ZINCKSOFT

Give The Dream New Life.

JSU.Dictionary

The complete Module API Reference

Quick Introduction

The JSU Dictionary Module (**JSU.Dictionary / JSUDictionary**) provides a new JavaScript data type known as associative arrays which are not supported by default in this language. The Dictionary allows you to store anything you want in it, from native objects to custom objects, and even other dictionaries, either as a key, or as a value, or both. With this modules comes also a new elementary data type used by the dictionary objects - **JSU.Pair**. Each item of a dictionary is a (key, value) pair, so this elementary data type allows you to work easier with dictionary items.

You can also teach a dictionary how to handle your custom objects! And the tip of the iceberg is that this module supports *method chaining*, allowing you write less and do more.

For some awesome code samples, check out the last three chapters from this book where I'm showing ***how the method chaining works***, ***how to sort a dictionary using custom comparing methods***, but also ***how to teach the Dictionary module to work with your custom objects*** - whether they're new JSU objects or custom JavaScript objects.

Including the module in your page

Before including this module in your page, first include the core module:

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

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

JSU.Pair API Reference

A. Constructor

Syntax:

```
JSU.Pair(key [, value])
```

Arguments:

- ***key***: the key used to identify the pair;
- ***value***: the pair's value; defaults to null.

Returns: The JSU.Pair object.

Example:

```
var p1 = new JSU.Pair("value", 2);
var p2 = new JSU.Pair([1, 2, 3], "array1"); // the key is an array
var p3 = new JSU.Pair("key1"); // has a null value
```

B. Properties

1. __key__ / __value__

These are private properties, do not use directly and use the setter/getter methods provided to access them. These properties hold a pair's key and value elements.

2. __comparers__

This is a private property, do not use it directly and use the setter method provided to add new comparer methods. This property holds the comparer methods used for testing normal equality (==), comparing and sorting the pair's elements. For more information on working with comparers, check the last chapter of this book.

3. __exactcomparers__

This is a private property, do not use it directly and use the setter method provided to add new comparer methods. This property holds the comparer methods used for testing strict equality (===), comparing and sorting the pair's elements. For more information on working with comparers, check the last chapter of this book.

C. Static Methods

1. fromArray

This static method will create a new Pair from an existing array. The array must have at least one element, and for more than 2 elements it will take into account only the first two.

Syntax:

```
JSU.Pair.fromArray(arr)
```

Arguments:

- ***arr***: the array used as a data source for the new Pair; an array with only one element will take that element and make it the pair's key, while the value will be null.

Returns: A new JSU.Pair object containing the array's elements.

Example:

```
var l = JSU.Pair.fromArray( ["value", 2] );
```

2. fromString

This static method will create a new Pair from an existing string containing the elements.

Syntax:

```
JSU.Pair.fromString(s [, separator])
```

Arguments:

- ***s***: the string used as a data source for the new Pair;
- ***separator***: the character/substring used within the string to separate the contained elements; defaults to the comma character.

Returns: A new JSU.Pair object containing the string's elements.

Example:

```
var l = JSU.Dictionary.fromString("2,3");
var l = JSU.Dictionary.fromString("value;2", ";");
```

Creating a pair from a string is restrictive because you can only add string elements to the dictionary. In the first line from the above example code, we get a new JSU.Pair containing

only strings for both the key and value, so you cannot add other types (use **fromArray** method to add any type of object to the dictionary).

3. fromPair

This static method will create a new Pair from an existing one.

Syntax:

```
JSU.Pair.fromPaur(pair)
```

Arguments:

- ***pair***: the pair used as a data source for the new Pair.

Returns: A new JSU.Pair object containing the other pair's key and value.

Example:

```
var p = new JSU.Pair("value", 2);
alert(JSU.Pair.fromPair(p));
```

D. General Methods

1. getKey / getValue

Gets the key or value of a pair.

Syntax:

```
getKey()
getValue()
```

Returns: An object.

Example:

```
var p = new JSU.Pair("value", 2);
alert(p.getKey() + ";" + p.getValue()); // value;2
```

2. get

Gets an array containing the key and value of the pair.

Syntax:

```
get()
```

Returns: An array with two elements - the key and the value.

Example:

```
var p = new JSU.Pair("value", 2);
alert(p.get()); // [value,2]
```

3. set (~ setValue)

Sets the value of a pair.

Syntax:

```
set(value)        alias: setValue(value)
```

Arguments:

- ***value***: the new value of the pair.

Returns: A JSU.Pair.

Example:

```
var p = new JSU.Pair("value", 2);
alert(p.set("3")); // {"value", "3"}
```

4. ~~*setKey*~~

This method doesn't exist. It is so because if the JSU.Pair object had this method, you'll be able to set keys within a dictionary without the dictionary being able to check if you duplicate any key. The JSU.Dictionary has its own method for setting keys to avoid this problem, so if you really need to set a pair's key directly, use its **__value__** property, but I don't recommend you to do that, except when working only with JSU.Pair objects and not with JSU.Dictionary.

5. equals / same

These two methods are used to check the equality of two pairs. The **equals** method uses the normal equality operator (==), while the **same** method uses the strict equality operator (===).

Syntax:

```
equals(pair)
same(pair)
```

Arguments:

- ***pair***: the pair to compare to.

Returns: A boolean.

Note: The equals method uses the default comparer methods, while same uses the default exact comparer methods.

Example:

```
var p1 = new JSU.Pair("value", 2),
    p2 = new JSU.Pair("value", "2");
alert(p1.equals(p2)); // true
alert(p1.same(p2)); // false
```

For more information on how to define your own comparer or exact comparer methods to teach the Pair object recognize your custom objects, check the last chapter of this book where I show you how to do this with dictionaries and then you can adapt that example for the pair object.

6. toArray / toString

These two methods convert the Pair object to a native JavaScript array/string.

Syntax:

```
toArray()
toString([separator])
```

Arguments:

- ***separator***: the character/substring used to separate the key from the value within the resulting string; defaults to the comma character.

Returns: An array or a string.

Example:

```
var p1 = new JSU.Pair("value", 2),
    p2 = new JSU.Pair("value", "2");
alert(p1.toArray()); // [value,2]
alert(p1.toString(";")); // "value;2"
```

E. Comparer Methods

1. resetComparers / resetExactComparers

These methods are used to reset the comparer methods, usually if the developer added one or more custom comparers (thus disabling all custom comparers).

Syntax:

```
resetComparers()
resetExactComparers()
```

Returns: Nothing.

Example:

```
p1 = new JSU.Pair();
p1.resetComparers(); // default comparers will be used
p1.resetExactComparers(); // default comparers will be used
```

2. addComparer / addExactComparer

These methods allow you to add your custom comparer methods so that you teach the JSU.Pair object how to handle custom objects. Notice that you can add comparer methods only to instances of the JSU.Pair object, allowing you this way to have dictionaries with different comparer methods depending on what you store in them.

Syntax:

```
addComparer(comp)
addExactComparer(comp)
```

Comparer Syntax:

```
function(a, b)
```

Predicate Arguments:

- ***a, b***: the objects that will be checked for equality at any time of the iteration.

Predicate must return: A boolean indicating whether the current objects are equal or not.

Returns: Nothing.

Note: The comparers must always return a boolean, otherwise your could hang the browser in an infinite loop. Also, be careful to use strict equality (===) for the exact comparers, and normal equality (==) for the standard comparers, or you'll get unexpected results.

Example:

```
p1 = new JSU.Pair();
var comp = function(a, b) {
    return a == b;
};
p1.addComparer(comp);
```

JSU.Dictionary API Reference

A. Constructor

Syntax:

```
JSU.Dictionary()
JSU.Dictionary(dict)
JSU.Dictionary(obj1 [, obj2 [, obj3 ... ]] [, replace])
```

Arguments:

- ***obj1-N***: either arrays with at least one element (the pair's key), or a JSU.Pair objects (can be combined);
- ***dict***: a dictionary to use as a copy;
- ***replace***: a boolean indicating whether any new pair that has a duplicate key will replace the existing key's value with the new value.

Returns: The JSU.Dictionary object.

Example:

```
var d1 = new JSU.Dictionary(["a", 1], ["b", 2]);
    // d1 = { {"a", 1}, {"b", 2} }
var d2 = new JSU.Dictionary(d1);
    // d2 = { {"a", 1}, {"b", 2} }
var d3 = new JSU.Dictionary(
    new JSU.Pair([1], 1), new JSU.Pair([2], 2));
    // d3 = { {[1], 1}, {[2], 2} }
var d4 = new JSU.Dictionary(["a", 1], ["b", 2], ["a", 3], true);
    // d4 = { {"a", 3}, {"b", 2} }, the last pair replaced the first
```

B. Properties

1. __list__

This is a private property, do not use it directly and use the getter/setter methods provided. This property holds the dictionary's internal structure (an array of pairs).

2. __comparers__

This is a private property, do not use it directly and use the setter method provided to add new comparer methods. This property holds the comparer methods used for testing normal equality (==), comparing and sorting the dictionary's elements.

3. __exactcomparers__

This is a private property, do not use it directly and use the setter method provided to add new comparer methods. This property holds the comparer methods used for testing strict equality (===), comparing and sorting the dictionary's elements.

C. Static Methods

1. fromArray

This static method will create a new Dictionary from an existing array containing smaller arrays (at least one element) with the key/value pairs.

Syntax:

```
JSU.Dictionary.fromArray(arr)
```

Arguments:

- ***arr***: the array used as a data source for the new Dictionary.

Returns: A new JSU.Dictionary object containing the array's pair arrays.

Example:

```
var d1 = JSU.Dictionary.fromArray([ ["a", 1], ["b", 2] ]);
```

2. fromPairsArray

This static method will create a new Dictionary from an existing array containing JSU.Pair objects.

Syntax:

```
JSU.Dictionary.fromPairsArray(arr)
```

Arguments:

- ***arr***: the array used as a data source for the new Dictionary.

Returns: A new JSU.Dictionary object containing the array's pairs.

Example:

```
var d1 = JSU.Dictionary.fromPairsArray(
    [new JSU.Pair("a", 1), new JSU.Pair("b", 3)]
);
```

D. Manipulation Methods

1. clear (~ *empty*)

This method will empty a dictionary.

Syntax:

```
clear()      alias: empty()
```

Returns: The empty JSU.Dictionary.

Example:

```
var d1 = new JSU.Dictionary(["a", 1], ["b", 2]);
d1.clear(); // the dictionary is now empty
```

2. initWith

Creates a new Dictionary containing a specified number of the same object, the keys being generated automatically.

Syntax:

```
initWith(val, size [, keyPrefix])
```

Arguments:

- ***val***: the object used to fill the dictionary's values;
- ***size***: the size of the new dictionary; must be at least 1.
- ***keyPrefix***: the string used to prefix the generated keys (the method will generate string keys containing the index of the key within the dictionary); defaults to an empty string.

Returns: A new JSU.Dictionary.

Example:

```
var d1 = new JSU.Dictionary().initWith(0, 3);
    // { {"0", 0}, {"1", 0}, {"2", 0} }
d1.initWith(0, 3, "key ");
    // { {"key 0", 0}, {"key 1", 0}, {"key 2", 0} }
```

3. initWithArray

Creates a new Dictionary containing specific values, the keys being generated automatically. The size of the new dictionary depends on the size of the specified array of values.

Syntax:

```
initWithArray(valuesArr [, keyPrefix])
```

Arguments:

- ***valuesArr***: an array with the values of each pair;
- ***keyPrefix***: the string used to prefix the generated keys (the method will generate string keys containing the index of the key within the dictionary); defaults to an empty string.

Returns: A new JSU.Dictionary.

Example:

```
var d1 = new JSU.Dictionary().initWithArray( [1, 5, "test", []] );
    // { {"0", 1}, {"1", 5}, {"2", "test"}, {"3", [] } }
d1.initWithArray([1, 5, "test"], "key ");
    // { {"key 0", 1}, {"key 1", 5}, {"key 2", "test"} }
```

4. push

Adds a new pair at the end of the dictionary.

Syntax:

```
push(obj)
```

Arguments:

- ***obj***: either an array with at least one element (the pair's key), or a JSU.Pair object.

Returns: A JSU.Dictionary.

Example:

```
d1.push(["value", 2]);
d1.push(new JSU.Pair("b", 3));
// d1 = { {"value", 2}, {"b", 3} }
```

5. pop

Gets the last pair from the dictionary and then removes it.

Syntax:

```
pop()
```

Returns: A JSU.Pair.

Example:

```
d1.push(["value", 2]);
d1.push(new JSU.Pair("b", 3));
// d1 = { {"value", 2}, {"b", 3} }

alert(d1.pop());
// outputs {"b", 3}, d1 = { {"value", 2} }
```

6. shift

Similar to **pop**, only it gets the first pair then removes it from the dictionary.

Syntax:

```
shift()
```

Returns: A JSU.Pair.

Example:

```
d1.push(["value", 2]);
d1.push(new JSU.Pair("b", 3));
// d1 = { {"value", 2}, {"b", 3} }

alert(d1.shift());
// outputs {"value", 2}, d1 = { {"b", 3} }
```

7. unshift

Similar to **push**, only it inserts a new pair at the beginning of the dictionary.

Syntax:

```
unshift(obj)
```

Arguments:

- ***obj***: either an array with at least one element (the pair's key), or a JSU.Pair object.

Returns: A JSU.Dictionary.

Example:

```
d1.push(["value", 2]);
d1.push(new JSU.Pair("b", 3));
// d1 = { {"value", 2}, {"b", 3} }

d1.unshift(["c", 0]);
// d1 = { {"c", 0}, {"value", 2}, {"b", 3} }
```

8. add

Adds new pairs to a dictionary, similar to the JSU.Dictionary's constructor method, except that id doesn't allow adding new dictionaries like the JSU.List and JSU.Set do (use **addFromDictionary** / **addFromDictionaries** mehods instead).

Syntax:

```
add(obj1 [, obj2 [, obj3 ...]])
```

Arguments:

- ***obj1-N***: either arrays with at least one element (the pair's key), or a JSU.Pair objects (can be combined).

Returns: A JSU.Dictionary.

Example:

```
d1.add(["value", 2], new JSU.Pair("b", 3));
// d1 = { {"value", 2}, {"b", 3} }
```

9. addFromDictionary / addFromDictionaries (~ *concat*)

Adds content from other JSU.Dictionary objects. The first method allows you to add contents from one dictionary, while the last one lets you add content from more dictionaries.

Syntax:

```
addFromDictionary(dict [, replace])
addFromDictionaries(dict1 [, dict2 [, dict3 ...]] [, replace])
        alias: concat(dict1 [, dict2 [, dict3 ...]] [,replace])
```

Arguments:

- ***dict, dict1-N***: the dictionaries from which the content will be added to the calling dictionary;
- ***replace***: a boolean indicating whether any new pair that has a duplicate key will replace the existing key's value with the new value.

Returns: A JSU.Dictionary.

Example:

```
var d1 = new JSU.Dictionary(["a", 1]);
var d2 = new JSU.Dictionary(["b", 1], ["a", 0]);
var d3 = new JSU.Dictionary(new JSU.Pair("c", 2));
d1.addFromDictionaries(d2, d3, false);
    // d1 = { {"a", 1}, {"b", 1}, {"c", 2} }
d1.addFromDictionaries(d2, d3, true);
    // d1 = { {"a", 0}, {"b", 1}, {"c", 2} }
```

10. addFromArray / addFromPairsArray

Adds content from an array of smaller arrays containing key/value elements, or an array containing JSU.Pair objects. These are similar to the static methods **fromArray** / **fromPairsArray**, but there are some differences - these two work only on instances of JSU.Dictionary (compared to the static methods), and these have an extra argument for managing duplicate keys.

Syntax:

```
addFromArray(arr [, replace])
addFromPairsArray(arr [, replace])
```

Arguments:

- ***arr***: the array used as a data source for the dictionary;
- ***replace***: a boolean indicating whether any new pair that has a duplicate key will replace the existing key's value with the new value.

Returns: A JSU.Dictionary.

Example:

```
d1.addFromArray([["a", 1], ["b", 1]]);
    // d1 = { {"a", 1}, {"b", 1} }
d1.addFromPairsArray([new JSU.Pair("a", 0)], true);
    // d1 = { {"a", 0}, {"b", 1} }
```

11. set

Sets the value of an existing pair identified by a specified key. Notice that if the key doesn't exist, a new pair will be created.

Syntax:

```
set(key, value)
```

Arguments:

- ***key***: the key of the pair who's value needs to be changed;
- ***value***: the new value for the pair.

Returns: A JSU.Dictionary.

Example:

```
var d1 = new JSU.Dictionary(["a", 1], ["b", 2]);
d1.set("a", 0);
d1.set("c", 1);
// d1 = { {"a", 0}, {"b", 2}, {"c", 1} }
```

12. setAt

Sets the value of a pair at the specified position within the dictionary.

Syntax:

```
setAt(index, value)
```

Arguments:

- ***index***: the position of the pair who's value will be changed;
- ***value***: the new value for the pair.

Returns: A JSU.Dictionary.

Example:

```
var d1 = new JSU.Dictionary(["a", 1], ["b", 2]);
d1.setAt(0, 0);
// d1 = { {"a", 0}, {"b", 2} }
```

13. setKey / setValue

Sets the key or the value of an existing pair. Notice that **setValue** will not a new pair if the key doesn't exist (use **set** instead), and the **setKey** method will not work if the new key already exists.

Syntax:

```
setKey(oldKey, newKey)
setValue(key, newValue)
```

Arguments:

- ***oldKey***: the existing key to change;
- ***newKey***: the new key for the pair;
- ***key***: the key for the pair who's value needs to be changed;
- ***newValue***: the new value for the pair.

Returns: A JSU.Dictionary.

Example:

```
var d1 = new JSU.Dictionary(["a", 1], ["b", 2]);
d1.setKey("b", "c");
// d1 = { {"a", 1}, {"c", 2} }
d1.setValue("c", 3);
// d1 = { {"a", 1}, {"c", 3} }
```

14. reverse

Reverses the Dictionary's pairs.

Syntax:

```
reverse()
```

Returns: A JSU.Dictionary.

Example:

```
var d1 = new JSU.Dictionary(["a", 1], ["b", 2]);
d1.reverse(); // d1 = { {"b", 2}, {"a", 1} }
```

15. insert

Inserts a new pair at the beginning of the dictionary. This is similar to the **add** method, the syntax and arguments being the same. For more information, check the **add** method from this chapter.

Example:

```
var d1 = new JSU.Dictionary(["a", 1], ["b", 2]);
d1.insert(["c", 3], new JSU.Pair("d", 4));
// d1 = { {"c", 3}, {"d", 4}, {"a", 1}, {"b", 2} }
d1.insert(["d", 0], true);
// d1 = { {"c", 3}, {"d", 0}, {"a", 1}, {"b", 2} }
```

16. insertFromArray / insertFromPairsArray

Adds content at the beginning of the dictionary from an array of key/value arrays, or from a pairs array containing the new elements. These are similar to the **addFromArray** / **addFromPairsArray** methods, the only differences being that adding is done at the beginning of the dictionary. As a note, inserting a pair that contains a duplicate key will change the existing pair's value (the pair that has the same key as the one to be inserted), and will not move it at the beginning of the dictionary.

The syntax and arguments are the same, and return the modified calling dictionary, so check those two methods above mentioned for more information.

Example:

```
var d1 = new JSU.Dictionary(["a", 1], ["b", 2]);
d1.insertFromArray([["c", 3], ["a", -1]]);
// d1 = { {"c", 3}, {"a", 1}, {"b", 2} }
d1.insertFromPairsArray(
    [new JSU.Pair("a", 0), new JSU.Pair("d", 4)], true);
// d1 = { {"d", 4}, {"c", 3}, {"a", 0}, {"b", 2} }
```

17. insertFromDictionary / insertFromDictionaries (~ *inverseConcat*)

Inserts content at the beginning of the calling dictionary from other JSU.Dictionary objects. These two are identical with the **addFromDictionary** / **addFromDictionaries** methods, the only difference being that adding is done at the beginning of the dictionary. The syntax and arguments are the same, so check the other two methods mentioned above for more information.

Example:

```
var d1 = new JSU.Dictionary(["a", 1], ["b", 2], ["c", 3]);
var d2 = new JSU.Dictionary(["b", 0]);
var d3 = new JSU.Dictionary(["e", -1], ["f", -2]);
var d4 = new JSU.Dictionary(["g", -3]);
d1.insertFromDictionary(d2, true);
// d1 = { {"a", 1}, {"b", 0}, {"c", 3} }
d1.insertFromDictionaries(d3, d4);
// d1 = { {"e", -1}, {"f", -2}, {"g", -3}, {"a", 1}, {"b", 0}, {"c", 3} }
```

18. insertAt

Inserts one or more pairs at a specified location within the dictionary.

Syntax:

```
insertAt(index, obj1 [, obj2, [ obj3 ... ]] [, replace])
```

Arguments:

- ***index***: the position where the insertion will take place;
- ***obj1-N***: the pairs to insert (either arrays with key/value elements, or JSU.Pair objects);
- ***replace***: a boolean indicating whether any new pair that has a duplicate key will replace the existing key's value with the new value.

Returns: A JSU.Dictionary.

Example:

```
var d1 = new JSU.Dictionary(["a", 1], ["b", 2], ["c", 3]);

d1.insertAt(1, ["x", 0], new JSU.Pair("a", -1), ["b", -1]);
// d1 = { {"a", 1}, {"x", 0}, {"b", 2}, {"c", 3} }
d1.insertAt(1, ["x", 0], new JSU.Pair("a", -1), ["b", -1], true);
// d1 = { {"a", -1}, {"x", 0}, {"b", -1}, {"c", 3} }
```

19. remove

Removes one or more pairs from the dictionary.

Syntax:

```
remove(obj1 [, obj2 [, obj3 ... ]])
```

Arguments:

- ***obj1-N***: the pairs to be removed (either arrays with key/value elements, or JSU.Pair objects).

Returns: A JSU.Dictionary.

Example:

```
var d1 = new JSU.Dictionary(["a", 1], ["b", 2], ["c", 3]);
d1.remove(["b", 2], new JSU.Pair("a", 1));
// d1 = { {"c", 3} }
```

20. removeAt

Removes the pair at a specified position within the dictionary.

Syntax:

```
removeAt(index)
```

Arguments:

- ***index***: the position where the removal takes place.

Returns: A JSU.Dictionary.

Example:

```
var d1 = new JSU.Dictionary(["a", 1], ["b", 2], ["c", 3]);
d1.removeAt(1);
// d1 = { {"a", 1}, {"c", 3} }
```

21. removeRange

Removes a range of pairs from the dictionary.

Syntax:

```
removeRange(start, count)
```

Arguments:

- ***start***: the position from where to remove;
- ***count***: the number of objects to remove.

Returns: A JSU.Dictionary.

Example:

```
var d1 = new JSU.Dictionary(["a", 1], ["b", 2], ["c", 3]);
d1.removeRange(2,1);
// d1 = { {"a", 1}, {"b", 2} }
```

22. removeKey (~ *removeByKey*) / removeKeys (~ *removeByKeys*)

Removes one or more pairs identified by the specified key(s).

Syntax:

```
removeKey(key)          alias: removeByKey(key)
removeKeys(key1 [, key2 ...])   alias: removeByKeys(key1 [, key2 ...])
```

Arguments:

- ***key, key1-N***: the keys identifying the pairs to remove;

Returns: A JSU.Dictionary.

Example:

```
var d1 = new JSU.Dictionary(["a", 1], ["b", 2], ["c", 3]);
d1.removeKeys("a", "c");
// d1 = { {"b", 2} }
```

23. removeValue (~ *removeByValue*) / removeValues (~ *removeByValues*)

Removes one or more pairs identified by the specified value(s). Notice that if the dictionary contains multiple pairs having the same value(s), all those pairs will be removed.

Syntax:

```
removeValue(val)      alias: removeByValue(val)
removeKeys(val1 [,val2 ..]])   alias: removeByValues(val1 [,val2 ..])
```

Arguments:

- ***val, val1-N***: the values identifying the pairs to remove;

Returns: A JSU.Dictionary.

Example:

```
var d1 = new JSU.Dictionary(["a", 1], ["b", 2], ["c", 3]);
d1.removeValues(1, 3);
// d1 = { {"b", 2} }
```

24. sortByKey / sortByValue

Sorts the dictionary by key or by value using the QuickSort method for best performance. You can sort a dictionary using your own method for comparing the dictionary's key or value objects, or you can use the default comparing method (but can lead to unwanted results when using custom objects).

Syntax:

```
sortByKey([comparer [, dir]])
sortByKey([dir [, comparer]])

sortByValue([comparer [, dir]])
sortByValue([dir [, comparer]])
```

Arguments:

- ***dir***: the direction of sorting; can be either "*ASC*" or "*DESC*" (case insensitive); defaults to "ASC";
- ***comparer***: the method used for comparing two objects from the dictionary; the default method compares objects directly using the ">" and "<" operators.

Comparer Syntax:

```
function(a, b [, dir])
```

Comparer Arguments:

- ***a, b***: the two objects to be compared; the comparer is called for each two objects from the dictionary, the order of comparing depending on the sorting algorithm;
- dir: the direction of sorting that is usually passed to the sort method; defaults to "ASC".

Note: Typically, the comparer method must return -1 for a < b, 1 for a > b and 0 for a = b.

Returns: A sorted JSU.Dictionary.

Example:

```
var d1 = new JSU.Dictionary(["c", 1], ["a", 2], ["b", 3]);
d1.sortByKey(); // d1 = { {"a", 2}, {"b", 3}, {"c", 1} }
d1.sortByKey("DESC"); // d1 = { {"c", 1}, {"b", 3}, {"a", 2} }
d1.sortByValue(); // d1 = { {"c", 1}, {"a", 2}, {"b", 3} }
d1.sortByValue("DESC"); // d1 = { {"b", 3}, {"a", 2}, {"c", 1} }
```

For an example of how to sort a dictionary using a custom comparer method, check the last two chapters of this book.

E. Information Methods

1. length (~ *size*)

Gets the size of the dictionary (the number of pairs it has).

Syntax:

```
length()     alias: size()
```

Returns: A number.

Example:

```
var d1 = new JSU.Dictionary(["c", 1], ["a", 2], ["b", 3]);
alert(d1.size()); // outputs 3
```

2. get (~ *getPair, getAt, getPairAt, items*)

Gets the pair at a specified position within the dictionary.

Syntax:

```
get(index)        alias: items(index), getPair(index),
                         getPairAt(index), getAt(index)
```

Arguments:

- ***index***: the position from which to retrieve the pair.

Returns: A JSU.Pair.

Example:

```
var d1 = new JSU.Dictionary(["c", 1], ["a", 2], ["b", 3]);
alert(d1.get(1)); // outputs {"a", 2}
```

3. item

Gets or sets the value of a pair found at a specified position or identified by a specified key.

Syntax:

```
item(index [, value])
item(key [, value])
```

Arguments:

- ***index***: the position from which to retrieve the pair, or at which to set the new value;
- ***value***: the new value for the pair at the specified index or identified by the specified key; ignoring this value will make the method retrieve the value at the specified key/index.
- ***key***: the key of the pair who's value needs to be set or retrieved; beware that if you have keys of type Number, the method will think you passed an index and not a key, so the key parameter must be everything but a number.

Returns: If both arguments are passed, it returns the modified dictionary object; if only the index is passed, it returns the object at that location.

Example:

```
var d1 = new JSU.Dictionary(["c", 1], ["a", 2], ["b", 3]);
alert(d1.item(1)); // outputs {"a", 2}
alert(d1.item("a")); // outputs {"a", 2}
alert(d1.item(1, 0)); // d1 = { {"c", 1}, {"a", 0}, {"b", 3} }
alert(d1.item("a", -1)); // d1 = { {"c", 1}, {"a", -1}, {"b", 3} }
```

4. containsKey (~ *hasKey*) / containsValue (~ *hasValue*) / containsPair (~ *hasPair*)

Checks whether the dictionary contains a specified pair, or a pair who's key or value matches the specified argument.

Syntax:

```
containsKey(obj)          alias: hasKey(obj)
containsValue(obj)        alias: hasValue(obj)
containsPair(obj)         alias: hasPair(obj)
```

Arguments:

- ***obj***: the key/value/JSU.Pair object to search for.

Returns: A boolean.

Note: This method uses the default exact comparers.

Example:

```
var d1 = new JSU.Dictionary(["c", 1], ["a", 2], ["b", 3]);
alert(d1.hasKey("a")); // true
alert(d1.hasKey("x")); // false
alert(d1.hasValue(1)); // true
alert(d1.hasValue(0)); // false
alert(d1.hasPair(new JSU.Pair("c", 1))); // true
alert(d1.hasPair(new JSU.Pair("c", 3))); // false
```

5. equals / same

These two methods are used to check the equality of the dictionary with another specified dictionary. The **equals** method uses the normal equality operator (==), while the **same** method uses the strict equality operator (===).

Syntax:

```
equals(dict)
same(dict)
```

Arguments:

- ***dict***: the dictionary to compare to.

Returns: A boolean.

Note: The equals method uses the default comparer methods, while same uses the default exact comparer methods.

Example:

```
var d1 = new JSU.Dictionary(["c", 1], ["a", 2], ["b", 3]);
var d2 = new JSU.Dictionary(["c", "1"], ["a", 2], ["b", 3]);
alert(d1.equals(d2)); // true
alert(d1.same(d2)); // false
```

For more information on how to define your own comparer or exact comparer methods to teach the Dictionary object recognize your custom objects, check the last chapter of this book.

6. getTypeOfKeyAt (~ *getTypeOfKey*) / getTypeOfValueAt (~ *getTypeOfValue*)

These method get the type of a pair's key or value found at a specified position within the dictionary. The method uses the JSU core's **getType** method, so any new module added to the framework will automatically be recognized (but only if the module's developer made sure to add the global type for his/her module by using the **addType** method of the JSU core).

Syntax:

```
getTypeOfKeyAt(index)              alias: getTypeOfKey(index)
getTypeOfValueAt(index)            alias: getTypeOfValue(index)
```

Arguments:

- ***index***: the index of the pair.

Returns: A string containing the type's name (typically a lowercase string if module developers haven't altered the JSU naming conventions).

Example:

```
var d1 = new JSU.Dictionary(["c", 1], ["a", 2], ["b", 3]);
alert(d1.getTypeOfKeyAt(1)); // string
alert(d1.getTypeOfValueAt(2)); // number
```

7. indexOfKey / indexOfValue / indexOfPair

Gets the index of a specified key, value or pair object.

Syntax:

```
indexOfKey(obj)
indexOfValue(obj)
indexOfPair(obj)
```

Arguments:

- ***obj***: the object to search for, depending on the method it could be a key, a value or a JSU.Pair.

Returns: A number.

Note: all three methods use the default exact comparer methods.

Example:

```
var d1 = new JSU.Dictionary(["c", 1], ["a", 2], ["b", 3]);
alert(d1.indexOfKey("a")); // 1
alert(d1.indexOfKey("x")); // -1
alert(d1.indexOfValue(3)); // 2
alert(d1.indexOfValue(-1)); // -1
alert(d1.indexOfPair(new JSU.Pair("b", 3))); // 2
```

8. indexesOfValue (~ *indexesOf*)

Gets all indexes of a specified value.

Syntax:

```
indexesOfValue(val)      alias: indexesOf(val)
```

Arguments:

- ***val***: the value to search for within the dictionary.

Returns: An array with all the indexes, or an empty array if the value is not found.

Note: This method uses the default exact comparer methods.

Example:

```
var d1 = new JSU.Dictionary(["c", 1], ["a", 2], ["b", 1]);
alert(d1.indexesOf(1)); // [0, 2]
alert(d1.indexesOf(3)); // []
```

9. lastIndexOfValue (~ *lastIndexOf*)

Gets the index of the last pair containing a specified value.

Syntax:

```
lastIndexOfValue(val)     alias: lastIndexOf(val)
```

Arguments:

- ***val***: the value to search for within the dictionary.

Returns: A number.

Note: This method uses the default exact comparer methods.

Example:

```
var d1 = new JSU.Dictionary(["c", 1], ["a", 2], ["b", 1]);
alert(d1.lastIndexOf(1)); // 2
```

10. count

Counts how many times a specified value occurs within the dictionary.

Syntax:

```
count(value)
count(predicate)
```

Arguments:

- ***value***: the value to search for;
- ***predicate***: the method used to search positive matches.

Predicate Syntax:

```
function(index, key, val)
```

Predicate Arguments:

- ***index***: the index of the pair at which the searching takes place;
- ***key***: the key of the pair at which the searching has arrived;
- ***value***: the value of the pair at which the searching has arrived;

Note: The predicate must always return a boolean value indicating whether the current position represents a positive match.

Returns: A number.

Note: this method uses the default exact comparer methods.

Example:

```
var d1 = new JSU.Dictionary(["c", 1], ["a", 2], ["b", 1]);
alert(d1.count(1)); // 2

var match = function(index, key, val) {
    // any pair found at an even index
    // that has a value of 1
    return (index + 1) % 2 == 0 && val == 1;
};
alert(d1.count(match)); // 0 (no 1's are on even indexes)
```

11. first / last

Gets the first/last object from the dictionary, or the first/last *n* objects.

Syntax:

```
first([n])
last([n])
```

Arguments:

- ***n***: the number of pairs to retrieve.

Returns: If no argument is passed, the methods return the first/last JSU.Pair object from the dictionary, otherwise it returns a subdictionary containing the first/last *n* pairs from the dictionary.

Example:

```
var d1 = new JSU.Dictionary(["c", 1], ["a", 2], ["b", 1]);
alert(d1.first()); // {"c", 1}
alert(d1.last()); // {"b", 1}
alert(d1.first(2)); // { {"c", 1}, {"a", 2} }
alert(d1.last(2)); // { {"a", 2}, {"b", 1} }
```

12. isEmpty

Checks if the dictionary is empty.

Syntax:

```
isEmpty()
```

Returns: A boolean.

Example:

```
var d1 = new JSU.Dictionary(["c", 1], ["a", 2], ["b", 1]);
alert(d1.isEmpty()); // false
alert(new JSU.Dictionary().isEmpty()); // true
```

13. isTyped

Checks if all the pairs have the same type for the keys, and the same type for the values (the type of the key may differ from the type of the value).

Syntax:

```
isTyped()
```

Returns: A boolean.

Example:

```
var d1 = new JSU.Dictionary(["c", 1], ["a", 2], ["b", 1]);
alert(d1.isTyped()); // true
alert(new JSU.Dictionary(["a", 1], ["b", "test"]).isTyped()); // false
alert(new JSU.Dictionary().isTyped()); // false
```

14. keysOf

Gets all the keys for the pairs that have a specified value.

Syntax:

```
keysOf(value)
```

Arguments:

- ***value***: the value to search for.

Returns: An array containing the keys.

Example:

```
var d1 = new JSU.Dictionary(["c", 1], ["a", 2], ["b", 1]);
alert(d1.keysOf(1)); // ["c", "b"]
```

15. getKey (~ *getFirstKey*) / tryGetKey (~ *tryGetFirstKey*)

Gets the key for the first pair that has a specified value. The **tryGetKey** method will try to get the key, and if it doesn't succeed, it will return the default specified value for this case.

Syntax:

```
getKey(val)                    alias: getFirstKey(val)
tryGetKey(val, defKey)         alias: tryGetFirstKey(val, defKey)
```

Arguments:

- ***val***: the value to search for;
- ***defKey***: the default key value to return in case the value doesn't exist.

Returns: An object.

Example:

```
var d1 = new JSU.Dictionary(["c", 1], ["a", 2], ["b", 1]);
alert(d1.getKey(1)); // "c"
alert(d1.tryGetKey(5, "x")); // "x"
```

16. getKeyAt / getValueAt

Gets the key or value of the pair found at a specified position within the dictionary.

Syntax:

```
getKeyAt(index)
getValueAt(index)
```

Arguments:

- ***index***: the index for the pair who's key or value to retrieve.

Returns: An object.

Example:

```
var d1 = new JSU.Dictionary(["c", 1], ["a", 2], ["b", 1]);
alert(d1.getKeyAt(1)); // "a"
alert(d1.getValueAt(1)); // "2"
```

17. getLastKey / tryGetLastKey

Gets the key for the last pair that has a specified value. The **tryGetLastKey** method will try to get the key, and if it doesn't succeed, it will return the default specified value for this case.

Syntax:

```
getLastKey(val)
tryGetLastKey(val, defKey)
```

Arguments:

- ***val***: the value to search for;
- ***defKey***: the default key value to return in case the value doesn't exist.

Returns: An object.

Example:

```
var d1 = new JSU.Dictionary(["c", 1], ["a", 2], ["b", 1]);
alert(d1.getLastKey(1)); // "b"
alert(d1.tryGetLastKey(3, "x")); // "x"
```

18. getKeys / getValues

Gets all keys or values from the dictionary.

Syntax:

```
getKeys()
getValues()
```

Returns: An array.

Example:

```
var d1 = new JSU.Dictionary(["c", 1], ["a", 2], ["b", 1]);
alert(d1.getKeys()); // ["c", "a", "b"]
alert(d1.getValues()); // [1, 2, 1]
```

19. getValue / tryGetValue

Gets the value of a pair identified by a specified key. The **tryGetValue** will try to get the value, but if the specified key doesn't exist, it will return the default value for this case.

Syntax:

```
getValue(key)
tryGetValue(key, defValue)
```

Arguments:

- ***key***: the key to search for;
- ***defValue***: the default value to return in case of a non existing key.

Returns: An object.

Example:

```
var d1 = new JSU.Dictionary(["c", 1], ["a", 2], ["b", 1]);
alert(d1.getValue("a")); // 2
alert(d1.tryGetValue("x", -1)); // -1
```

20. getPairForKey (~ *getForKey*)

Gets the pair identified by a specified key.

Syntax:

```
getPairForKey(key)    alias: getForKey(key)
```

Arguments:

- ***key***: the key to search for.

Returns: A JSU.Pair, or null if the key doesn't exist.

Example:

```
var d1 = new JSU.Dictionary(["c", 1], ["a", 2], ["b", 1]);
alert(d1.getForKey("a")); // {"a", 2}
alert(d1.getForKey("x")); // null
```

F. Extraction Methods

1. grepKeys / grepValues

Extracts a subdictionary with only the pairs who's keys or values match a specific regular expression.

Syntax:

```
grepKeys(regex)          alias: grepKey(regex)
grepValues(regex)            alias: grepValue(regex)
```

Arguments:

- ***regex***: the JavaScript regular expression used to extract pairs.

Returns: A new JSU.Dictionary.

Example:

```
var d1 = new JSU.Dictionary(["c", 1], ["a", 2], ["b", 1]);
alert(d1.grepKeys(/^[ab]$/)); // { {"a", 2}, {"b", 1} }
alert(d1.grepValues(/^[2-9]$/)); // { {"a", 2} }
```

2. getRange

Extracts a subdictionary from a specified range within the calling dictionary.

Syntax:

```
getRange(start, count)
```

Arguments:

- ***start***: the position from where to extract;

- ***count***: the number of pairs to extract.

Returns: A new JSU.Dictionary.

Example:

```
var d1 = new JSU.Dictionary(["c", 1], ["a", 2], ["b", 1]);
alert(d1.getRange(2, 1)); // { {"b", 1} }
```

3. getSubDictionary

Extracts a subdictionary from the calling dictionary starting at a specified position up to the end of the dictionary or to a specified end position. Similar to **getRange**, but the second parameter is not the number of pairs to extract, but the end position at which the extraction ends.

Syntax:

```
getSubDictionary(start [, end])
```

Arguments:

- ***start***: the position from where to start extracting;
- ***end***: the position at which the extraction will stop (the pair at that position will be excluded); defaults to the end of the dictionary.

Returns: A new JSU.Dictionary.

Example:

```
var d1 = new JSU.Dictionary(["c", 1], ["a", 2], ["b", 1]);
alert(d1.getSubDictionary(1, 2)); // { {"a", 2} }
alert(d1.getSubDictionary(2)); // { {"b", 1} }
```

4. distinct (~ *removeDuplicates*)

Extracts a subdictionary containing only the pairs with distinct values.

Syntax:

```
distinct()      alias: removeDuplicates()
```

Returns: A new JSU.Dictionary.

Note: This method uses the default exact comparer methods.

Example:

```
var d1 = new JSU.Dictionary(["c", 1], ["a", 2], ["b", 1]);
alert(d1.distinct()); // { {"c", 1}, {"a", 2} }
```

5. skip / inverseSkip

The skipping methods allow you to extract a subdictionary from the calling dictionary by skipping a specified number of pairs from it. The **inverseSkip** method skips from the end of the dictionary, and extracts the subdictionary in reverse.

Syntax:

```
skip(count)
inverseSkip(count)
```

Arguments:

- ***count***: the number of pairs to skip on extraction.

Returns: A new JSU.Dictionary.

Example:

```
var d1 = new JSU.Dictionary(["c", 1], ["a", 2], ["b", 1]);
alert(d1.skip(1)); // { {"a", 2}, {"b", 1} }
alert(d1.inverseSkip(1)); // { {"a", 2}, {"c", 1} }
```

If you want to skip from the end and return a subdictionary that's not reversed, you have more options of doing this, but here's the main two methods you could use:

```
alert(d1.inverseSkip(1).reverse()); // { {"c", 1}, {"a", 2} }

alert(d1.take(d1.length() - 1)); // { {"c", 1}, {"a", 2} }
```

I consider the first one being the most obvious method of doing this, but there are more methods to do this because of the multitude of methods the JSU.Dictionary object has (it was designed to work like this so you can do everything you want in various ways, depending on how you know or want to do things).

6. take / inverseTake

These methods are the opposite of the **skip** / **inverseSkip** methods. While the last two skip a number of pairs before the extraction begins, the taking methods work by first extracting and then skipping. The **take** method is similar to **first** method, except that it must have an argument. The **inverseTake** method does not work like the **last** method, because the **last** method returns the last *n* pairs from the dictionary in the order they appear, while the **inverseTake** method extracts the last *n* pairs in reverse order.

Syntax:

```
take(count)
inverseTake(count)
```

Arguments:

- ***count***: the number of pairs to take from the dictionary.

Returns: A new JSU.Dictionary.

Example:

```
var d1 = new JSU.Dictionary(["c", 1], ["a", 2], ["b", 1]);
alert(d1.take(2)); // { {"c", 1}, {"a", 2} }
alert(d1.inverseTake(2)); // { {"b", 1}, {"a", 2} }

alert(d1.last(2)); // { {"a", 2}, {"b", 1} }
```

7. toArray / toPairsArray / toString

These three methods convert the Dictionary object to a native JavaScript array/string. The **toPairsArray** converts it to an array of JSU.Pair objects.

Syntax:

```
toArray()
toPairsArray()

toString([separator])
toString([prefix, suffix])
toString([separator, prefix, suffix])
toString([separator, pairSeparator, prefix, suffix])
```

Arguments:

- ***separator***: the character/substring used to separate pairs within the resulting string; defaults to *"; "*;
- ***pairSeparator***: the character/substring used to separate the key from the value within a pair; defaults to *", "*;
- ***prefix***: the character/substring used to prefix a pair; defaults to *"{ "*;
- ***suffix***: the character/substring used to suffix a pair; defaults to *" }"*.

Returns: An array or a string, depending on the method.

Example:

```
var d1 = new JSU.Dictionary(["c", 1], ["a", 2], ["b", 1]);
console.debug(d1.toArray()); // [ ["c", 1], ["a", 2], ["b", 1] ]
console.debug(d1.toPairsArray()); // [ {"c", 1}, {"a", 2}, {"b", 1} ]
console.debug(d1.toString()); // "{c, 1}; {a, 2}; {b, 1}"
console.debug(d1.toString(",",";","(",")")); // "(c;1),(a;2),(b;1)"
```

G. Predicate-based Methods

1. exists

Checks if a pair matching a specified predicate exists within the dictionary.

Syntax:

```
exists(predicate)
```

Arguments:

- ***predicate***: the function used to check for matching pairs.

Predicate Syntax:

```
function(index, key, value)
```

Predicate Arguments:

- ***index***: the position at which the iteration arrived;
- ***key***: the key for the pair at which the iteration arrived;
- ***value***: the value for the pair at which the iteration arrived.

Predicate must return: A boolean indicating whether the current pair is a match.

Returns: A boolean.

Example:

```
var d1 = new JSU.Dictionary(["c", 1], ["a", 12], ["b", 5]);

var pred = function(i, k, v) {
    return v < 6;
}
alert(d1.exists(pred)); // true
```

2. find (~ *findFirst*)

Finds the first pair that matches a specified predicate.

Syntax:

```
find(predicate)      alias: findFirst(predicate)
```

Arguments:

- ***predicate***: the function used to check for matching pairs.

Predicate Syntax:

```
function(index, key, value)
```

Predicate Arguments:

- ***index***: the position at which the iteration arrived;
- ***key***: the key for the pair at which the iteration arrived;
- ***value***: the value for the pair at which the iteration arrived.

Predicate must return: A boolean indicating whether the current pair is a match.

Returns: A JSU.Pair.

Example:

```
var d1 = new JSU.Dictionary(["c", 1], ["a", 12], ["b", 5]);

var pred = function(i, k, v) {
    return v < 6;
}
alert(d1.find(pred)); // {"c", 1}
```

3. findAll (~ *filter*, ~ *accepted*)

Finds all the pairs that match a specified predicate.

Syntax:

```
findAll(pred)        alias: filter(pred), accepted(pred)
```

Arguments:

- ***pred***: the function used to check for matching elements.

Predicate Syntax:

```
function(index, key, value)
```

Predicate Arguments:

- ***index***: the position at which the iteration arrived;
- ***key***: the key for the pair at which the iteration arrived;
- ***value***: the value for the pair at which the iteration arrived.

Predicate must return: A boolean indicating whether the current pair is a match..

Returns: A JSU.Dictionary with the matching objects.

Example:

```
var d1 = new JSU.Dictionary(["c", 1], ["a", 12], ["b", 5]);

var pred = function(i, k, v) {
    return v < 6;
}
alert(d1.findAll(pred)); // { {"c", 1}, {"b", 5} }
```

4. findIndex (~ *findFirstIndex*)

Finds the first index of the pair matching a specified predicate.

Syntax:

```
findIndex(pred)      alias: findFirstIndex(pred)
```

Arguments:

- ***pred***: the function used to check for matching elements.

Predicate Syntax:

```
function(index, key, value)
```

Predicate Arguments:

- ***index***: the position at which the iteration arrived;
- ***key***: the key for the pair at which the iteration arrived;
- ***value***: the value for the pair at which the iteration arrived.

Predicate must return: A boolean indicating whether the current pair is a match.

Returns: A number.

Example:

```
var d1 = new JSU.Dictionary(["c", 1], ["a", 12], ["b", 5]);

var pred = function(i, k, v) {
    return v < 6;
}
alert(d1.findIndex(pred)); // 0
```

5. findIndexes

Finds all the indexes at which a matching pair occurs.

Syntax:

```
findIndexes(pred)
```

Arguments:

- ***pred***: the function used to check for matching pairs.

Predicate Syntax:

```
function(index, key, value)
```

Predicate Arguments:

- ***index***: the position at which the iteration arrived;
- ***key***: the key for the pair at which the iteration arrived;
- ***value***: the value for the pair at which the iteration arrived.

Predicate must return: A boolean indicating whether the current pair is a match.

Returns: An array with the indexes.

Example:

```
var d1 = new JSU.Dictionary(["c", 1], ["a", 12], ["b", 5]);

var pred = function(i, k, v) {
    return v < 6;
}
alert(d1.findIndexes(pred)); // [0, 2]
```

6. findLast

Finds the last pair that matches a specified predicate.

Syntax:

```
findLast(predicate)
```

Arguments:

- ***predicate***: the function used to check for matching pairs.

Predicate Syntax:

```
function(index, key, value)
```

Predicate Arguments:

- ***index***: the position at which the iteration arrived;
- ***key***: the key for the pair at which the iteration arrived;
- ***value***: the value for the pair at which the iteration arrived.

Predicate must return: A boolean indicating whether the current pair is a match.

Returns: An object.

Example:

```
var d1 = new JSU.Dictionary(["c", 1], ["a", 12], ["b", 5]);

var pred = function(i, k, v) {
    return v < 6;
}
alert(d1.findLast(pred)); // {"b", 5}
```

7. findLastIndex

Finds the last index at which a matching pair occurs.

Syntax:

```
findLastIndex(pred)
```

Arguments:

- ***pred***: the function used to check for matching pairs.

Predicate Syntax:

```
function(index, key, value)
```

Predicate Arguments:

- ***index***: the position at which the iteration arrived;
- ***key***: the key for the pair at which the iteration arrived;
- ***value***: the value for the pair at which the iteration arrived.

Predicate must return: A boolean indicating whether the current pair is a match.

Returns: A number.

Example:

```
var d1 = new JSU.Dictionary(["c", 1], ["a", 12], ["b", 5]);

var pred = function(i, k, v) {
    return v < 6;
}
alert(d1.findLastIndex(pred)); // 2
```

8. skipWhile (~ *removeWhile*) / inverseSkipWhile (~ *inverseRemoveWhile*)

Similar to the skipping extraction methods **skip** / **inverseSkip**, except that these two will skip pairs from the dictionary while the pairs match a specified predicate.

Syntax:

```
skipWhile(pred)       alias: removeWhile(pred)
inverseSkipWhile(pred)        alias: inverseRemoveWhile(pred)
```

Arguments:

- ***pred***: the function used to check for matching pairs.

Predicate Syntax:

```
function(index, key, value)
```

Predicate Arguments:

- ***index***: the position at which the iteration arrived;
- ***key***: the key for the pair at which the iteration arrived;
- ***value***: the value for the pair at which the iteration arrived.

Predicate must return: A boolean indicating whether the current pair is a match.

Returns: A JSU.Dictionary.

Example:

```
var d1 = new JSU.Dictionary(["c", 1], ["a", 12], ["b", 5]);

var pred = function(i, k, v) {
    return v < 6;
}
alert(d1.skipWhile(pred)); // { {"a", 12}, {"b", 5} }
alert(d1.inverseSkipWhile(pred)); // { {"a", 12}, {"c", 1} }
alert(d1.inverseSkipWhile(pred).reverse()); // { {"c", 1}, {"a", 12} }
```

9. takeWhile (~ *firstWhile*) / inverseTakeWhile (~ *lastWhile*)

Similar to the **take** / **inverseTake** extraction methods, except that these two will take pairs from the dictionary while the pairs match a specified predicate.

Syntax:

```
takeWhile(pred)      alias: firstWhile(pred)
inverseTakeWhile(pred)       alias: lastWhile(pred)
```

Arguments:

- ***pred***: the function used to check for matching pairs.

Predicate Syntax:

```
function(index, key, value)
```

Predicate Arguments:

- ***index***: the position at which the iteration arrived;
- ***key***: the key for the pair at which the iteration arrived;
- ***value***: the value for the pair at which the iteration arrived.

Predicate must return: A boolean indicating whether the current pair is a match.

Returns: A JSU.Dictionary.

Example:

```
var d1 = new JSU.Dictionary(["c", 1], ["a", 12], ["b", 5]);

var pred = function(i, k, v) {
    return v < 6;
}
alert(d1.takeWhile(pred)); // { {"c", 1} }
alert(d1.lastWhile(pred)); // { {"b", 5} }
```

10. forEach / forEachIf

These two methods are used for manipulating pair's values (updating only). The first one will iterate through all the pairs, while the last one will iterate through the dictionary's pairs and update their values only if they match a specified condition.

Syntax:

```
forEach(updatePred)
forEachIf(updatePred, conditionPred)
```

Arguments:

- ***updatePred***: the function used to update pairs' values;
- ***conditionPred***: the function that checks whether a condition is met or not.

Predicate Syntax:

```
function(index, key, value)
```

Predicate Arguments:

- ***index***: the position at which the iteration arrived;
- ***key***: the key for the pair at which the iteration arrived;
- ***value***: the value for the pair at which the iteration arrived.

Predicate must return: A boolean in case of the condition predicate, and the modified value in case of the update predicate.

Returns: A JSU.Dictionary.

Example:

```
var d1 = new JSU.Dictionary(["c", 2], ["a", 12], ["b", 5]);

var pred = function(i, k, v) {
    return v < 6;
};
var update = function(i, k, v) {
    return v * v;
};

alert(d1.forEach(update)); // { {"c", 4}, {"a", 144}, {"b", 25} }

d1 = new JSU.Dictionary(["c", 2], ["a", 12], ["b", 5]);
alert(d1.forEachIf(update, pred)); // { {"c", 4}, {"a", 12}, {"b", 25} }
```

11. reverseForEach (~ *inverseForEach*) / reverseForEachIf (~ *inverseForEachIf*)

These two methods work almost the same with **forEach** / **forEachIf**, except that these two will iterate through all the pairs starting at the end of the dictionary towards the beginning, so the big difference comes with the predicates that will give you the indexes and pairs in reverse order, but the results will not be reversed. The syntax, arguments, predicates and

returning values are the same, so for more information check the previous two documented methods.

Example:

```
var d1 = new JSU.Dictionary(["c", 2], ["a", 12], ["b", 5]);

var pred = function(i, k, v) {
    return v < 6;
};
var update = function(i, k, v) {
    return v * v;
};

alert(d1.reverseForEach(update)); // { {"c", 4}, {"a", 144}, {"b", 25} }

d1 = new JSU.Dictionary(["c", 2], ["a", 12], ["b", 5]);
alert(d1.reverseForEachIf(update, pred));
          // { {"c", 4}, {"a", 12}, {"b", 25} }
```

12. rejected

This will work the opposite as to the **accepted** method. While the **accepted** method will return all the pairs that match a specified predicate, the **rejected** method will return the other pairs (that didn't match the predicate).

Syntax:

```
rejected(pred)
```

Arguments:

- ***pred***: the function used to check for matching pairs.

Predicate Syntax:

```
function(index, key, value)
```

Predicate Arguments:

- ***index***: the position at which the iteration arrived;
- ***key***: the key for the pair at which the iteration arrived;
- ***value***: the value for the pair at which the iteration arrived.

Predicate must return: A boolean indicating whether the current pair is a match.

Returns: A JSU.Dictionary with the non-matching pairs.

Example:

```
var d1 = new JSU.Dictionary(["c", 2], ["a", 12], ["b", 5]);

var pred = function(i, k, v) {
    return v < 6;
};
alert(d1.rejected(pred)); // { {"a", 12} }
```

13. removeAll

This method will remove all the pairs that match a specified predicate.

Syntax:

```
removeAll(pred)
```

Arguments:

- ***pred***: the function used to check for matching elements.

Predicate Syntax:

```
function(index, key, value)
```

Predicate Arguments:

- ***index***: the position at which the iteration arrived;
- ***key***: the key for the pair at which the iteration arrived;
- ***value***: the value for the pair at which the iteration arrived.

Predicate must return: A boolean indicating whether the current pair is a match.

Returns: A JSU.Dictionary.

Example:

```
var d1 = new JSU.Dictionary(["c", 2], ["a", 12], ["b", 5]);

var pred = function(i, k, v) {
    return v < 6;
};
alert(d1.removeAll(pred)); // { {"a", 12} }
```

14. trueForAll

Checks if all the dictionary's pairs meet a specified condition.

Syntax:

```
trueForAll(pred)
```

Arguments:

- ***pred***: the function used to check for matching elements.

Predicate Syntax:

```
function(index, key, value)
```

Predicate Arguments:

- ***index***: the position at which the iteration arrived;
- ***key***: the key for the pair at which the iteration arrived;
- ***value***: the value for the pair at which the iteration arrived.

Predicate must return: A boolean indicating whether the current pair is a match.

Returns: A boolean.

Example:

```
var d1 = new JSU.Dictionary(["c", 2], ["a", 12], ["b", 5]);

var pred = function(i, k, v) {
    return v < 6;
};
alert(d1.trueForAll(pred));
    // false, {"a", 12} doesn't meet the condition
```

15. trueForOne

Checks if only one pair of the dictionary meets a specified condition. If none or more than one meet the condition, it returns false. Syntax, arguments and return value are identical to the **trueForAll** method, so for more information check that method's documentation.

Example:

```
var d1 = new JSU.Dictionary(["c", 2], ["a", 12], ["b", 5]);

var pred = function(i, k, v) {
    return v > 6;
};
alert(d1.trueForOne(pred)); // true
```

16. trueForAny

Checks if at least one pair of the dictionary meets a specified condition. Syntax, arguments and return value are identical to the **trueForAll** method, so for more information check that method's documentation.

Example:

```
var d1 = new JSU.Dictionary(["c", 2], ["a", 12], ["b", 5]);

var pred = function(i, k, v) {
    return v > 6;
};
alert(d1.trueForAny(pred)); // true
```

17. trueForN

Checks if exactly n pairs from the dictionary meet a specified condition.

Syntax:

```
trueForN(n, pred)
```

Arguments:

- ***n***: the number of pairs that must match the specified condition.
- ***pred***: the function used to check for matching pairs.

Predicate Syntax:

```
function(index, key, value)
```

Predicate Arguments:

- ***index***: the position at which the iteration arrived;
- ***key***: the key for the pair at which the iteration arrived;
- ***value***: the value for the pair at which the iteration arrived.

Predicate must return: A boolean indicating whether the current pair is a match.

Returns: A boolean.

Example:

```
var d1 = new JSU.Dictionary(["c", 2], ["a", 12], ["b", 5]);

var pred = function(i, k, v) {
    return v > 6;
};
alert(d1.trueForN(1, pred)); // true
alert(d1.trueForN(0, pred)); // false
alert(d1.trueForN(2, pred)); // false
```

H. Comparer Methods

1. resetComparers / resetExactComparers

These methods are used to reset the comparer methods, usually if the developer added one or more custom comparers (thus disabling all custom comparers).

Syntax:

```
resetComparers()
resetExactComparers()
```

Returns: Nothing.

Example:

```
d1 = new JSU.Dictionary();
d1.resetComparers(); // default comparers will be used
d1.resetExactComparers(); // default comparers will be used
```

2. addComparer / addExactComparer

These methods allow you to add your custom comparer methods so that you teach the JSU.Dictionary object how to handle custom objects. Notice that you can add comparer methods only to instances of the JSU.Dictionary object, allowing you this way to have dictionaries with different comparer methods depending on what you store in them.

Syntax:

```
addComparer(comp)
addExactComparer(comp)
```

Comparer Syntax:

```
function(a, b)
```

Predicate Arguments:

- ***a, b***: the objects that will be checked for equality at any time of the iteration.

Predicate must return: A boolean indicating whether the current objects are equal or not.

Returns: Nothing.
Note: The comparers must always return a boolean, otherwise your could hang the browser in an infinite loop. Also, be careful to use strict equality (===) for the exact comparers, and normal equality (==) for the standard comparers, or you'll get unexpected results.

Example:

```
d1 = new JSU.Dictionary();
var comp = function(a, b) {
    return a == b;
};
d1.addComparer(comp);
```

I. How the Method Chaining works

The JSU.Dictionary module, being an instantiable module, allows you to chain methods together so that you write less repetitive code. Below are two examples of code without and with method chaining in action:

```
var d = new JSU.Dictionary();

d.initWith(null, 2); // { {"0", null}, {"1", null} }
d.set("0", 0); // { {"0", 0}, {"1", null} }
alert(d.hasValue(0)); // true
```

and with method chaining:

```
var d = new JSU.Dictionary().initWith(null, 2);

alert(d.set("0", 0).hasValue(0)); // true
```

A better way of writing code that uses method chaining is to put every method on a new line (increasing readability), like this:

```
var d = new JSU.Dictionary();

alert(d.initWith(null, 2)
       .set("0", 0)
       .hasValue(0)); // true
```

But how does it work ? It's quite simple, almost obvious - almost all methods (excepting most of the *Information Methods*) return the current JSU.Dictionary object after altering it depending on the called methods. So, in our last example, **initWith** method will return the new dictionary object filled with pairs who's values are null, and the return value being a JSU.Dictionary you can call another method on it - for example **set**, which sets the value of the pair identified by key "0" to 0. The **set** method also returned the modified JSU.Dictionary object, so we call the **hasValue** method to check whether it has a pair containing a 0 value. Note that calling the **hasValue** method will break the chain because it returns a boolean value.

J. Sorting dictionaries using a custom comparing method

The Dictionary's sort method allows you to specify your own comparing method for comparing two objects (keys or values) from the dictionary, at any time of the sorting iterations. By default, the sort method uses a standard comparing method which compares the whole objects with the usual comparing operators (">" and "<"). So, if you'd want to sort a dictionary based on some property of each object, you'd have to create your custom comparing method. This chapter will give you an example of how do that. Let's first look at the whole code, and then explain what it does:

```
// the task used by all the timers
var task = function() { };

// define the timer objects
var t1 = new JSU.Timer(task, 1000),
    t2 = new JSU.Timer(task, 2000),
    t3 = new JSU.Timer(task, 1500);

// create a new dictionary with the timers
var d1 = new JSU.Dictionary(["t1", t1], ["t2", t2], ["t3", t3]);

// define the comparing method
var comp = function(a, b, dir) {
    if (a.getInterval() > b.getInterval())
        return (dir == "DESC" ? -1 : 1);
    else if (a.getInterval() < b.getInterval())
        return (dir == "DESC" ? 1 : -1);
    else
        return 0;
};

// sort the dictionary descending by value based on the timers' interval
d1.sortByValue("DESC", comp); // l1 = { {"t2", t2}, {"t3", t3}, {"t1", t1} }
```

In this example I've used the JSU.Timer object (also an instantiable module) which is used to execute a specified task (a function) at a regular interval of time. So, the example above creates a dictionary of timers with different intervals, and then sorts the dictionary descending by value based on each timer's interval. Let's look at the individual pieces of code:

```
// the task used by all the timers
var task = function() { };
```

For simplicity reasons, I've used the same task for every timer - a task that doesn't do anything at all.

```
// define the timer objects
var t1 = new JSU.Timer(task, 1000),
    t2 = new JSU.Timer(task, 2000),
    t3 = new JSU.Timer(task, 1500);
```

Here I've defined three timers, each of them using the same task but having different intervals at which the task will run.

```
// create a new dictionary with the timers
var d1 = new JSU.Dictionary(["t1", t1], ["t2", t2], ["t3", t3]);
```

I've created a new dictionary containing all the timers. Notice that the dictionary isn't sorted by default because of the order in which I've added the timers.

```
// define the comparing method
var comp = function(a, b, dir) {
    if (a.getInterval() > b.getInterval())
        return (dir == "DESC" ? -1 : 1);
    else if (a.getInterval() < b.getInterval())
        return (dir == "DESC" ? 1 : -1);
    else
        return 0;
};
```

Now we get to the point where we have to create the custom comparing method. Following the documentation, this method must have 2 or 3 arguments: the two objects to compare, and an optional argument which represents the sorting direction used by the **sort** method. Also, it must return one of three values: -1 when $a < b$, 0 when $a = b$, 1 when $a > b$.

As you can see, the comparing method also takes into account the direction of sorting. Ignoring this argument would always sort the dictionary ascending (except if you make sure the comparison works the other way around).

In this method we compare the timers' intervals using the **getInterval** method (check the JSU.Timer module documentation for more information). The tricky part comes when we have to return a value. Let's think this a little: if a's interval is lower than b's interval, then we should return -1, right ? Half right. If we sort the dictionary ascending, it's the right value to return, but when we sort it descending, we have to return the opposite value, and that is 1. This is what I've done in this example by checking the direction of sorting. And finally:

```
// sort the dictionary descending by value based on the timers' interval
d1.sortByValue("DESC", comp); // l1 = { {"t2", t2}, {"t3", t3}, {"t1", t1} }
```

We sort the dictionary descending, and we see that it worked - the timers are sorted from the highest interval to the lowest. And that's pretty much about it.

K. Teaching the dictionary how to handle new types

There are times when you have to deal with more than just native JavaScript objects (numbers, strings, and so on). For example, the JSU.Dictionary module has two methods for checking the equality of two dictionaries that store native objects only, but these methods will not work as expected when you add custom objects to a dictionary, because it doesn't know how to treat them. This chapter will show you how to "*teach*" the dictionary to compare custom objects, allowing you to use more than just equality methods on them and obtain the expected results.

How to do it

To avoid repeating myself, I'll break up the code in a few pieces:

```
// create three sets
var s1 = new JSU.Set(1, 2, 3),
    s2 = new JSU.Set(1, 2, 3),
    s3 = new JSU.Set("1", "2", "3");

// create dictionary containing sets as a value
// for one pair (it's enough to check equality)
var d1 = new JSU.Dictionary(["name", "test"], ["value", s1]),
    d2 = new JSU.Dictionary(["name", "test"], ["value", s3]);
```

This code creates three dictionaries containing some JSU.Set objects as values (for more information about sets, check the *JSU.Set API Reference*). These lines are just for the sake of showing you how stuff works, and for that I must have some sample objects to test on.

```
// wrong results
console.debug(d1.equals(d2)); // false
console.debug(d1.same(d2)); // false
```

When running the above code, you can see that we got some wrong results. Let's dig a little bit through these alert calls:

- *d1.equals(d2)* - this should return true; if you look at the initialization code, you can see that s1 == s3 (not ===);
- *d1.same(d2)* - this should also return false, and so it does, but it's just an exception.

Now, the comparers:

```
// create the comparer
var dcomparer = function(a, b) {
    var ok = true;
    if (JSU.isJSUSet(a.getKey()) && JSU.isJSUSet(b.getKey()))
        ok = ok && JSU.Array.equals(a.getKey().toArray(),
b.getKey().toArray());
    if (JSU.isJSUSet(a.getValue()) && JSU.isJSUSet(b.getValue()))
        ok = ok && JSU.Array.equals(a.getValue().toArray(),
b.getValue().toArray());
    return ok;
};
```

```
// create the exact comparer
var dexactComparer = function(a, b) {
    var ok = true;
    if (JSU.isJSUSet(a.getKey()) && JSU.isJSUSet(b.getKey()))
        ok = ok && JSU.Array.same(a.getKey().toArray(),
b.getKey().toArray());
    if (JSU.isJSUSet(a.getValue()) && JSU.isJSUSet(b.getValue()))
        ok = ok && JSU.Array.same(a.getValue().toArray(),
b.getValue().toArray());
    return ok;
};
```

As a note, you should write this code only once for a new custom object, and then use it with the **addComparers** / **addExactComparers** methods. Now let's explain what this code does:

- the normal comparer checks the key and value to see if one of them are a JSU.Set and, if so, it will convert the key or value sets to arrays and compare them using the **equals** method of the JSU.Array module; another way of doing this would be to loop through all the sets's items and check for equality for each pair of the sets' values, but I consider this method of using arrays more easy to write and read, and also bug-free;
- the exact comparer does the same thing as the normal comparer, except that it uses the JSU.Array's **same** method.

After creating the comparers, you have to add them to the calling dictionary's comparers:

```
// add the new comparers to the dictionary
// calling the methods that use comparers
d1.addComparer(dcomparer);
d1.addExactComparer(dexactComparer);
```

Beware that after these lines of code, the d2 dictionary will be the same and only d1 will have the new comparers. Why ? Because in this case we'll only use d1's methods to check equality. If you know you'll also use d2's methods to compare d2 to other dictionaries, you'll have to add those comparers to d2, too. And now, after adding the new comparers, let's see what we get:

```
// correct results
console.debug(d1.equals(d2)); // true
console.debug(d1.same(d2)); // false
```

We got the correct results, as expected!