [null])); // false
alert(JSU.Array.same([null], [null])); // true
alert(JSU.Array.same([""], [""])); // true
alert(JSU.Array.same([""], [])); // false
alert(JSU.Array.same([2, 3], [2, 3])); // true
alert(JSU.Array.same([2, "3"], ["2", 3])); // false
```

3. copy

Offers you the posibility to copy the contents of an array to another without referencing it like JavaScript is doing by default.

Syntax:

```
copy(arrFrom)
```

Arguments:

- ***arrFrom***: the array which will be copied;

Returns: in normal conditions, this method will return a copy of the array; if *arrFrom* is null, the method returns also null, and if it's a non-array object, it will return an empty array.

Example:

```
var a1 = [1, 2, 3],
    a2 = [4, 5],
    a4;

a4 = JSU.Array.copy(a1); // a4 is a copy of a1, and not a reference
a4 = JSU.Array.copy([]); // a4 is now empty
a4 = JSU.Array.copy(a2); // a4 is now a copy of a2
```

4. **contains** and **containsExactly**

These methods check whether an object is contained within an array. The difference between these two methods is that the first will use the normal equality to find out if an array contains the specified object, while the last one will use the strict equality.

Syntax:

```
contains(arr, obj)
containsExactly(arr, obj)
```

Arguments:

- ***arr***: the array that will be used to search for matches;
- ***obj***: the object checked for existence in the array.

Returns: Boolean

Example:

```
var a = [1, 2, "3", null, [], [1, 2], [1, "3"] ];

alert(JSU.Array.contains(a, 1)); // true
alert(JSU.Array.contains(a, 3)); // true
alert(JSU.Array.contains(a, null)); // true
alert(JSU.Array.contains(a, [])); // true
alert(JSU.Array.contains(a, [1, "2"])); // true
alert(JSU.Array.contains(a, [1, "3"])); // true
alert(JSU.Array.contains(a, 0)); // false
```

```
alert(JSU.Array.containsExactly(a, 1)); // true
alert(JSU.Array.containsExactly(a, 3)); // true
alert(JSU.Array.containsExactly(a, null)); // true
alert(JSU.Array.containsExactly(a, [])); // true
alert(JSU.Array.containsExactly(a, [1, "2"])); // false
alert(JSU.Array.containsExactly(a, [1, "3"])); // true
alert(JSU.Array.containsExactly(a, 0)); // false
```

5. indexesOf

Searches an object within an array and returns another array with all the positions where the object was found. The method will use the strict equality check.

Syntax:

```
indexesOf(arr, obj)
```

Arguments:

- ***arr***: the array used to search the positions;
- ***obj***: the object that will be searched within the array.

Returns: An array with all the indexes where the object was found within the given array, or an empty array if the object wasn't found.

Example:

```
var a = [1, 2, 3, 1, [], 1, "1", [], null, [1,2], 1];

alert(JSU.Array.indexesOf(a, 1)); // [0, 3, 5, 10]
alert(JSU.Array.indexesOf(a, [])); // [4, 7]
alert(JSU.Array.indexesOf(a, null)); // [8]
alert(JSU.Array.indexesOf(a, -1)); // []
alert(JSU.Array.indexesOf(a, "1")); // [6]
alert(JSU.Array.indexesOf(a, [1,2])); // [9]
```

6. createWith

Allows you to create and initialize an array with a specified size containing specific values.

Syntax:

```
createWith(obj, size)
createWith(predicate, size)
```

Predicate syntax:

```
function(index)
```

Arguments:

- ***obj***: the object which will be used to fill the array;
- ***size***: the size of the new array;
- ***predicate***: the function that returns the values used to fill the array;

Returns: A new array.

Example:

```
var a;
a = JSU.Array.createWith(-1, 4); // [-1, -1, -1, -1]

var p = function(i) {
    return "value " + (i+1);
};
a = JSU.Array.createWith(p, 3); // ["value 1", "value 2", "value 3"]
```

JSU.String API Reference

Methods

1. toInt, toFloat and toNumber

These three methods allow you to parse strings to integers or float numbers using the decimal base for converting. The first two are self explanatory, while the last one will try to convert the string into an integer (it checks the string's format to see if it's an integer or a float) and if it doesn't succeed, it will try to convert it to a float.

Syntax:

```
toInt(str)
toFloat(str)
toNumber(str)
```

Arguments:

- ***str***: the string to convert into a number.

Returns: toInt always returns an integer, toFloat a float and toNumber returns an integer or a float, depending on the string's format.

Example:

```
alert(JSU.String.toInt("2")); // 2
alert(JSU.String.toInt(2)); // 2
alert(JSU.String.toInt("2.5px")); // 2
alert(JSU.String.toInt("-.5")); // NaN
alert(JSU.String.toInt("0.51")); // 0

alert(JSU.String.toFloat("2")); // 2
alert(JSU.String.toFloat("2.51")); // 2.51
alert(JSU.String.toFloat("-.27")); // -0.27

alert(JSU.String.toNumber("2")); // 2
alert(JSU.String.toNumber(2.67)); // 2.67
alert(JSU.String.toNumber("-2.411")); // -2.411
alert(JSU.String.toNumber("0.5px")); // 0.5
```

2. **trim**, **trimLeft** and **trimRight**

Allows you to trim spaces from the ends of a string, whether it's a single line or a string with multiple lines (by trimming the new line character, too). If you want to trim a string with multiple lines without trimming the new line character, use the **strip** methods.

Syntax:

```
trim(str)        //trims both ends
trimLeft(str)    //trims only the left side
trimRight(str)   //trims only the right side
```

Arguments:

- ***str***: the string to be trimmed.

Returns: A trimmed string.

Example:

```
alert(JSU.String.trimLeft("  test  ")); // "test  "
alert(JSU.String.trimLeft("test  ")); // "test  "
alert(JSU.String.trimLeft("  \n  test  ")); // "test  "
alert(JSU.String.trimLeft("  \r\ntest  ")); // "test  "

alert(JSU.String.trimRight("  test  ")); // "  test"
alert(JSU.String.trimRight("test  ")); // "test"
alert(JSU.String.trimRight("  \n  test \n  ")); // "  \n  test"
alert(JSU.String.trimRight("  \r\ntest \n\r ")); // "  \r\ntest"

alert(JSU.String.trim("  test  ")); // "test"
alert(JSU.String.trim("test  ")); // "test"
alert(JSU.String.trim("  \n  test \n  ")); // "test"
alert(JSU.String.trim("  \r\ntest \n\r ")); // "test"
```

3. **strip**, **stripLeft** and **stripRight**

Allows you to trim a specified character or a substring from the ends of a string.

Syntax:

```
strip(str [, character])      //strips both ends
strip(str [, substring])
stripLeft(str [, character]) //strip only the left side
stripLeft(str [, substring])
stripRight(str [, character]) //strip only the right side
stripRight(str [, substring])
```

Arguments:

- ***str***: the string to strip;
- ***character***: the character to be stripped; optional, the default being an empty space;
- ***substring***: the substring to be stripped; optional, the default being an empty space.

Returns: A stripped string.

Example:

```
alert(JSU.String.stripLeft("   test ", " ")); // "test"
alert(JSU.String.stripLeft("       ")); // an empty string
alert(JSU.String.stripLeft("^^ ^test|^", "^")); // " ^test|^"
alert(JSU.String.stripLeft("|^||^||test|^|", "|^|")); // "|test|^|"

alert(JSU.String.stripRight("   test ", " ")); // "   test"
alert(JSU.String.stripRight("       ")); // an empty string
alert(JSU.String.stripRight("^^ ^test|^", "^")); // "^^ ^test|"
alert(JSU.String.stripRight("|^||^||test|^|", "|^|")); // "|^||^||test"

alert(JSU.String.strip("   test ", " ")); // "test"
alert(JSU.String.strip("       ")); // an empty string
alert(JSU.String.strip("^^ ^test|^", "^")); // " ^test|"
alert(JSU.String.strip("|^||^||test|^|", "|^|")); // "|test"
```

4. stripAndTrim

Strips and also trims a string, but the new lines will not be trimmed.

Syntax:

```
stripAndTrim(str [, character])
stripAndTrim(str [, substring])
```

Arguments:

- ***str***: the string to be stripped and trimmed;
- ***character***: the character to be stripped; optional, the default being an empty space;
- ***substring***: the substring to be stripped; optional, the default being an empty space.

Returns: A stripped and trimmed string.

Example:

```
alert(JSU.String.stripAndTrim("aaa   testa a ", "a")); // "test"
alert(JSU.String.stripAndTrim("       ")); // an empty string
alert(JSU.String.stripAndTrim(" tt estatt  tt", "tt")); // "esta"
```

5. getTextColumn

Gets a column of text from a string with one or more lines of text. If a line of the string has less characters than the number of capture characters + skip characters, the column's line will contain that entire line.

Syntax:

```
getTextColumn(str, skip, capture)
```

Arguments:

- ***str***: the string from which the column will be extracted;
- ***skip***: number of characters to skip at the beginning of each line;
- ***capture***: number of characters to extract (the column's characters).

Returns: A string with one or more lines containing the extracted column.

Example:

```
alert(JSU.String.getTextColumn("a test a", 2, 4)); // "test"
alert(JSU.String.getTextColumn("a test a\n", 2, 4)); // "test\n"
alert(JSU.String.getTextColumn("a tesA a\nb tes\n\nc tesC c",
            2, 4)); // "tesA\ntes\ntesC"
```

6. getTextBlock

Similar to getTextColumn, this method extracts a block of text from a string (just a number of lines of a column, not the entire column unless specified so).

Syntax:

```
getTextBlock(str, skipLines, skipChars, captureLines, captureChars)
```

Arguments:

- ***str***: the string from which the column will be extracted;
- ***skipLines***: number of lines to skip from the string;
- ***skipChars***: number of characters to skip at the beginning of each line;
- ***captureLines***: number of lines to extract from the string;
- ***captureChars***: number of characters to extract (the column's characters).

Returns: A string with one or more lines containing the extracted column.

Example:

```
alert(JSU.String.getTextBlock("a test a",
            0, 2, 1, 4)); // "test"
alert(JSU.String.getTextBlock("a tesA a\nb tes\n\nc tesC c",
            1, 2, 4, 4)); // "tes\ntesC"
alert(JSU.String.getTextBlock("a tesA a\nb tes\n\nc tesC c",
            0, 2, 0, 4)); // an empty string (captureLines = 0)
```

7. containsWord and containsWords

These methods allow you to check if a string contains one or more words (not substrings, but words delimited by various punctuation characters or spaces).

Syntax:

```
containsWord(str, word)    //check if the word is found
containsWords(str, words)  //check if all words are found
```

Arguments:

- ***str***: the string used for searching;
- ***word***: a string containing the word to search;
- ***words***: an array of strings with the words to search.

Returns: Boolean

Example:

```
alert(JSU.String.containsWord("testing word now", "word")); // true
alert(JSU.String.containsWord("testing aword", "word")); // false
alert(JSU.String.containsWord("   ", " ")); // false - no word specified

alert(JSU.String.containsWords("testing word and now another word",
            ["word", "testing"])); // true
alert(JSU.String.containsWords("testing word and now another word",
            ["word", "tester"])); // false
```

8. pad, padLeft and padRight

These methods add characters to the sides of a given string so that its length matches a given size. Excepting the **pad** method, the other two also support padding a string with another string.

Syntax:

```
pad(str, char, size)                    //pad on both sides
padLeft(str, char, size)                //pad only on the left side
padLeft(str, substr, size)
padRight(str, char, size [,right])      //pad only on the right side
padRight(str, substr, size [,right])
```

Aliases:

- **padBoth** - alias for **pad**

Arguments:

- ***str***: the string which will be padded;
- ***char***: a character used to pad;
- ***substr***: a string used to pad.

- ***size***: the desired size of the string including padding; a size lower than or equal to the given string will make the method return that string without any padding;
- ***right***: a boolean indicating whether the right side has priority over the left side when padding; optional, the default being false.

Returns: A padded string.

Example:

```
alert(JSU.String.padLeft("testing", "0", 4)); // "testing"
alert(JSU.String.padLeft("test", " ", 5)); // " test"
alert(JSU.String.padLeft("test", "a|", 6)); // "a|test"
alert(JSU.String.padLeft(7, "0", 3)); // "007"

alert(JSU.String.padRight("testing", "0", 4)); // "testing"
alert(JSU.String.padRight("test", " ", 5)); // "test "
alert(JSU.String.padRight("test", "a|", 6)); // "testa|"
alert(JSU.String.padRight(7, "0", 3)); // "700"

alert(JSU.String.pad("test", "0", 7)); // "00test0"
alert(JSU.String.pad("test", "0", 7, true)); // "0test00"
```

9. back2slash and slash2back

Useful for converting between backslashes and slashes, for example in filenames.

Syntax:

```
back2slash(str)  //converts backslashes to forward slashes
slash2back(str)  //convers forward slashes to backslashes
```

Arguments:

- ***str***: the string that will be converted.

Returns: A converted string.

Example:

```
alert(JSU.String.back2slash("test\\")); // "test/"
alert(JSU.String.back2slash("test\\a//")); // "test/a//"

alert(JSU.String.slash2back("test//")); // "test\\\\"
alert(JSU.String.slash2back("test/a\\")); // "test\\a\\"
```

10. clean

Removes characters or substrings from a string.

Syntax:

```
clean(str, list)
```

Arguments:

- ***str***: the string to be cleaned;
- ***list***: an array of characters or strings to be removed from the given string.

Returns: A clean string.

Example:

```
JSU.String.clean("test, another ? yey !",
          [",", "?"])); // "test another  yey !"
JSU.String.clean("    ", [" "])); // an empty string (same as trimming)
JSU.String.clean("test,, another ?! wow!",
          ["?!", ",,"])); // "test another  wow!"
```

11. br2return and return2br

Useful for converting between new line characters (\n, \r, \n\r, \r\n) and HTML new line tags (*
, < br >,
 or
*).

Syntax:

```
br2return(str)
return2br(str)
```

Arguments:

- ***str***: the string to be converted.

Returns: A converted string.

Example:

```
alert(JSU.String.br2return("test\n<br />aha")); // "test\n\raha"
alert(JSU.String.return2br("test\n<br/>aha")); // "test<br/><br/>aha"
```

12. getPhoneDigit

A method useful when you need to convert letters to phone digits.

Syntax:

```
getPhoneDigit(character)
```

Arguments:

- ***character***: the letter to convert into a phone digit.

Returns: A number with the corresponding digit of a standard phone, or -1 if *character* is not an alphabet letter.

Example:

```
alert(JSU.String.getPhoneDigit("a")); // 2
alert(JSU.String.getPhoneDigit("b")); // 2
alert(JSU.String.getPhoneDigit("cc")); // -1
```

13. toProperCase

Converts the first character of a word to upper case but only if it the first character is a letter.

Syntax:

```
toProperCase(word)
```

Arguments:

- ***word***: the word to convert to proper case.

Returns: A string containing the proper case word.

Example:

```
alert(JSU.String.toProperCase("aha")); // "Aha"
alert(JSU.String.toProperCase("  a")); // "  a"
```

14. groupDigits

Method useful for grouping digits within numbers (floats and integers).

Syntax:

```
groupDigits(nr [, groupSeparator])
```

Arguments:

- ***nr***: the number, either a string containing a decimal number, or a standard decimal number (float or integer);
- ***groupSeparator***: the character used to group the digits (default is "**.**").

Returns: A string containing the formatted number.

Example:

```
alert(JSU.String.groupDigits(-1.111)); // -1.111
alert(JSU.String.groupDigits(111.0)); // 111
alert(JSU.String.groupDigits("1111,0")); // 1,111.0
alert(JSU.String.groupDigits("11111,0", ",")); // 11,111.0
alert(JSU.String.groupDigits("11111.0", ",")); // 11,111.0
alert(JSU.String.groupDigits(11111.10, ".")); // 11.111,1
alert(JSU.String.groupDigits(123456)); // 123,456
alert(JSU.String.groupDigits(-0.1234)); // -0.1234
alert(JSU.String.groupDigits(-1111.1111)); // -1,111.1111
```

15. count

Allows you to return the number of occurences for a specific substring within a given string.

Syntax:

```
count(s, sub [,deep])
```

Arguments:

- ***s***: the string to search within;
- ***sub***: the substring to search for;
- ***deep***: if true, it allows you to find substrings within substrings (see examples); the default is false.

Returns: The number of occurences.

Example:

```
alert(JSU.String.count("ahat", "ha")); // 1
alert(JSU.String.count("tesestes", "es")); // 3
alert(JSU.String.count("repeater", "e")); // 3
alert(JSU.String.count("ciu", "a")); // 0
alert(JSU.String.count("atttatt", "tt")); // 2
alert(JSU.String.count("atttatt", "tt", true)); // 3
```

16. indexesOf

Used to find all the indexes of a specific substring within a given string. Same as **count**, but returns an array with all indexes at which the substring was found instead the number of occurences. The syntax and arguments are the same as for **count**, but where the count method would return 0 if no occurence was found, this method returns an empty array.

Example:

```
alert(JSU.String.indexesOf("ahat", "ha")); // [1]
alert(JSU.String.indexesOf("tesestes", "es")); // [1, 3, 6]
alert(JSU.String.indexesOf("repeater", "e")); // [1, 3, 6]
alert(JSU.String.indexesOf("ciu", "a")); // []
alert(JSU.String.indexesOf("atttatt", "tt")); // [1, 5]
alert(JSU.String.indexesOf("atttatt", "tt", true)); // [1, 2, 5]
```

17. startsWith, endsWith and contains

These three methods are self-explanatory, the first one checking whether the given string starts with a specific substring, the second checking if it ends with that substring and the last if it contains the substring.

Syntax:

```
startsWith(s, sub)
endsWith(s, sub)
contains(s, sub)
```

Arguments:

- ***s***: the string to search within;
- sub: the substring to search for.

Returns: Boolean

Example:

```
alert(JSU.String.startsWith("ahat", "ha")); // false
alert(JSU.String.startsWith("ahat", "ah")); // true
alert(JSU.String.startsWith("\nahat", "\n")); // true
alert(JSU.String.startsWith("ahat", "a")); // true
alert(JSU.String.startsWith("   ", " ")); // true

alert(JSU.String.endsWith("ahat", "ha")); // false
alert(JSU.String.endsWith("ahat", "at")); // true
alert(JSU.String.endsWith("ahat\n", "\n")); // true
alert(JSU.String.endsWith("ahat", "t")); // true
alert(JSU.String.endsWith("   ", " ")); // true

alert(JSU.String.contains("ahat", "ha")); // true
alert(JSU.String.contains("ahat", "at")); // true
alert(JSU.String.contains("ahat\n", "\n")); // true
alert(JSU.String.contains("ahat", "aa")); // false
alert(JSU.String.contains("   ", " ")); // true
```

Developing your own modules

A. Creating a static module

First of all, the JSU framework is designed so that you always have an alias for an object, whether it's static or instantiable. For example, the *JSU.String* object can also be used with *JSUString* without the dot. The first calling convention depicts that JSU is a namespace and String is an object inside that namespace, while the second calling convention eliminates the namespace notation.Whatever notation you choose is up to you, depending on whether you are used to work with namespaces or not.

Below is a template for a static JSU object that contains a property and two methods:

```
(function(JSU) {
    JSUMyObject = JSU.MyObject = {
        property: "value",
        method1: function(arg) {
            alert(arg);
        },
        method2: function() {
            alert(JSU.MyObject.property)
        }
    }
})(JSU);
```

As you can see, you start by defining an anonymous function that calls itself with the JSU core object as its argument, allowing you to use the JSU core object inside your object's methods. The object is defined using the JavaScript literal object notation, so any properties or methods are written between "{ }" and are each separated by a comma. Let's look at an example of how to use this object:

```
JSU.MyObject.method1("test"); // outputs "test"
JSU.MyObject.method2(); // outputs "value"
```

It's pretty straightforward, isn't it ? But remember: you cannot instantiate this object!

B. Creating an instantiable module

For this one you'll need to write more code so that you integrate it with the framework. First I'll show you the template code, then I'll discuss each part of it so you can understand what happens.

```
(function(JSU) {
    function JSU_MyObject(constructor_args) {
        this.property1 = "value1"; this.property2 = " value2 ";
        // other object initializations
    };

    // class methods
    JSU_MyObject.prototype.method1 = function(arg) {
        alert(this.property1);
    };
    JSU_MyObject.prototype.method2 = function() {
        alert(JSU.String.trim(this.property2));
    };

    // static methods
    JSU_MyObject.create = function() {
        return new JSU_MyObject();
    };

    // method aliases
    JSU_MyObject.init = JSU_MyObject.create;
    JSU_MyObject.prototype.method0 = JSU_MyObject.prototype.method1;

    // make it globally available
    JSUMyObject = JSU_MyObject;
    JSU.MyObject = JSU_MyObject;

    // make the JSU framework recognize this object
    JSU.extend({
        isJSUMyObject: function(obj) {
            return (obj instanceof JSU.MyObject);
        },
        areJSUMyObjects: function() {
            if (arguments.length == 0) return false;
            for (var i = 0; i < arguments.length; i++)
                if (!this.isJSUMyObject(arguments[i]))
                    return false;
            return true;
        }
    });
    JSU.addType(JSU.MyObject, "jsumyobject");
})(JSU);
```

As you can see, like the static module you start by defining an anonymous function that calls itself with the JSU core object as its argument allowing you to use it inside the methods. In terms of syntax, the big difference between a static module and an instantiable one is that the static module uses the JavaScript literal object notation, while the last one uses standard OOP notation, with constructors, class and static methods, and so on. Now let's look at each part of this code:

```
function JSU_MyObject(constructor_args) {
    this.property1 = "value1"; this.property2 = " value2 ";
    // other object initializations
};
```

Here we define a temporary class named JSU_MyObject, avoiding the namespace and non-namespace notations so that we don't get ourselves into a conflict. This class is defined as a constructor method in OOP terms, and here is the perfect place to define the object's properties, and even do some initializations. Remember that this method will be called whenever you create a new instance of this object with the *new* keyword.

```
// class methods
JSU_MyObject.prototype.method1 = function(arg) {
    alert(this.property1);
};
JSU_MyObject.prototype.method2 = function() {
    alert(JSU.String.trim(this.property2));
};
```

A class method is a method that can be called only on an instance of the object. You cannot call it without instantiating the class. Above we've defined two class methods that use the object's properties (class methods are defined by accessing the *prototype* object that comes with every new object). Remember to always use the **this** keyword to access the properties and methods of the class inside its methods.

```
// static methods
JSU_MyObject.create = function() {
    return new JSU_MyObject();
};
```

A static method is a method that can be called only on the class and not on instances of it. Typically, static methods return a new instance of the class. In this case I've created a static method that returns a new *JSU_MyObject* object.

```
// method aliases
JSU_MyObject.init = JSU_MyObject.create;
JSU_MyObject.prototype.method0 = JSU_MyObject.prototype.method1;
```

Method aliases are optional. You can use them to allow the developer to use the methods with names that make more sense to him or her. For example, some developers are used to using a **length** method, while others are used with names such as **size** or **capacity**. In our case, I've defined an alias for the *method1* and *init* methods. Notice how you can make an alias for either a static or a class method.

```
// make it globally available
JSUMyObject = JSU_MyObject;
JSU.MyObject = JSU_MyObject;
```

Here I've defined the JSU object for global use (the namespace and non-namespace names).

```
// make the JSU framework recognize this object
JSU.extend({
    isJSUMyObject: function(obj) {
        return (obj instanceof JSU.MyObject);
    },
    areJSUMyObjects: function() {
        if (arguments.length == 0) return false;
        for (var i = 0; i < arguments.length; i++)
            if (!this.isJSUMyObject(arguments[i]))
                return false;
        return true;
    }
});
```

Now it's time to integrate this object with the JSU framework. The code above is optional, allowing you to define methods for use within the JSU core object regarding our custom module. In this case, we've extended the core's functionality with two methods that allow us to check whether one or more objects are of type JSU.MyObject.

```
JSU.addType(JSU.MyObject, "jsumyobject");
```

This code is also optional, but it allows us to extend the JSU's **getType** method so that it recognizes our JSU.MyObject objects. And now, let's look at some examples of how to use this module:

```
var o1 = new JSU.MyObject(),
    o2 = JSU.MyObject.init();

o1.method0(); // "value1"
o2.method2(); // "value2"
alert(JSU.isJSUMyObject(o1)); // true
alert(JSU.getType(o2)); // "jsumyobject"
```

Here, o1 is an JSU.MyObject created with the *new* keyword, while o2 is another JSU.MyObject but created using the class' static method **init**. Then, we call the method0 (alias for method1) on o1, and method2 on o2, the results being those we expected. Next, we check if o1 is of type JSU.MyObject by using the extension method of the JSU core object which we've added with the **extend** method. And finally, we check if o2 is of type "*jsumyobject*" by using the core's **getType** method, testing if our custom type was successfully added with the **addType** method.