ZINCKSOFT

Give The Dream New Life.

# JSU.List

*The complete Module API Reference*

# Quick Introduction

The JSU List Module (**JSU.List / JSUList**) provides a new JavaScript data type for storing everything you want, from native objects to custom objects, and even other lists. The List module offers you an alternative to JavaScript's arrays, with tons of features for all your needs, features that bring the list concept to a whole new level.

It's true, arrays also let you store any objects you want, but they can't learn to handle new data types. Yes, you can teach a list 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 list using custom comparing methods***, but also ***how to teach the List 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.list.js"></script>
```

# JSU.List API Reference

## A. Constructor

***Syntax:***

```
JSU.List([obj1 [, obj2 [, obj3 ...]]])
```

***Arguments:***

- ***obj1-N***: the objects to add to the list; if there's no argument specified, the list will be created empty.

***Returns:*** The JSU.List object.

***Example:***

```
var l1 = new JSU.List(); // empty list
var l2 = new JSU.List(1, [2, 3], "4", null, l1);
    // l2 is a list with a number, a two-element array,
    // a string, a null value and the empty list l1
var l3 = new JSU.List(l2);
    // creates an exact copy (not a reference) of l2
```

## 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 list's internal structure (an array).

### 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 list'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 list's elements.

## C. Static Methods

### 1. fromArray

This static method will create a new List from an existing array.

***Syntax:***

```
JSU.List.fromArray(arr)
```

***Arguments:***

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

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

***Example:***

```
var l = JSU.List.fromArray( [2, "3"] );
```

### 2. fromString

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

***Syntax:***

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

***Arguments:***

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

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

***Example:***

```
var l = JSU.List.fromString("2, 3,'4',[5]");
var l = JSU.List.fromString("2; 3;'4';[5,2]", ";");
```

Creating a list from a string is restrictive because you can only add string elements to the list. In the first line from the above example code, we get a new JSU.List containing only strings - "*2*", "*3*", " *'4'* ", "*[5]*", so you cannot add other types (use **fromArray** method to add any type of object to the list).

## D. Manipulation Methods

### 1. clear (~ *empty*)

This method will empty a list.

***Syntax:***

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

***Returns:*** The empty JSU.List.

***Example:***

```
var l = new JSU.List(1, [2, 3], "4");
l.clear(); // the list is now empty
```

### 2. initWith

Creates a new List containing a specified number of the same object.

***Syntax:***

```
initWith(val, size)
```

***Arguments:***

- ***val***: the object used to fill the list;
- ***size***: the size of the new list; must be at least 1.

***Returns:*** A new JSU.List.

***Example:***

```
var l = new JSU.List();
l.initWith(" ", 4); // contains 4 white spaces
```

### 3. push

Adds a new element at the end of the list.

***Syntax:***

```
push(obj)
```

***Arguments:***

- ***obj***: the object to add.

***Returns:*** A JSU.List.

***Example:***

```
var l = new JSU.List();
l.push("12"); // list contains one string
```

### 4. pop

Gets the last element from the list and then removes it.

***Syntax:***

```
pop()
```

***Returns:*** An object.

***Example:***

```
var l = new JSU.List(1, 10, 100);
alert(l.pop()); // outputs 100
alert(l.size()); // outputs 2, list = {1, 10}
```

### 5. shift

Similar to **pop**, only it gets the first element then removes it from the list.

***Syntax:***

```
shift()
```

***Returns:*** An object.

***Example:***

```
var l = new JSU.List(10, 100, 1);
alert(l.shift()); // outputs 10
alert(l.size()); // outputs 2, list = {100, 1}
```

### 6. unshift

Similar to **push**, only it inserts an object at the beginning of the list.

***Syntax:***

```
unshift(obj)
```

***Arguments:***

- ***obj***: the object to insert.

***Returns:*** A JSU.List.

***Example:***

```
var l = new JSU.List(10, 100);
l.insert("12"); // list = {12, 10, 100}
```

### 7. add

Adds objects to a list, similar to the JSU.List's constructor method.

***Syntax:***

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

***Arguments:***

- ***obj1-N***: the objects to add to the list.

***Returns:*** A JSU.List.

***Example:***

```
l.add(1, [2, 3], "4");
l.add(5);
```

### 8. addFromList / addFromLists

Adds content from other JSU.List objects. The first method allows you to add contents from one list, while the last one lets you add content from more lists.

***Syntax:***

```
addFromList(list)
addFromLists(list1 [, list2 [, list3 ...]])
```

***Arguments:***

- ***list, list1-N***: the lists from which the content will be added to the calling list.

***Returns:*** A JSU.List.

***Example:***

```
var l1 = new JSU.List(1, 10, 100);
var l2 = new JSU.List("2", "3");
var l3 = new JSU.List([]);
l3.addFromLists(l1, l2); // l3 = { [], 1, 10, 100, "2", "3" }
```

### 9. addFromArray / addFromString

Adds content from an array or a string containing the new elements. These are similar to the static methods **fromArray** / **fromString**, the only difference being that these two are used only on instances of the JSU.List object. The syntax and arguments are the same, and return the modified calling list, so check the *Static Methods* chapter for more information.

### 10. set

Sets the value of an item from the list at a specified index. Notice that this method cannot be used to add new items, but only for modifying existing ones.

***Syntax:***

```
set(index, value)
```

***Arguments:***

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

***Returns:*** A JSU.List.

***Example:***

```
var l1 = new JSU.List(1, 10, 100);
l1.set(2, 0); // l1 = {1, 10, 0}
```

### 11. reverse

Reverses the List's items.

***Syntax:***

```
reverse()
```

***Returns:*** A JSU.List.

***Example:***

```
var l1 = new JSU.List(1, 10, 100);
l1.reverse(); // l1 = {100, 10, 1}
```

### 12. insert

Inserts a new object at the beginning of the list. 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 l1 = new JSU.List(1, 10, 100);
l1.insert(0); // l1 = {0, 1, 10, 100}
```

### 13. insertFromArray / insertFromString

Adds content at the beginning of the list from an array or a string containing the new elements. These are similar to the static methods **fromArray** / **fromString**, the only differences being that adding is done at the beginning of the list, and these two are used only

on instances of the JSU.List object. The syntax and arguments are the same, and return the modified calling list, so check the *Static Methods* chapter for more information.

### 14. insertFromList / insertFromLists

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

### 15. insertAt

Inserts one or more objects at a specified location within the list.

***Syntax:***

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

***Arguments:***

- ***index***: the position where the insertion will take place;
- ***obj1-N***: the objects to insert.

***Returns:*** A JSU.List.

***Example:***

```
var l1 = new JSU.List(1, 10, 100);
l1.insertAt(1, "a", "b"); // l1 = { 1, "a", "b", 10, 100 }
```

### 16. insertFromListAt / insertFromListsAt

Inserts contents from one or more lists at a specified location within the calling list.

***Syntax:***

```
insertFromListAt(index, list)
insertFromListsAt(index, list1 [, list2, [ list3 ... ]])
```

***Arguments:***

- ***index***: the position where the insertion will take place;
- ***list, list1-N***: the lists from which the content will be gathered.

***Returns:*** A JSU.List.

***Example:***

```
var l1 = new JSU.List(1, 10, 100);
var l2 = new JSU.List(2, 20, 200);
var l3 = new JSU.List(3, 30, 300);
l3.insertFromListsAt(3, l2, l1);
    // l3 = { 3, 30, 300, 2, 20, 200, 1, 10, 100 }
```

Notice how insertion took place at the 3rd position (counting starts at 0), even if all the lists have only 3 elements (indexes: 0, 1, 2). Specifying a higher index than the usually allowed maximum index will mimic the **add** method, creating a new slot at the end of the list.

### 17. insertFromArrayAt / insertFromStringAt

Inserts content at a specified position within the list from an array or a string containing the new elements.

***Syntax:***

```
insertFromArrayAt(index, arr)
insertFromStringAt(index, str [, separator])
```

***Arguments:***

- ***index***: the position where the insertion will take place;
- ***arr***: the array used as a data source;
- ***str***: the string used as a data source;
- ***separator***: the character/substring used to separate elements within the string data source.

***Returns:*** A JSU.List.

***Example:***

```
var l1 = new JSU.List(1, 10, 100);
l1.insertFromArrayAt(0, [1, 2, 0]);
    // l1 = { 1, 2, 0, 1, 10, 100 }
l1.insertFromStringAt(0, "1;2;5", ";");
    // l1 = { "1", "2", "5", 1, 2, 0, 1, 10, 100 }
```

### 18. remove

Removes one or more objects from the list.

***Syntax:***

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

***Arguments:***

- ***obj1-N***: the objects to be removed.

***Returns:*** A JSU.List.

***Example:***

```
var l1 = new JSU.List(1, 10, 100, 1, 2, 3);
l1.remove(1, 2); // l1 = {10, 100, 3}
l1.remove(5); // l1 remains the same
```

### 19. removeAt

Removes the object at a specified position within the list.

***Syntax:***

```
removeAt(index)
```

***Arguments:***

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

***Returns:*** A JSU.List.

***Example:***

```
var l1 = new JSU.List(1, 10, 100, 1, 2, 3);
l1.removeAt(1); // l1 = {1, 100, 1, 2, 3}
```

### 20. removeRange

Removes a range of objects from the list.

***Syntax:***

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

***Arguments:***

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

***Returns:*** A JSU.List.

***Example:***

```
var l1 = new JSU.List(1, 10, 100, 1, 2, 3);
l1.removeRange(1, 3); // l1 = {1, 2, 3}
```

### 21. sort

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

***Syntax:***

```
sort([comparer [, dir]])
sort([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 list; 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 list, 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.List.

***Example:***

```
var l1 = new JSU.List(1, 10, 100, 1, -1, 3);
l1.sort(); // l1 = {-1, 1, 1, 3, 10, 100}
```

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

## E. Information Methods

### 1. length (~ *size*)

Gets or sets the size of the list.

***Syntax:***

```
length([newSize])        alias: size([newSize])
```

***Arguments:***

- ***newSize***: the new size of the list; passing a number lower than the actual size will remove objects from the end of the list until its size reaches newSize; passing a number greater than the actual size will add null objects at the end of the list until its size reaches newSize.

***Returns:*** A number.

***Example:***

```
l1 = new JSU.List(1, 10, 100, 1, 2, 3);
alert(l1.size()); // outputs 6
alert(l1.size(3)); // outputs 3, l1 = {1, 10, 100}
alert(l1.size(4)); // outputs 4, l1 = {1, 10, 100, null}
```

### 2. get (~ *items*)

Gets the object at a specified position within the list.

***Syntax:***

```
get(index)       alias: items(index)
```

***Arguments:***

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

***Returns:*** An object.

***Example:***

```
l1 = new JSU.List(1, 10, 100, 1, 2, 3);
alert(l1.get(3)); // outputs 1
```

### 3. item

Gets or sets the item at a specified position within the list.

***Syntax:***

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

***Arguments:***

- ***index***: the position from which to retrieve the object, or at which to set the new value;
- ***value***: the new value for the object at the specified index.

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

***Example:***

```
l1 = new JSU.List(1, 10, 100, 1, 2, 3);
alert(l1.item(3)); // outputs 1
l1.item(3, 0); // l1 = {1, 10, 100, 0, 2, 3}
```

### 4. contains (~ *has*)

Checks whether the list contains a specified object.

***Syntax:***

```
contains(obj)          alias: has(obj)
```

***Arguments:***

- ***obj***: the object to search for.

***Returns:*** A boolean.

***Note:*** This method uses the default exact comparers.

***Example:***

```
l1 = new JSU.List(1, 10, 100, 1, 2, 3);
alert(l1.contains(1)); // true
alert(l1.contains(0)); // false
alert(l1.contains("1")); // false
```

### 5. binarySearch

Finds the index of an object using the binary search method. Be careful that this method *must be called ONLY after sorting the list*. Not sorting the list before the call of this method could result in an infinite loop breaking your browser's execution.

***Syntax:***

```
binarySearch(obj)
```

***Arguments:***

- ***obj***: the object to search for.

***Returns:*** A number.

***Example:***

```
l1 = new JSU.List(1, 10, 100, 1, 2, 3);
l1.sort(); // l1 = {1, 1, 2, 3, 10, 100}
alert(l1.binarySearch(100)); // outputs 5
```

### 6. equals / same

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

***Syntax:***

```
equals(list)
same(list)
```

***Arguments:***

- ***list***: the list to compare to.

***Returns:*** A boolean.

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

***Example:***

```
l1 = new JSU.List(1, 10, 100, 1, 2, 3);
l2 = new JSU.List("1", 10, 100, 1, 2, 3);
alert(l1.equals(l2)); // true
alert(l1.same(l2)); // false
```

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

### 7. getTypeOf (~ *getTypeAt*)

This method gets the type of the object found at a specified position within the list. 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:***

```
getTypeOf(index)      alias: getTypeAt(index)
```

***Arguments:***

- ***index***: the index of the object to check for its type;

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

***Example:***

```
l1 = new JSU.List("1", 10, new JSU.List());
alert(l1.getTypeOf(1)); // number
alert(l1.getTypeOf(0)); // string
alert(l1.getTypeOf(2)); // jsulist
```

### 8. indexOf / indexesOf / lastIndexOf

Gets the index or indexes of a specified object.

***Syntax:***

```
indexOf(obj)
indexesOf(obj)
lastIndexOf(obj)
```

***Arguments:***

- ***obj***: the object to search for.

***Returns:*** **indexOf** and **lastIndexOf** return a number, while **indexesOf** returns an array containing all the indexes at which the object was found.

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

***Example:***

```
l1 = new JSU.List(1, 2, 3, 1, 0, 1, 12);
alert(l1.indexOf(1)); // 0
alert(l1.lastIndexOf(1)); // 5
alert(l1.indexesOf(1)); // [0, 3, 5]
```

### 9. count

Counts how many times a specified object occurs within the list.

***Syntax:***

```
count(obj)
count(predicate)
```

***Arguments:***

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

***Predicate Syntax:***

```
function(index, obj)
```

***Predicate Arguments:***

- ***index***: the index at which the searching takes place;
- ***obj***: the object 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:***

```
l1 = new JSU.List(1, 12, 3, 1, 5, 1, 5);
alert(l1.count(1)); // 3

var match = function(index, obj) {
    // any object found at an even index
    // and has a value < 5 is a match
    return (index + 1) % 2 == 0 && obj < 5;
};
alert(l1.count(match)); // 2 (the last two 1's)
```

### 10. first / last

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

***Syntax:***

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

***Arguments:***

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

***Returns:*** If no argument is passed, the methods return the first/last object from the list, otherwise it returns a sublist containing the first/last *n* objects from the list.

***Example:***

```
l1 = new JSU.List(1, 12, 3, 1, 5, 1, 6);
alert(l1.first()); // 1
alert(l1.last()); // 6
alert(l1.first(3)); // {1, 12, 3}
alert(l1.last(3)); // {5, 1, 6}
```

### 11. isEmpty

Checks if the list is empty.

***Syntax:***

```
isEmpty()
```

***Returns:*** A boolean.

***Example:***

```
l1 = new JSU.List(1, 12, 3, 1, 5, 1, 6);
alert(l1.isEmpty()); // false
alert(new JSU.List().isEmpty()); // true
```

### 12. isTyped

Checks if all the list's elements are of the same type.

***Syntax:***

```
isTyped()
```

***Returns:*** A boolean.

***Example:***

```
l1 = new JSU.List(1, 12, 3, 1, 5, 1, 6);
alert(l1.isTyped()); // true
alert(new JSU.List("1", 2).isTyped()); // false
alert(new JSU.List().isTyped()); // false
```

## F. Extraction Methods

### 1. grep

Extracts a sublist with only the items that match a specific regular expression.

***Syntax:***

```
grep(regex)
```

***Arguments:***

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

***Returns:*** A new JSU.List.

***Example:***

```
l1 = new JSU.List(1, 12, 3, 1, 5, 1, 6);
alert(l1.grep(/^[0-5]$/)); // {1, 3, 1, 5, 1}
```

### 2. getRange

Extracts a sublist from a specified range within the calling list.

***Syntax:***

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

***Arguments:***

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

***Returns:*** A new JSU.List.

***Example:***

```
l1 = new JSU.List(1, 12, 3, 1, 5, 1, 6);
alert(l1.getRange(1, 3)); // {12, 3, 1}
```

### 3. getSubList

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

***Syntax:***

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

***Arguments:***

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

***Returns:*** A new JSU.List.

***Example:***

```
l1 = new JSU.List(1, 12, 3, 1, 5, 1, 6);
alert(l1.getSubList(2, 4)); // {3, 1}
alert(l1.getSubList(4)); // {5, 1, 6}
```

### 4. distinct (~ *removeDuplicates*)

Extracts a sublist containing only the distinct elements from the calling list.

***Syntax:***

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

***Returns:*** A new JSU.List.

***Note:*** This method uses the default exact comparer methods.

***Example:***

```
l1 = new JSU.List(1, 12, 3, 1, 5, 1, 6);
alert(l1.distinct()); // {1, 12, 3, 5, 6}
```

### 5. skip / inverseSkip

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

***Syntax:***

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

***Arguments:***

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

***Returns:*** A new JSU.List.

***Example:***

```
l1 = new JSU.List(1, 12, 3, 1, 5, 1, 6);
alert(l1.skip(3)); // {1,5,1,6}
alert(l1.inverseSkip(3)); // {1,3,12,1}
```

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

```
alert(l1.inverseSkip(3).reverse()); // {1,12,3,1}

alert(l1.take(l1.length() - 3)); // {1,12,3,1}
```

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.List 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 objects 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* items from the list in the order they appear, while the **inverseTake** method extracts the last *n* items in reverse order.

***Syntax:***

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

***Arguments:***

- ***count***: the number of objects to take from the list.

***Returns:*** A new JSU.List.

***Example:***

```
l1 = new JSU.List(1, 12, 3, 1, 5, 1, 6);
alert(l1.take(3)); // {1,12,3}
alert(l1.inverseTake(3)); // {6,1,5}

alert(l1.last(3)); // {5,1,6}
```

### 7. toArray / toString

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

***Syntax:***

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

***Arguments:***

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

***Returns:*** An array or a string.

***Example:***

```
l1 = new JSU.List(1, 12, 3, 1, 5, 1, 6);
alert(l1.toArray()); // [1,12,3,1,5,6]
alert(l1.toString()); // "1,12,3,1,5,1,6"
alert(l1.toString(";")); // "1;12;3;1;5;1;6"
```

## G. Predicate-based Methods

### 1. exists

Checks if an element matching a specified predicate exists within the list.

***Syntax:***

```
exists(predicate)
```

***Arguments:***

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

***Predicate Syntax:***

```
function(obj)
```

***Predicate Arguments:***

- ***obj***: the object at which the iteration arrived.

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

***Returns:*** A boolean.

***Example:***

```
l1 = new JSU.List(1, 12, 3, 1, 5, 1, 6);

var pred = function(o) {
    if (o < 5) return true;
    else return false;
};
alert(l1.exists(pred)); // true
```

### 2. find (~ *findFirst*)

Finds the first object that matches a specified predicate.

***Syntax:***

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

***Arguments:***

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

***Predicate Syntax:***

```
function(index, obj)
```

***Predicate Arguments:***

- ***obj***: the object at which the iteration arrived;
- ***index***: the index at which the iteration arrived.

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

***Returns:*** An object.

***Example:***

```
l1 = new JSU.List(1, 12, 3, 1, 5, 1, 6);

var pred = function(i, o) {
    if (o > 5) return true;
    else return false;
};
alert(l1.find(pred)); // 12
```

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

Finds all the objects 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, obj)
```

***Predicate Arguments:***

- ***obj***: the object at which the iteration arrived;
- ***index****:* the index at which the iteration arrived.

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

***Returns:*** A JSU.List sublist with the matching objects.

***Example:***

```
l1 = new JSU.List(1, 12, 3, 1, 5, 1, 6);

var pred = function(i, o) {
    if (o > 5) return true;
    else return false;
};
alert(l1.findAll(pred)); // {12,6}
```

### 4. findIndex (~ *findFirstIndex*)

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

***Syntax:***

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

***Arguments:***

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

***Predicate Syntax:***

```
function(index, obj)
```

***Predicate Arguments:***

- ***obj***: the object at which the iteration arrived;
- ***index****:* the index at which the iteration arrived.

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

***Returns:*** A number.

***Example:***

```
var pred = function(i, o) {
    if (o > 5) return true;
    else return false;
};
alert(l1.findIndex(pred)); // 1
```

### 5. findIndexes

Finds all the indexes at which a matching object occurs.

***Syntax:***

```
findIndexes(pred)
```

***Arguments:***

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

***Predicate Syntax:***

```
function(index, obj)
```

***Predicate Arguments:***

- ***obj***: the object at which the iteration arrived;
- ***index****:* the index at which the iteration arrived.

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

***Returns:*** An array with the indexes.

***Example:***

```
l1 = new JSU.List(1, 12, 3, 1, 5, 1, 8);

var pred = function(i, o) {
    if (o > 5) return true;
    else return false;
};
alert(l1.findIndexes(pred)); // [1,6]
```

### 6. findLast

Finds the last object that matches a specified predicate.

***Syntax:***

```
findLast(predicate)
```

***Arguments:***

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

***Predicate Syntax:***

```
function(index, obj)
```

***Predicate Arguments:***

- ***obj***: the object at which the iteration arrived;
- ***index****:* the index at which the iteration arrived.

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

***Returns:*** An object.

***Example:***

```
l1 = new JSU.List(1, 12, 3, 1, 5, 1, 8);

var pred = function(i, o) {
    if (o > 5) return true;
    else return false;
};
alert(l1.findLast(pred)); // 8
```

### 7. findLastIndex

Finds the last index at which a matching object occurs.

***Syntax:***

```
findLastIndex(pred)
```

***Arguments:***

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

***Predicate Syntax:***

```
function(index, obj)
```

***Predicate Arguments:***

- ***obj***: the object at which the iteration arrived;
- ***index****:* the index at which the iteration arrived.

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

***Returns:*** A number.

***Example:***

```
l1 = new JSU.List(1, 12, 3, 1, 5, 1, 8);

var pred = function(i, o) {
    if (o > 5) return true;
    else return false;
};
alert(l1.findLastIndex(pred)); // 6
```

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

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

***Syntax:***

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

***Arguments:***

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

***Predicate Syntax:***

```
function(index, obj)
```

***Predicate Arguments:***

- ***obj***: the object at which the iteration arrived;
- ***index****:* the index at which the iteration arrived.

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

***Returns:*** A JSU.List sublist.

***Example:***

```
l1 = new JSU.List(1, 4, 3, 12, 5, 1, 2);

var pred = function(i, o) {
    if (o < 5) return true;
    else return false;
};
alert(l1.skipWhile(pred)); // {12,5,1,2}
alert(l1.inverseSkipWhile(pred)); // {5,12,3,4,1}
alert(l1.inverseSkipWhile(pred).reverse()); // {1,4,3,12,5}
```

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

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

***Syntax:***

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

***Arguments:***

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

***Predicate Syntax:***

```
function(index, obj)
```

***Predicate Arguments:***

- ***obj***: the object at which the iteration arrived;
- ***index****:* the index at which the iteration arrived.

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

***Returns:*** A JSU.List sublist.

***Example:***

```
l1 = new JSU.List(1, 4, 3, 12, 5, 1, 2);

var pred = function(i, o) {
    if (o < 5) return true;
    else return false;
};
alert(l1.takeWhile(pred)); // {1,4,3}
alert(l1.lastWhile(pred)); // {1,2}
```

### 10. forEach / forEachIf

These two methods are used for manipulating a list's objects (updating only). The first one will iterate through all the items, while the last one will iterate through the list's items and update them only if they match a specified condition.

***Syntax:***

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

***Arguments:***

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

***Predicates Syntax:***

```
function(index, obj)
```

***Predicates Arguments:***

- ***obj***: the object at which the iteration arrived;
- ***index****:* the index at which the iteration arrived.

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

***Returns:*** A JSU.List sublist.

***Example:***

```
l1 = new JSU.List(1, 4, 3, 12, 5, 1, 2);

var updater = function(i, o) {
    return o * o;
};
var cond = function(i, o) {
    if (o < 5) return true;
    else return false;
};
alert(l1.forEach(updater)); // {1,16,9,144,25,1,4}


l1 = new JSU.List(1, 4, 3, 12, 5, 1, 2);
alert(l1.forEachIf(updater, cond)); // {1,16,9,12,5,1,4}
```

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

These two methods work almost the same with **forEach** / **forEachIf**, except that these two will iterate through all the list's items starting at the end of the list towards the beginning, so the big difference comes with the predicates that will give you the indexes and objects 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:***

```
l1 = new JSU.List(1, 4, 3, 12, 5, 1, 2);

var updater = function(i, o) {
    return o * o;
};
var cond = function(i, o) {
    if (i > 4) return true;
    else return false;
};
alert(l1.reverseForEach(updater)); // {1,16,9,144,25,1,4}

l1 = new JSU.List(1, 4, 3, 12, 5, 1, 2);
alert(l1.reverseForEachIf(updater, cond)); // {1,4,3,12,25,1,4}
```

### 12. rejected

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

***Syntax:***

```
rejected(pred)
```

***Arguments:***

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

***Predicate Syntax:***

```
function(index, obj)
```

***Predicate Arguments:***

- ***obj***: the object at which the iteration arrived;
- ***index****:* the index at which the iteration arrived.

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

***Returns:*** A JSU.List sublist with the non-matching objects.

***Example:***

```
l1 = new JSU.List(1, 4, 3, 12, 5, 1, 2);

var pred = function(i, o) {
    return o < 5;
};
alert(l1.rejected(pred)); // {12,5}
```

### 13. removeAll

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

***Syntax:***

```
removeAll(pred)
```

***Arguments:***

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

***Predicate Syntax:***

```
function(index, obj)
```

***Predicate Arguments:***

- ***obj***: the object at which the iteration arrived;
- ***index****:* the index at which the iteration arrived.

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

***Returns:*** A JSU.List sublist.

***Example:***

```
l1 = new JSU.List(1, 4, 3, 12, 5, 1, 2);

var pred = function(i, o) {
    return o > 5;
};
alert(l1.removeAll(pred)); // {1,4,3,5,1,2}
```

### 14. trueForAll

Checks if all the list's items meet a specified condition.

***Syntax:***

```
trueForAll(pred)
```

***Arguments:***

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

***Predicate Syntax:***

```
function(index, obj)
```

***Predicate Arguments:***

- ***obj***: the object at which the iteration arrived;
- ***index****:* the index at which the iteration arrived.

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

***Returns:*** A boolean.

***Example:***

```
l1 = new JSU.List(1, 4, 3, 5, 1, 2);

var pred = function(i, o) {
    return o < 5;
};
alert(l1.trueForAll(pred)); // false, 5 doesn't meet the condition
```

### 15. trueForOne

Checks if only one item of the list 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:***

```
l1 = new JSU.List(1, 4, 3, 5, 1, 2);
var pred = function(i, o) {
    return o >= 5;
};
alert(l1.trueForOne(pred)); // true
```

### 16. trueForAny

Checks if at least one item of the list 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:***

```
l1 = new JSU.List(1, 4, 3, 5, 1, 2);

var pred = function(i, o) {
    return o < 5;
};
alert(l1.trueForAny(pred)); // true
```

### 17. trueForN

Checks if exactly n items from the list meet a specified condition.

***Syntax:***

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

***Arguments:***

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

***Predicate Syntax:***

```
function(index, obj)
```

***Predicate Arguments:***

- ***obj***: the object at which the iteration arrived;
- ***index****:* the index at which the iteration arrived.

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

***Returns:*** A boolean.

***Example:***

```
l1 = new JSU.List(1, 4, 3, 5, 1, 2);

var pred = function(i, o) {
    return o > 4;
};
alert(l1.trueForN(1, pred)); // true
alert(l1.trueForN(0, pred)); // false
alert(l1.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:***

```
l1 = new JSU.List();
l1.resetComparers(); // default comparers will be used
l1.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.List object how to handle custom objects. Notice that you can add comparer methods only to instances of the JSU.List object, allowing you this way to have lists 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:***

```
l1 = new JSU.List();
var comp = function(a, b) {
    return a == b;
};
l1.addComparer(comp);
```

## I. How the Method Chaining works

The JSU.List 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 l = new JSU.List();

l.initWith(null, 3); // {null,null,null}
l.set(1, ""); // {null, "", null}
alert(l.has("")); // true
```

and with method chaining:

```
var l = new JSU.List().initWith(null, 3);

alert(l.set(1, "").has("")); // 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 l = new JSU.List();

alert(l.initWith(null, 3)
       .set(1, "")
       .has("")); // 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.List object after altering it depending on the called methods. So, in our last example, **initWith** method will return the new list object filled with nulls, and the return value being a JSU.List you can call another method on it - for example **set**, which sets the item at the index 1 to an empty space. The **set** method also returned the modified JSU.List object, so we call the **has** method to check whether it has the empty string. Note that calling the **has** method will break the chain because it returns a boolean value.

## J. Sorting lists using a custom comparing method

The List's sort method allows you to specify your own comparing method for comparing two objects from the list, 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 list 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 list with the timers
var l1 = new JSU.List(t1, t2, 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 list descending based on the timers' interval
l1.sort("DESC", comp); // l1 = {t2, t3, 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 list of timers with different intervals, and then sorts the list descending 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 list with the timers
var l1 = new JSU.List(t1, t2, t3);
```

I've created a new list containing all the timers. Notice that the list 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 list 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 list 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 list descending based on the timers' interval
l1.sort("DESC", comp); // l1 = {t2, t3, t1}
```

We sort the list 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 list 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.List module has two methods for checking the equality of two lists that store native objects only, but these methods will not work as expected when you add custom objects to a list, because it doesn't know how to treat them. This chapter will show you how to "*teach*" the list to compare custom objects, allowing you to use more than just equality methods on them and obtain the expected results.

### 1. Methods that use comparers

Here's a table of all the methods that use comparers. You can check what type of comparer each method uses, and if that method uses comparers directly, or indirectly by calling other methods based on comparers.

| Method | Uses.. | How ? |
|---|---|---|
| contains | exact comparers | direct |
| equals | normal comparers | direct |
| same | exact comparers | direct |
| indexOf | exact comparers | direct |
| indexesOf | exact comparers | direct |
| lastIndexOf | exact comparers | direct |
| count | exact comparers | direct |
| distinct | exact comparers | indirect |

## 2. 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 lists containing the sets
var l1 = new JSU.List(s1, s3),
    l2 = new JSU.List(s2, s3),
    l3 = new JSU.List(s1, s2);
```

This code creates three lists containing some JSU.Set objects (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
alert(l1.equals(l2)); // false
alert(l1.same(l2)); // false

alert(l1.equals(l3)); // false
alert(l1.same(l3)); // false

alert(l1.contains(s2)); // false
alert(l1.count(s2)); // 0
```

When running the above code, you can see that we got some wrong results (the fourth alert method actually shows the correct result, but this is just an exception). Let's dig a little bit through these alert calls:

- *l1.equals(l2)* - this should return true; if you look at the initialization code, you can see that s1 == s2, and s3 == s3;
- *l1.same(l2)* - this should also return true, because s1 === s2, and s3 === s3;
- *l1.equals(l3)* - this should return true because s1 == s1 and s3 == s2 (even if the object types are different, because the equals method uses the normal equality operator, not the strict one);
- *l1.same(l3)* - this should return false, and so it does; this happens because of the default exact comparers that have some logic behind them, but it's not enough;
- *l1.contains(s2)* - this should return true, because it contains s1 and s1 == s2;

- *l1.count(s2)* - this should return 1, for the same reason stated on the previous item.

Now, the comparers:

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

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

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 if both objects are sets and, if so, it will convert the 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 set's items and check for equality for each pair of 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 list's comparers:

```
// add the new comparers to the list calling
// the methods that use comparers
l1.addComparer(comparer);
l1.addExactComparer(exactComparer);
```

Beware that after these lines of code, the l2 list will be the same and only l1 will have the new comparers. Why ? Because in this case we'll only use l1's methods to check equality. If you know you'll also use l2's methods to compare l2 to other lists, you'll have to add those comparers to l2, too.

And now, after adding the new comparers, let's see what we get:

```
// correct results
alert(l1.equals(l2)); // true
alert(l1.same(l2)); // true

alert(l1.equals(l3)); // true
alert(l1.same(l3)); // false

alert(l1.contains(s2)); // true
alert(l1.count(s2)); // 1
```

We got the correct results, as expected!