cull.js

Cull is a toolbelt for writing functional javascript. It is based on these core values:

- **Pure JavaScript** - no programming in strings
- **Pure functions** - prefer side-effect free functions wherever possible
- **No chaining** - chaining obscures side-effects and invites spaghetti code
- **No wrapping** - work with native arrays and DOM elements
- **Fail early** - complains loudly about API misuse in development

This is totally a work in progress. The API may change until we reach the big one-oh.

A consistent API

In an effort to create a consistent API, here are some basic function parameter guidelines:

Function first, collection second

The readability of

```
map(name, filter(underage, persons))
```

over

```
map(filter(persons, underage), name)
```

tells us which order to put the parameters. It is hard to spot what `name` belongs to, dangling at the end. This gets harder with more nested functions.

You might argue that when you inline the functions, the second form looks better. For maximum visual pleasure, don't inline your functions.

Function list

- isList `(list)`
- toList `(value)`
- doall `(fn, list)`
- isFunction `(fn)`
- reduce `(fn, initial, items)`
- all `(pred, list)`

- some `(pred, list)`
- onlySome `(pred, list)`
- trim `(string)`
- identity `(arg)`
- defined `(o)`
- unary `(fn)`
- prop `(name)`
- func `(name, args)`
- eq `(x)`
- compose `(fns, thisp)`
- callWith `()`
- partial `(fn)`
- bind `(obj, callee)`
- flatten `(list)`
- indexOf `(needle, list)`
- uniq `(list)`
- first `(fn, list)`
- select `(fn, list)`
- difference `(list, other)`
- keys `(object)`
- values `(object)`
- map `(fn, list)`
- negate `(pred)`
- reject `(pred, list)`
- concat `(list1, list2)`
- partition `(n, list)`
- mapdef `(fn, list)`
- mapcat `(fn, list)`
- interpose `(sep, list)`
- after `(obj, name, fn)`
- before `(obj, name, fn)`
- around `(obj, name, fn)`

Documentation and examples

isList `(list)`

Is `list` an object with a numeric length, but not a DOM element?

```
assert(cull.isList([]));
refute(cull.isList({}));
assert(cull.isList({ length: 4 }));
refute(cull.isList({ length: 4, tagName: "DIV" }));
```

toList `(value)`

Returns a version of `value` that is an actual Array.

```
assert.equals(cull.toList(1), [1]);
assert.equals(cull.toList(null), []);
assert.equals(cull.toList(undefined), []);

var args = function () { return arguments; };
assert.isArray(cull.toList(args(1, 2, 3)));
```

doall `(fn, list)`

Calls `fn` on every item in `list`, presumably for side-effects.

```
var result = [];
cull.doall(function (item) {
    result.unshift(square(item));
}, [1, 2, 3, 4]);
assert.equals(result, [16, 9, 4, 1]);
```

isFunction `(fn)`

Is `fn` a function?

```
assert(cull.isFunction(function () {}));
assert(cull.isFunction(square));
refute(cull.isFunction({}));
```

reduce `(fn, initial, items)`

Returns the result of applying `fn` to `initial` and the first item in `list`, then applying `fn` to that result and the 2nd item, etc.

Can also be called without `initial`, in which case the first invocation of `fn` will be with the first two items in `list`.

```
var list = [1, 2, 3, 4];
var add = function (a, b) { return a + b; };
assert.equals(cull.reduce(add, list), 10);
assert.equals(cull.reduce(add, 5, list), 15);
```

all `(pred, list)`

Is `pred` truthy for all items in `list`?

```
refute(cull.all(isEven, [1, 2, 3, 4]));
assert(cull.all(isEven, [2, 4, 6, 8]));
```

some `(pred, list)`

Is `pred` truthy for any items in `list`?

```
assert(cull.some(isEven, [1, 2, 3, 4]));
refute(cull.some(isEven, [1, 3, 5, 7]));
```

onlySome `(pred, list)`

Is `pred` truthy for at least one item in `list`, and also falsy for at least one item in `list`?

```
assert(cull.onlySome(isEven, [1, 2, 3, 4]));
refute(cull.onlySome(isEven, [2, 4, 6, 8]));
refute(cull.onlySome(isEven, [1, 3, 5, 7]));
```

trim `(string)`

Returns `string` with white space at either end removed.

```
assert.equals(cull.trim("  abc  "), "abc");
assert.equals(cull.trim("  abc  def  "), "abc  def");
```

identity `(arg)`

Returns `arg` unchanged.

```
assert.equals(cull.identity(4), 4);
```

defined `(o)`

Is `o` neither undefined nor null?

```
assert(cull.defined({}));
refute(cull.defined(null));
refute(cull.defined(undefined));
```

unary `(fn)`

Returns a version of `fn` that only accepts one argument.

```
var add = function (a, b) { return a + b; };
assert.isNaN(cull.unary(add)(2, 3));
```

prop `(name)`

Returns a function that takes one argument and returns its `name`-property.

```
assert.equals(cull.prop("id")({ id: 42 }), 42);
```

func `(name, args)`

Returns a function that takes one argument and calls its `name`-function with `args` (optional).

```
var f = cull.func("getId");
var obj = { getId: function () { return 42; } };
assert.equals(42, f(obj));
```

eq `(x)`

Returns a function that takes one argument and returns true if it is equal to `x`.

```
var isFive = cull.eq(5);
assert(isFive(5));
refute(isFive("5"));
```

compose `(fns, thisp)`

Returns a function that calls the last function in `fns`, then calls the second to last function in `fns` with the result of the first, and so on, with an optional this-binding in `thisp`.

```
var identity = cull.compose();
assert.equals(identity(2, 3), 2);
```

callWith `()`

Takes any number of arguments, and returns a function that takes one function and calls it with the arguments.

```
var add = function (a, b) { return a + b; };
var fn = cull.callWith(1, 2);
assert.equals(fn(add), 3);
```

partial `(fn)`

Takes a function `fn` and any number of additional arguments, fewer than the normal arguments to `fn`, and returns a function. When called, the returned function calls `fn` with the given arguments first and then additional args.

```
var fn = function (a, b) { return a + b; };
var curried = cull.partial(fn, 3);
assert.equals(curried(5), 8);
```

bind `(obj, callee)`

Returns a function that calls `callee` with `obj` as this. `callee` can be a function, or it can be a string - in which case it will be used to look up a method on `obj`.

Optionally takes additional arguments that are partially applied.

```
var func = this.spy();
var obj = {};
var bound = cull.bind(obj, func);

bound();
assert.equals(func.thisValues[0], obj);

bound.call({});
assert.equals(func.thisValues[1], obj);

bound.apply({});
assert.equals(func.thisValues[2], obj);
```

flatten `(list)`

Flatten `list` recursively and return a list of non-list values

```
assert.equals(cull.flatten([1, 2, 3, 4]), [1, 2, 3, 4]);
```

indexOf `(needle, list)`

Return the first index of `needle` in `list`, otherwise < 0

```
assert.equals(1, cull.indexOf("b", ["a", "b", "c"]));
```

uniq `(list)`

Return a list with only the unique values in `list`

```
assert.equals(cull.uniq([1, 2, 3, 4]), [1, 2, 3, 4]);
```

first `(fn, list)`

Return the first item in `list` for which `fn` returns `true`

```
var items = [1, 2, 3, 4];
var even = function (i) { return i % 2 === 0; };
assert.equals(cull.first(even, items), 2);
```

select `(fn, list)`

Return a new list containing the items from `list` for which `fn` is `true`

```
var items = [0, 1, 2, null, 3, 4, undefined, 5, 6];
var result = cull.select(function (i) { return !!i; }, items);
assert.equals(result, [1, 2, 3, 4, 5, 6]);
```

difference `(list, other)`

Return a list of properties present in `list` but not in `other`

```
var result = cull.difference([1, 2, 3, 4], [2, 3]);
assert.equals(result, [1, 4]);
```

keys `(object)`

Return a list of enumerable own property keys in `object`

```
assert.equals(cull.keys({
    id: 1,
    num: 42,
    name: "Mr"
}), ["id", "num", "name"]);
```

values `(object)`

Return a list of enumerable own property values in `object`

```
assert.equals(cull.values({
    id: 1,
```

```
    num: 42,
    name: "Mr"
}), [1, 42, "Mr"]);
```

map `(fn, list)`

Returns a new list consisting of the result of applying `fn` to the items in `list`.

```
var square = function (num) { return num * num; };
assert.equals(cull.map(square, [1, 2, 3]), [1, 4, 9]);
```

negate `(pred)`

Returns the complement of `pred`, ie a function that returns true when `pred` would be falsy, and false when `pred` would be truthy.

```
var isOdd = cull.negate(isEven);
assert(isOdd(5));
```

reject `(pred, list)`

Returns a new list of the items in `list` for which `pred` returns nil.

```
var items = [1, 2, 3, 4, 5];
var odd = function (n) { return n % 2; };
assert.equals(cull.reject(odd, items), [2, 4]);
```

concat `(list1, list2)`

Returns a new list with the concatenation of the elements in `list1` and `list2`.

```
var a = [1, 2, 3];
var b = [4, 5, 6];
assert.equals(cull.concat(a, b), [1, 2, 3, 4, 5, 6]);
```

partition `(n, list)`

Returns a new list with the items in `list` grouped into `n-` sized sublists.

The last group may contain less than `n` items.

```
var n = 2;
var result = cull.partition(n, [1, 2]);
assert.equals(result, [[1, 2]]);
```

mapdef `(fn, list)`

Returns a new list consisting of the result of applying `fn` to the items in `list`, but filtering out all null or undefined values from both `list` and the resulting list.

```
this.list = [
    { id: 1 },
    { id: 2 },
    { different: false },
    { id: 3 }
];
```

mapcat `(fn, list)`

Returns the result of applying concat to the result of applying map to `fn` and `list`. Thus function `fn` should return a collection.

```
var dbl = function (single) { return [single, single]; };
assert.equals(
    cull.mapcat(dbl, [1, 2, 3]),
    [1, 1, 2, 2, 3, 3]
);
```

interpose `(sep, list)`

Returns a new list of all elements in `list` separated by `sep`.

```
var result = cull.interpose(":", [1, 2, 3]);
assert.equals(result, [1, ":", 2, ":", 3]);
```

after `(obj, name, fn)`

Advices the method `name` on `obj`, calling `fn` after the method is called. `fn` is called with the return value of the method as its first argument, then the methods original arguments. If `fn` returns anything, it will override the return value of the method.

```
cull.after(this.obj, "fn", function (ret, x) {
    this.list.push(ret * x);
});

this.obj.fn(3, 2);

assert.equals(this.obj.list, [3, 3]);
```

before `(obj, name, fn)`

Advices the method `name` on `obj`, calling `fn` before the method is called. `fn` is called with the same arguments as the method.

```
setupAdvice.call(this);
cull.before(this.obj, "fn", function (x) {
    this.list.push(x - 1);
});
```

around `(obj, name, fn)`

Advices the method `name` on `obj`, calling `fn` instead of the method. `fn` receives the original method as its first argument, and then the methods original arguments. It is up to the advicing function if and how the original method is called.

```
cull.around(this.obj, "fn", function () {});
this.obj.fn(3);
assert.equals(this.obj.list, []);
```

License