UNDERSCORE.JS EXAMPLES

A QUICK GUIDE OF UNDERSCORE THROUGH EXAMPLES TO GET YOU STARTED WORKING WITH UNDERSCORE

WRITTEN BY: TORE AURSTAD, SYSTEM DEVELOPER FROM NORWAY, 2013

UNDERSCORE.JS

- UNDERSCORE.JS IS A JAVASCRIPT LIBRARY, WRITTEN PRIMARILY FOR ARRAY MANIPULATION BUT ALSO A LOT OF OTHER USAGES.
- THE VERSION TESTED WHEN WRITING THESE SLIDES IS 1.5.0 AND THIS VERSION CONTAINS 109 FUNCTIONS IN ALL.
- UNDERSCORE.JS TESTED HERE PRIMARILY WITH HTTP://JSBIN.COM IN MOZILLA FIREFOX, THIS IS THE OUTPUT YOU GET WHEN YOU ASK FOR THE FUNCTIONS IN UNDERSCORE:

console.log(_.functions(_));

["after", "all", "any", "bind", "bindAll", "chain", "clone", "collect", "compact", "compose", "contains", "countBy", "debounce", "defaults", "defer", "delay", "detect", "difference", "drop", "each", "escape", "every", "extend", "filter", "find", "findWhere", "first", "flatten", "foldl", "foldr", "forEach", "functions", "groupBy", "has", "head", "identity", "include", "indexOf", "initial", "inject", "intersection", "invert", "invoke", "isArguments", "isArray", "isBoolean", "isDate", "isElement", "isEmpty", "isEqual", "isFinite", "isFunction", "isNaN", "isNull", "isNumber", "isObject", "isRegExp", "isString", "isUndefined", "keys", "last", "lastIndexOf", "map", "max", "memoize", "methods", "min", "mixin", "noConflict", "object", "omit", "once", "pairs", "partial", "pick", "pluck", "random", "range", "reduce", "reduceRight", "reject", "rest", "result", "select", "shuffle", "size", "some", "sortAscending", "sortBy", "sortDescending", "sortedIndex", "tail", "take", "tap", "template", "throttle", "times", "toArray", "unescape", "union", "uniq", "unique", "uniqueId", "unzip", "values", "where", "without", "wrap", "zip"]

BACKGROUND INFORMATION

- WEBSITE OF UNDERSCORE.JS IS: http://underscorejs.org/
- SOURCE CODE IS AVAILABLE ON GITHUB HERE: https://github.com/jashkenas/underscore
- UNDERSCORE.JS WAS CREATED BY **JEREMY ASHKENAS** IN 2009.
- CREATOR IS A WORLD RENOWNED PRODUCTIVE DEVELOPER, WORKING IN THE INTERACTIVE DIVISON OF NEW YORK TIMES.
- HE IS ALSO THE CREATOR OF OPEN SOURCE PROJECTS COFFEESCRIPT AND BACKBONE.JS
- UNDERSCORE.JS IS UNDER CONTINOUS IMPROVEMENT AND THE CURRENT VERSION (JULY 2013) IS 1.5.0
- SOURCE CODE IS AVAILABLE BOTH IN DEVELOPER VERSION,

  MINIFIED VERSION AND ANNOTATED SOURCE CODE.

  CURRENT VERSION IS ONLY 1245 LINES LONG: https://github.com/jashkenas/underscore/blob/master/underscore.js

FIRST GLANCE AT UNDERSCORE.JS – ELEMENT OPERATORS

- Underscore is used in JS code by using the _-character. This is similar to how jQuery is exposed, with the $-character.
- Let's look at how we can use some element operators on arrays in JS next.
- We create a simple list of numbers to use when looking at element operators in Underscore.Js

First and Last element operators

- These two element operators return the first and the last item in an array in JS with these two functions in Underscore.JS. They are similar to First() and Last() in LINQ.
- var nums = [-1, 8, 8, 2, 9, 13, 16, 63];
- _(nums).first(); //get the first item: -1 Similar to First() in LINQ
- _(nums).last(); //get the last item: 63 Similar to Last() in LINQ
- Note: you can pass in a number to specify the n number of items to retrieve of the first and last items as an overload here

Map and Reduce

- Map is used to transform an array into another array. The array can be nested and contain object with properties of their own of arbitrary complexity.
- Reduce is used to aggregate a resulting scalar or object from an array, for example to generate an avergage or similar statistics.
- Let us look at map next how we can remap an array to include another property to create another array – i.e. do a transformation with underscore.js:

```
var stateRegion = [
    { State: 'South-Trondelag', Region: 'Central' },
    { State: 'Oslo', Region: 'Eastern' },
    { State: 'Hordaland', Region: 'West' },
    { State: 'Akershus', Region: 'Eastern' },
    { State: 'Oppland', Region: 'Eastern' }
   ];
```

Map continued

```
var getRegionOfState = function(state){
   return _.chain(stateRegion).where({ State: state}).pluck('Region').first().value();
}
var norwegianCities = [
  { City: 'Trondheim', State: 'South-Trondelag' },
  { City: 'Oslo', State: 'Oslo' },
  { City: 'Lillehammer', State: 'Oppland' },
  { City: 'Bergen', State: 'Hordaland'}];

var norwegianCitiesWithRegion = _.map(norwegianCities, function(nc){
   return { City: nc.City, State: nc.State, Region: getRegionOfState(nc.State) };
});

console.log(norwegianCitiesWithRegion);
```

Map continued (2)

- This mapped array results in the following array:
- We can see that the mapped array now not only contains cities and states, but also *Regions*, which was retrieved with the method in the previous sample.
- Note how we used the *chain* function to allow chaining ourself to the result like in LINQ. We also used where and pluck (which does a single property projection) and in the end last() to pick the single Region mapping. We do not care about error handling here for now, in case there is no reg… a state is mapped to.
- Let's look at *Reduce* next. Let's count the number of cities with Region *Eastern.*

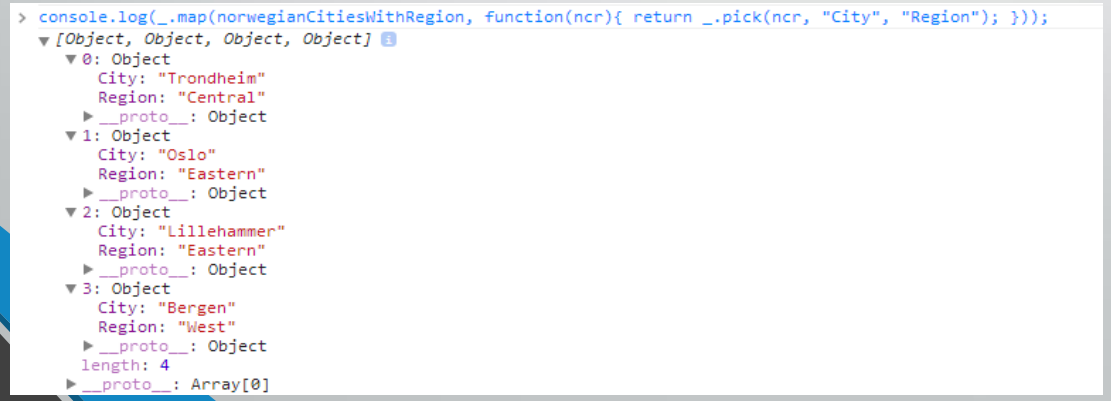
Reduce

```
var norwegianCitiesInEasterRegion = _.reduce(norwegianCitiesWithRegion,
function (memo, ncr) {
        return memo + ((ncr.Region == "Eastern") ? 1 : 0);
}, 0);
console.log(norwegianCitiesInEasterRegion);
```

- We get the answer of 2 cities in the Eastern region of Norway.
- Please note that in the example above, we use *Reduce* together with complex properties. To get a scalar into the memo parameter, it was necessary to set 0 as the third parameter of *Reduce* as the context parameter. For scalar arrays, this third parameter is not always required.

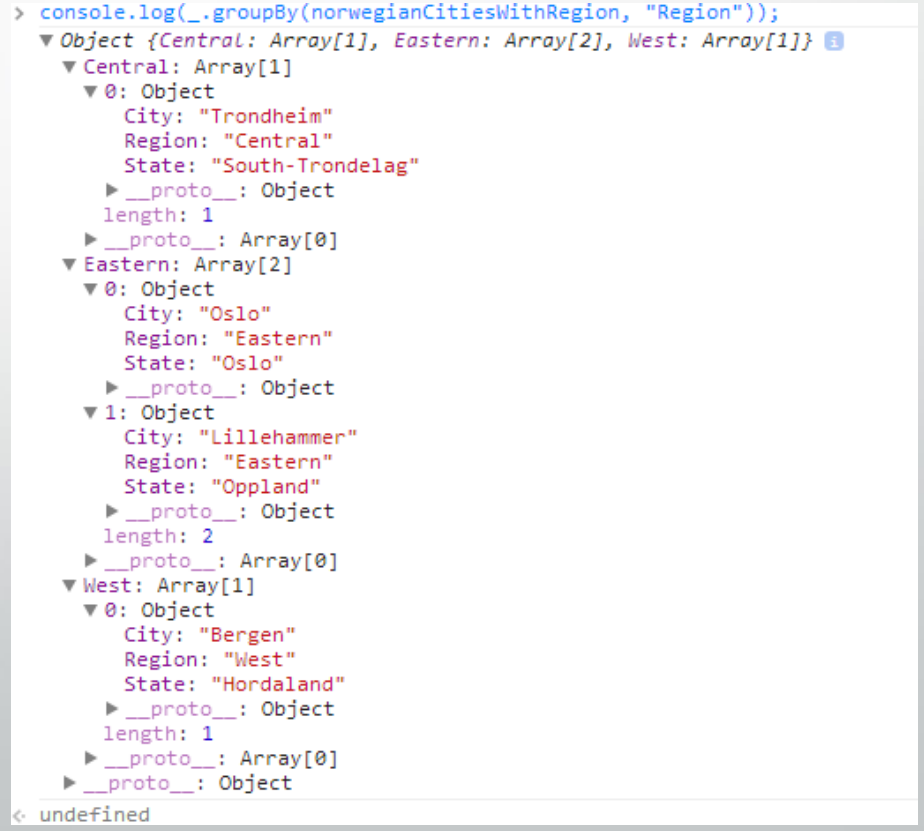
Projection operators

- Projection is available in Underscore.js with projection operators *pluck* and *pick*. The operator *pluck* projects a single property, and *pick* can project multiple properties. This is similar to the *Select* operator in LINQ.
- console.log(_.pluck(norwegianCitiesWithRegion, "City")); //pluck gives:
- ["Trondheim", "Oslo", "Lillehammer", "Bergen"]
- The *pick* operator works on an individual basis on items in arrays. This is how we can use it for example:
- console.log(_.map(norwegianCitiesWithRegion, function(ncr){ return _.pick(ncr, "City", "Region"); }));



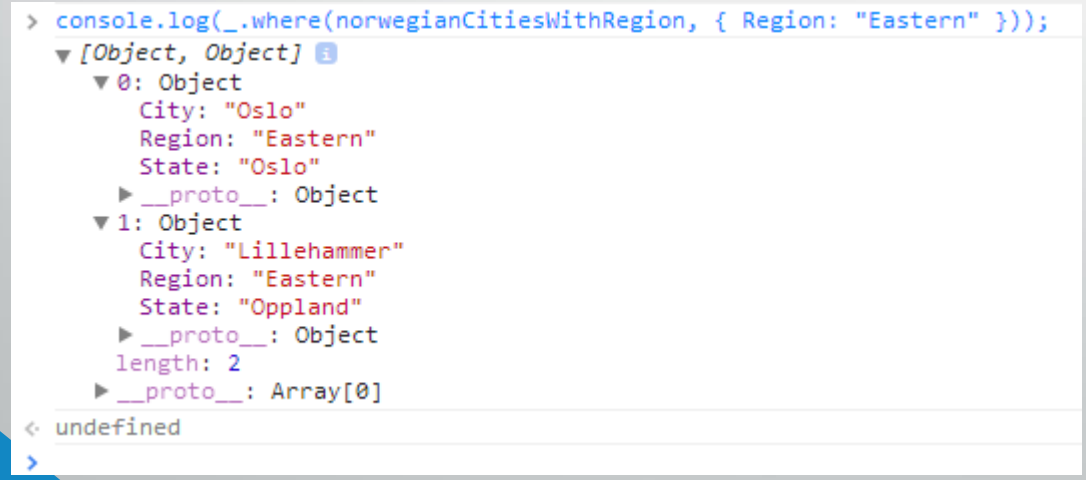
groupBy in Underscore.js

- Grouping is supported in Underscore.JS and is similar to GroupBy in LINQ.
- console.log(_.groupBy(norwegianCitiesWithRegion, "Region"));
- We group by a group key, usually a property of the item type in the array we group. We get arrays as the values. The following image shows the structure of the resulting grouped array in raw format:
- The array contains the objects that match the same grouping criteria.



Where operator in Underscore.js

- We can search for items in an array with a property of given value as in the following example:
- console.log(_.where(norwegianCitiesWithRegion, { Region: «Eastern» }));
- We get the two cities in our array matching the condition. We could have passed in multiple property conditions here.



countBy operator in Underscore.js

- CountBy is similar to GroupBy. It however just returns an object with properties set to the keys of the groups and the number of instances in each group key.

```
var cr = _.countBy(norwegianCitiesWithRegion, "Region");
```

- This object can then display its data for example in the following manner:

```
for(var c in cr){
console.log(c + ": " + cr[c]);
}
```

- This gives the following output of the number of cities in each region in our sample:

```
Central: 1
Eastern: 2
West: 1
```

range operator in Underscore.js

- Generates an array of provided values. Handy for generating numeric range array to iterate over.
- Similar to the Enumerable.Range operator in Linq, same logic too
- Signature of the operator / function is start, stop and optionally steplength
- Stop value is exclusive like in LINQ (up to, but not including). Start value is inclusive, just like LINQ. Step value is default set to 1.
- Examples:

- _.range(0, 5)
- [0, 1, 2, 3, 4] //Note stop value 5 is not inclusive here in the set

- _.range(0,10,2)
- [0, 2, 4, 6, 8] //Note stop value is not inclusive here either and that the step length is set to 2

sortBy operator

- Similar in function as OrderBy in LINQ
- Allows to sort an array. Specify which property to sort after.
- Possible to pass in sorting logic function also, or multiple properties to sort with. Simple example next of single property sorting:

```
> _.sortBy(norwegianCitiesWithRegion, "City");
[▼Object i , ▼Object i , ▼Object i , ▼Object i ]
    City: "Bergen"          City: "Lillehammer"     City: "Oslo"          City: "Trondheim"
    Region: "West"          Region: "Eastern"       Region: "Eastern"     Region: "Central"
    State: "Hordaland"      State: "Oppland"        State: "Oslo"         State: "South-Trondelag"
  ▶__proto__: Object      ▶__proto__: Object      ▶__proto__: Object    ▶__proto__: Object
>
```

- Descending sort on an array like in LINQ OrderByDescending can be done using the reverse() function on an array (built in JS function).
- _.sortBy(norwegianCitiesWithRegion, "City").reverse();

```
> _.sortBy(norwegianCitiesWithRegion, "City").reverse();
[▼Object i , ▼Object i , ▼Object i , ▼Object i ]
    City: "Trondheim"       City: "Oslo"          City: "Lillehammer"     City: "Bergen"
    Region: "Central"       Region: "Eastern"     Region: "Eastern"       Region: "West"
    State: "South-Trondelag" State: "Oslo"        State: "Oppland"        State: "Hordaland"
  ▶__proto__: Object      ▶__proto__: Object      ▶__proto__: Object    ▶__proto__: Object
>
```

Filter function: filter and Search function: contains

- The operator filter will return all items that passes a truth condition (iterator). Example:
- console.log(_.filter([1,3,11,9,12,21,29,28], function(n) { return (n % 3) === 0; }));
- Returns: [3, 9, 12, 21] //all numbers in the passed in array that is divisible by 3

- The contains operator returns true if the array contains the specified value.
- console.log("Contains 25?: " + _.contains([1,3,16,25,44], 25));
- "Contains 25?: true"

Object operator

- Different from Underscore.Js multiple array manipulation operators or functions, the object operator allows you to create objects with keys and values specified in two separate arrays, example next shows how to use this:
- var mergedObject = _.object(["Trondheim", "Oslo", "Lillehammer", "Bergen"], ["South-Trondelag", "Oslo", "Oppland", "Hordaland"])
- *Object {Trondheim: "South-Trondelag", Oslo: "Oslo", Lillehammer: "Oppland", Bergen: "Hordaland"}*
- The ordering in these two lists counts. The first item in the first list is paired with the first item in the second list, then moving on to the second item in both lists and so on.
- Underscore.Js has multiple utility functions and is therefore in many ways a general purpose utility library that fills in the missing gaps that pure Javascript can not fix

Set operators: Union and intersection

- We can use set operators in Underscore.js
- It is most useful with scalar arrays, but it can be used with complex arrays.
- Union returns a distinct list of elements of elements present in both arrays, which are passed to this operator
- Intersection returns a distinct list of elements which are present in both arrays , which are passed to this operator

- _.union([2,4,3,6,1],[7,8,2,4]);
- [2, 4, 3, 6, 1, 7, 8]

- _.intersection([2,4,3,6,1],[7,8,2,4]);
- [2, 4]

Specialized set operators: difference, without and uniq

- The operator difference is similar to the LINQ operator Except, it returns the values in the first array **not present** in the second array
- _.difference([2,4,3,6,1],[7,8,2,4]);
- [3, 6, 1]
- The operator without is similar to the operator difference, but the second part of the argument are values and not a single array to use for the exclusion.
- _.difference([2,4,3,6,1],7,8,2,4);
- [3, 6, 1] //same results as above, but that was because we used the same exclusion values and not an array

Special utility function - Once

- The special utility function once is one of Underscore.js many special but practical functions. This utility function will actually define a function that can be called only once. This is sometimes desired when initializing and wanting to avoid a function being possible to be called later on.
- Example next: (Anybody notice the reference from Allo Allo?)

- var onceFunc = _.once(function(){ console.log("Listen very carefully, I shall say this only once."); });
- onceFunc()
- Listen very carefully, I shall say this only once.
- onceFunc() //will give no output this second time (or any time later)

Special utility function: bind and bindAll

- The special utility functions bind and bindall is capable of binding a function's this pointer to a specified object of arbitrary complexity, such as a simple object or a complex object graph that is acting as a view model or similar uses. The difference between bind and bindAll is that bindAll allows specifiying multiple functions to be bound to a particular object.
- The following example shows an example how we can bind a function.

- var greeting = function(greeting) { return greeting + this.name; }
- greeting = _.bind(greeting, { name: "Leclerc", profession: "Stooge" });
- greeting("It is I, "); //gives the following output:
- "It is I, Leclerc"

- To use bindAll, just pass in multiple functions. We can of course define the object to bind against before calling bind, the example just did the bind inline as it is a simple example.

Special utility function: memoize

- With the memoize function, we can cache heavy calculations based upon arguments being passed in and returning a cached result, if it is not the first time the function is executed. This is convenient for heavy calculations that usually will not differ upon future executions, given the same arguments as a speed up technique.
- var fibonacci = _.memoize(function(n) { return n < 2 ? n : fibonacci(n - 1) + fibonacci(n - 2); })
- Calling fibonacci with the same argument for n, will return the same cached result instead of doing the calculation. *Memoizing* only happens with the exact same arguments here.

Special utility function: delay and defer

- The special utility function delay executes a function after specified amount of milliseconds. Example:
- _.delay(function(){ console.log("hi") }, 1000);
- «hi» //this executes after 1 second is passed

- The special utility function defer executes a function in a non-blocking fashion for the UI thread, waiting for the call stack to clear.

- _.defer(function(){ console.log(«This will execute when idle.»); }

Retrieving index: indexOf, lastIndexOf

- console.log("Item at: " + _.indexOf([1,2,3,5,11,17], 11));
- Returns: «Item at: 4», the index here is 0-based and returns the first index the search value, which was found
- console.log("Item at: " + _.lastIndexOf([1,2,3,5,11,11,17], 11));
- Returns: «Item at: 5», the index is 0-based and returns the last index of the search value, which was found.
- If the search value is not found, both methods here will return -1 to indicate that the item was not found

Object functions

- There are more object functions in Underscore.js! The functions keys and values can extract property names and values of properties of an object.

- _.keys({ "City": "Trondheim", "State": "South Trondelag" })
- ["City", "State"]
- _.values({ "City": "Trondheim", "State": "South Trondelag" })
- ["Trondheim", "South Trondelag"]

- The keys function can be used to find column headers and values will retrieve column values for each object, i.e. the row. In other words, an array of objects in Javascript can be displayed using these two methods in tabular form, usually with the _.each method or similar.

Object comparison: isEqual, isElement

- To do a complex equality comparison (not checking reference but that two objects have the same content), use isEqual

- console.log(_.isEqual([1,2,3,4,17], [1,2,3,4,17]));
- Returns: true

- This was a simple example, but we can also compare complex objects with arbitrary level of nesting

- To check if an object is a DOM element, use isElement:
- console.log("Object is DOM element: " + _.isElement(jQuery('body')[0]));
- Returns: «Object is DOM element: true»

Existence operators: some and every

- The some function is aliased to any and is similar to LINQ any in functionality.
- The every function is aliased to all and is similar to LINQ all in functionality.

- _.some([1,2,7,9,13,15], function(n) { return n % 2 == 0; });
- true
- _.every([1,2,7,9,13,15], function(n) { return n % 2 == 0; });
- false

- As we can see, some returns true because one element is an even number and therefore is true for the comparison function (iterator).
- The every operator returns false, since not all elements satisfy the comparison function (iterator).

Extending Underscore.js: mixin

- We can add custom functions into Underscore.js if we desire.

- ```
  _.mixin({ capitalize : function(string) {
   return string.charAt(0).toUpperCase() + string.substring(1).toLowerCase();
  } });
  _("fabio").capitalize();
  ```
- "Fabio"

- Here we define a new operator capitalize inside the object literal where we capitalize the first letter. Then we test out with the string «fabio» and we get the resulting string «Fabio».
- This is similar to what we see in jQuery extend method.
  ```

Defining a pipeline with: chain

- We have already seen chain in action:

```
var stateRegion = [
      { State: 'South-Trondelag', Region: 'Central' },
      { State: 'Oslo', Region: 'Eastern' },
      { State: 'Hordaland', Region: 'West' },
      { State: 'Akershus', Region: 'Eastern' },
      { State: 'Oppland', Region: 'Eastern' }
     ];
var getRegionOfState = function(state){
        return _.chain(stateRegion).where({ State: state}).pluck('Region').first().value();
     }
```

- To repeat – with chain we can set up a chain of arbitrary complexity where we chain together multiple array operators or other methods and in the end we must call the *value()* operator to retrieve the result.
- This is where we see Underscore.js being very much similar to LINQ by allowing chaining. Together with mixin we can tailor our Javascript experience much into LINQ-like syntax.

Random numbers: random

- Generates a random number between specified numbers. If only one number is passed, a random number between 0 and the provided number is generated.

- console.log(_.random(0,100));
- Returns random number between 0 and 100.
- The min and max values here are inclusive.

Iterating over an array: each

- The each operator iterates over an array an performs a given iterator function. Example:
- _.each([1,2,3], function(n){ console.log(n*n); });
- Outputs:
- 1
- 4
- 9

- We can of course define the array before running the each operator / function.
- Aliased to forEach

Special comparison operators: isFinite, isDate isRegExp

- console.log(_.isFinite(Infinity));
- Returns false
- console.log(_.isFinite(102));
- Returns true
- console.log(_.isRegExp(/moe/));
- Returns true: This can be done to validate a regular expression that is being created for correct syntax
- console.log(_.isDate(new Date()));
- Returns true
- console.log(_.isDate("12. september 2013"));
- Returns false: wrong syntax – this operator can validate a Date for correct syntax

Special comparison operators: isArray, isFunction, isString

- console.log(_.isArray([1,2,3,4,11]));
- Returns true – the object is an array

- console.log(_.isFunction(window.alert));
- Returns true – the object is a function

- console.log(_.isString('some string'));
- Returns true: this is a string

Summary

- Underscore.js has many different uses.
- It is impressive that the source code is only about 1250 lines of code for the current version (1.5.0) and still has so many different uses
- This shows how powerful Javascript can be and how compact it is
- The library is primarily a utility library. It is similar to a Swiss-army knife, there are several uses, primarily focused on array manipulation.
- We haven't even covered the template support yet and to discover more, check out the annotated source code of Underscore.js.

That's all folks!! You might also want to check out these JS frameworks: