ZINCKSOFT

Give The Dream New Life.

JSU.Set

The complete Module API Reference

Quick Introduction

The JSU Set Module (**JSU.Set / JSUSet**) provides a new JavaScript data type for storing everything you want, from native objects to custom objects, and even other sets, while allowing standard mathematical operations on them like union, intersection, checking for inclusion, extracting subsets, and so on. But excepting those new operations, the main difference between a JSU.List and a JSU.Set is that the set doesn't allow duplicate elements (elements that have the same value and also the same type - for custom objects you have to define comparers so that you get this functionality working correct).

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

JSU.Set API Reference

A. Constructor

Syntax:

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

Arguments:

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

Returns: The JSU.Set object.

Example:

```
var s1 = new JSU.Set(); // empty set
var s2 = new JSU.Set(1, [2, 3], "4", null, l1);
    // l2 is a set with a number, a two-element array,
    // a string, a null value and the empty set l1
var s3 = new JSU.Set(s2);
    // 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 set'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 set'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 set's elements.

C. Static Methods

1. fromArray

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

Syntax:

```
JSU.Set.fromArray(arr)
```

Arguments:

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

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

Example:

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

2. fromString

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

Syntax:

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

Arguments:

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

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

Example:

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

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

D. Manipulation Methods

1. clear (~ *empty*)

This method will empty a set.

Syntax:

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

Returns: The empty JSU.Set.

Example:

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

2. add

Adds objects to a set, similar to the JSU.Set's constructor method.

Syntax:

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

Arguments:

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

Returns: A JSU.Set.

Example:

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

3. set

Sets the value of an item from the set at a specified index. Notice that this method cannot be used to add new items, but only for modifying existing ones. Also, if you set a value that already exists, the method won't change anything.

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.Set.

Example:

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

4. remove

Removes one or more objects from the set.

Syntax:

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

Arguments:

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

Returns: A JSU.Set.

Example:

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

5. sort

Sorts the set using the QuickSort method for best performance. You can sort a set using your own method for comparing the set'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 set; 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 set, 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.Set.

Example:

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

For an example of how to sort a set 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 set.

Syntax:

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

Returns: A number.

Example:

```
l1 = new JSU.Set(1, 10, 100, 1, 2, 3);
alert(l1.size()); // outputs 5, one 1 is removed being a duplicate
```

2. get (~ *items*)

Gets the object at a specified position within the set.

Syntax:

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

Arguments:

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

Returns: An object.

Example:

```
l1 = new JSU.Set(1, 10, 100, 1, 2, 3);
alert(l1.get(3)); // outputs 2, 1 was removed being a duplicate
```

3. item

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

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 set object; if only the index is passed, it returns the object at that location.

Example:

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

4. contains (~ *has*)

Checks whether the set 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.Set(1, 10, 100, 1, 2, 3);
alert(l1.contains(1)); // true
alert(l1.contains(0)); // false
alert(l1.contains("1")); // false
```

5. equals / same

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

Syntax:

```
equals(set)
same(set)
```

Arguments:

- ***set***: the set 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.Set(1, 10, 100, 1, 2, 3);
l2 = new JSU.Set("1", 10, 100, 1, 2, 3);
l3 = new JSU.Set(1, 10, 100, 2, 3);
alert(l1.equals(l2)); // false, l2 has one extra item ("1")
alert(l1.equals(l3)); // true, one 1 from l1 being removed
alert(l1.same(l2)); // false
```

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

6. getTypeOf (~ *getTypeAt*)

This method gets the type of the object found at a specified position within the set. 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.Set("1", 10, new JSU.Set());
alert(l1.getTypeOf(1)); // number
alert(l1.getTypeOf(0)); // string
alert(l1.getTypeOf(2)); // jsuset
```

7. indexOf

Gets the index or indexes of a specified object.

Syntax:

```
indexOf(obj)
```

Arguments:

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

Returns: A number.

Note: this method uses the default exact comparer methods.

Example:

```
l1 = new JSU.Set(1, 2, 3, 1, 0, 1, 12);
alert(l1.indexOf(1)); // 0
```

8. isEmpty (~ *isVoid*)

Checks if the set is empty.

Syntax:

```
isEmpty()        alias: isVoid()
```

Returns: A boolean.

Example:

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

9. isTyped

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

Syntax:

```
isTyped()
```

Returns: A boolean.

Example:

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

F. Extraction Methods

1. getSubset

Extracts a subset of a specified size from the calling set starting at the given position, or extracts a subset of all elements that match a specified predicate.

Syntax:

```
getSubset(start [, count])
getSubset(predicate)
```

Arguments:

- ***start***: the position from where to start extracting;
- ***count***: the number of elements to extract;
- ***predicate***: the function used to check for matches.

Predicate Syntax:

```
function(index, object)
```

Predicate Arguments:

- ***object***: 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 new JSU.Set.

Example:

```
l1 = new JSU.Set(1, 12, 3, 1, 5, 1, 6);
    // l1 = {1, 12, 3, 5, 6}
var pred = function(i, o) {
    return o < 5;
}
alert(l1.getSubset(2, 1)); // {3}
alert(l1.getSubset(pred)); // {1, 3}
```

2. toArray / toString

These two methods convert the Set 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.Set(1, 12, 3, 1, 5, 1, 6);
alert(l1.toArray()); // [1,12,3,5,6]
alert(l1.toString()); // "1,12,3,5,6"
alert(l1.toString(";")); // "1;12;3;5;6"
```

G. Predicate-based Methods

exists

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

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.Set(1, 12, 3, 1, 5, 1, 6);

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

H. Mathematical Methods

1. intersect (~ *intersection*) / intersectWith (~ *intersectionWith*)

Intersects one or more sets with the calling set.

Syntax:

```
intersectWith(set)        alias: intersectionWith(set)
intersect(set1 [, set2 ...])  alias: intersection(set1 [, set2 ...])
```

Arguments:

- ***set, set1-N***: the sets to intersect with the calling set.

Returns: A JSU.Set.

Example:

```
var s1 = new JSU.Set(1, 2, 3, 4, 5),
    s2 = new JSU.Set(2, 3, 5, 6),
    s3 = new JSU.Set(2, 5, 7);
alert(s1.intersectWith(s2)); // s1 = {2, 3, 5}
alert(s1.intersect(s2, s3)); // s1 = {2, 5}
```

2. substract (~ *subtract*, *difference*) / substractFrom (~ *subtractFrom*, *differenceWith*)

Substracts one or more sets from the calling set.

Syntax:

```
substractFrom(set)          alias: subtractFrom(set)
                                   differenceWith(set)
substract(set1 [, set2 ...]) alias: subtract(set1 [, set2 ...])
                                    difference(set1 [, set2 ...])
```

Arguments:

- ***set, set1-N***: the sets to substract from the calling set.

Returns: A JSU.Set.

Example:

```
var s1 = new JSU.Set(1, 2, 3, 4, 5, 7),
    s2 = new JSU.Set(2, 3, 5, 6),
    s3 = new JSU.Set(2, 5, 7);
alert(s1.substractFrom(s2)); // s1 = {1, 4, 7}
alert(s1.substract(s2, s3)); // s1 = {1, 4}
```

3. union (~ *merge*) / unionWith (~ *mergeWith*)

Merges one or more sets with the calling set.

Syntax:

```
unionWith(set)        alias: mergeWith(set)
union(set1 [, set2 ...])     alias: merge(set1 [, set2 ...])
```

Arguments:

- ***set, set1-N***: the sets to merge with the calling set.

Returns: A JSU.Set.

Example:

```
var s1 = new JSU.Set(1, 2, 3, 4, 5, 7),
    s2 = new JSU.Set(2, 3, 5, 6),
    s3 = new JSU.Set(2, 5, 8);
alert(s1.unionWith(s2)); // s1 = {1, 2, 3, 4, 5, 7, 6}
alert(s1.union(s2, s3)); // s1 = {1, 2, 3, 4, 5, 7, 6, 8}
```

4. complement (~ *inverseDifference*) / complementWith (~ *inverseDifferenceWith*)

Substracts the calling set from one or more sets.

Syntax:

```
complementWith(set)       alias: inverseDifferenceWith(set)
complement(set1 [, set2 ...]) alias: inverseDifference(set1 [, ...])
```

Arguments:

- ***set, set1-N***: the sets substracted from the calling set.

Returns: A JSU.Set.

Example:

```
var s1 = new JSU.Set(1, 2, 3, 4, 5, 7),
    s2 = new JSU.Set(2, 3, 5, 6, 8, 9),
    s3 = new JSU.Set(2, 5, 8);
alert(s1.complementWith(s2)); // s1 = {6, 8, 9}
alert(s1.complement(s2, s3)); // s1 = {8}
```

5. xor / xorWith

The **xor** method is the opposite of the **intersect** method, so it returns a subset with all elements that each set has and that the others doesn't.

Syntax:

```
xorWith(set)
xor(set1 [, set2 ...])
```

Arguments:

- ***set, set1-N***: the sets to operate with.

Returns: A JSU.Set.

Example:

```
var s1 = new JSU.Set(1, 2, 3, 4, 5, 7),
    s2 = new JSU.Set(2, 3, 5, 6, 8, 9),
    s3 = new JSU.Set(2, 5, 8);
alert(s1.xorWith(s2)); // s1 = {1, 4, 7, 8, 9}
alert(s1.xor(s2, s3)); // s1 = {1, 3, 4, 7, 6, 8, 9}
```

6. isSubset (~ *isSubsetOf*, *includedIn*) / isSuperset (~ *isSupersetFor*, *includes*)

The **isSubset** method will check if the set is a subset of a specified set (called *the superset*), while the **isSuperset** method will check if the set is the superset of a specified set (called *the subset*).

Syntax:

```
isSubset(set)        alias: isSubsetOf(set), includedIn(set)
isSuperset(set)      alias: isSupersetFor(set), includes(set)
```

Arguments:

- ***set***: the set to check against.

Returns: A boolean.

Example:

```
var s1 = new JSU.Set(1, 2, 3),
    s2 = new JSU.Set(2, 3),
    s3 = new JSU.Set(2, 3, 4);
alert(s1.isSubsetOf(s2)); // false
alert(s2.isSubsetOf(s1)); // true
alert(s1.isSupersetFor(s2)); // true
alert(s2.isSupersetFor(s1)); // false
alert(s1.isSubsetOf(s3)); // false
```

I. 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.Set();
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.Set object how to handle custom objects. Notice that you can add comparer methods only to instances of the JSU.Set object, allowing you this way to have sets 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.Set();
var comp = function(a, b) {
    return a == b;
};
l1.addComparer(comp);
```

J. How the Method Chaining works

The JSU.Set 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.Set(1, 0, 3, 4);

l.set(1, 2); // {1, 2, 3, 4}
alert(l.has(2)); // true
```

and with method chaining:

```
var l = new JSU.Set(1, 0, 3, 4);

alert(l.set(1, 2).has(2)); // 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.Set(1, 0, 3, 4);

alert(l.set(1, 2)
       .has(2)); // 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.Set object after altering it depending on the called methods. So, in our last example, the **set** method will set the item at index 1 to 2, and then return the modified JSU.Set, allowing us to call more methods on it - for example, we call the **has** method to check whether it has the number 2. Note that calling the **has** method will break the chain because it returns a boolean value.

K. Sorting sets using a custom comparing method

The Set's sort method allows you to specify your own comparing method for comparing two objects from the set, 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 set 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 set with the timers
var l1 = new JSU.Set(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 set 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 set of timers with different intervals, and then sorts the set 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 set with the timers
var l1 = new JSU.Set(t1, t2, t3);
```

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

We sort the set 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.

L. Teaching the set 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.Set module has two methods for checking the equality of two sets that store native objects only, but these methods will not work as expected when you add custom objects to a set, because it doesn't know how to treat them. This chapter will show you how to "*teach*" the set 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
add	exact comparers	indirect
set	exact comparers	indirect
intersect / intersectWith	exact comparers	indirect
substract / substractFrom	exact comparers	indirect
union / unionWith	exact comparers	indirect
constructor	exact comparers	indirect
getSubset	exact comparers	indirect
remove	exact comparers	indirect
item	exact comparers	indirect
isSubset	exact comparers	indirect
isSuperset	exact comparers	indirect
xor / xorWith	exact comparers	indirect
complement / complementWth	exact comparers	indirect

2. How to do it

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

```
// create three lists
var l1 = new JSU.List(1, 2, 3),
    l2 = new JSU.List(1, 2, 3),
    l3 = new JSU.List("1", "2", "3");

// create sets containing the lists
var s1 = new JSU.Set(l1, l3),
    s2 = new JSU.Set(l2, l3),
    s3 = new JSU.Set(l1, l2);
```

This code creates three sets containing some JSU.List objects (for more information about sets, check the *JSU.List 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(s1.equals(s2)); // false
alert(s1.same(s2)); // false

alert(s1.equals(s3)); // false
alert(s1.same(s3)); // false

alert(s1.contains(l2)); // false
```

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:

- *s1.equals(s2)* - this should return true; if you look at the initialization code, you can see that l1 == l2, and l3 == l3;
- *s1.same(s2)* - this should also return true, because l1 === l2, and l3 === l3;
- *s1.equals(s3)* - this should return true because l1 == l1 and l3 == l2 (even if the object types are different, because the equals method uses the normal equality operator, not the strict one);
- *s1.same(s3)* - 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;
- *s1.contains(l2)* - this should return true, because it contains l1 and l1 == l2.

Now, the comparers:

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

// create the exact comparer
var exactComparer = function(a, b) {
    if (JSU.isJSUList(a) && JSU.isJSUList(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 lists and, if so, it will convert the lists 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 list'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 set's comparers:

```
// add the new comparers to the set calling
// the methods that use comparers
s1.addComparer(comparer);
s1.addExactComparer(exactComparer);
```

Beware that after these lines of code, the s2 set will be the same and only s1 will have the new comparers. Why ? Because in this case we'll only use s1's methods to check equality. If you know you'll also use s2's methods to compare s2 to other sets, you'll have to add those comparers to s2, too.

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

```
// correct results
alert(s1.equals(s2)); // true
alert(s1.same(s2)); // true

alert(s1.equals(s3)); // true
alert(s1.same(s3)); // false

alert(s1.contains(l2)); // true
```

We got the correct results, as expected!