

Divergence Improved

Spencer Tipping

August 31, 2010

Contents

1	Introduction	2
2	Instances	3
2.1	Return value of <code>new</code>	3
2.2	Roles	4
3	Building Functions	6
3.1	Numbers	6
3.1.1	Large positive integers	6

Chapter 1

Introduction

The original Divergence¹ has some shortcomings. For one thing, it puts a bunch of methods into the global prototype namespace (probably its biggest problem). This used to break jQuery, and now breaks jQuery UI's tab and accordion components.² Another problem is that macro definitions are permanent, unstructured, and collision-prone.

This rewrite of Divergence solves both problems. The only object placed into the global namespace is called `divergence`, and all customization is done to it or a copy of it. Macro definitions are scoped to a particular instance of Divergence; they are not global by default.

¹<http://github.com/spencertipping/divergence>

²Not that it has to in theory; someone made the assumption that all array methods would be `dontEnum`, which isn't the case if you add stuff. The workaround is to use `hasOwnProperty`, or more importantly, not to use `for...in` on arrays.

Chapter 2

Instances

Unlike before, Divergence isn't just one function. You can create a new instance of Divergence with its own configuration, which can be useful for isolated regions of code that require a particularly common pattern, unit tests, etc. The most common way to do this is to use `new`:

Listing 2.1 `examples/instance-new.js`

```
1 new divergence (function (d) {
2   // Code in here can access d, which is a copy of the global divergence.
3   // To create a copy of d:
4   new d (function (new_d) {
5     // new_d is a copy of d, and will inherit any d-specific customizations
6     // specified earlier.
7   });
8
9   // Another way to do it:
10  d.clone (function (new_d) {
11    // This is exactly the same as above, except that its return value is
12    // intact.
13  });
14
15  // To grab the copy for later:
16  var new_d = d.clone();
17 });
```

2.1 Return value of `new`

Because `new` always returns a hash, not a function, using the `new divergence(f)` constructor won't return either `f`'s return value, nor will it return the new Divergence. Instead, it returns an object containing both. So, for example:

Listing 2.2 examples/instance-new-return.js

```
1 var result = new divergence (function (d) {
2   d.foo = 'bar';
3   return 5;
4 });
5 result.result           // => 5
6 result.divergence.foo   // => 'bar'
```

If you care about the return value of your function, it's probably easier to use `divergence.clone`:

Listing 2.3 examples/instance-clone-return.js

```
1 var result = divergence.clone (function (d) {
2   d.foo = 'bar';
3   return 5;
4 });
5 result           // => 5
```

In this case there is no way to access the scoped `d`, though you can return it explicitly if you want to hang on to it.

2.2 Roles

Sometimes you want to keep a set of customizations around for reuse. You can do this by creating a *role*, which is simply a function that modifies a Divergence instance. For example, this role adds an `assert` method to `d`:

Listing 2.4 examples/instance-role-assert.js

```
1 divergence.role.create ('assert', function (d) {
2   d.assert = function (what, message) {
3     if (! what) throw new Error ('Assertion failed: ' + message);
4     return what;
5   };
6 });
7
8 d.assert           // => undefined
```

Roles are attached to whichever Divergence instance they were created on. You can now use that role:

Listing 2.5 examples/instance-role-use.js

```
1 new divergence (function (d) {
2   d.role.use ('assert');           // Adds 'assert' to d in-place
3   d.assert (3 === 3, 'basic math'); // => true
4 });
5
```

```

6 new divergence.using ('assert', function (d) {
7   // d is a clone of divergence, but also with 'assert'
8   d.assert (true, 'should pass');          // => true
9 });
10
11 divergence.role.use ('assert');             // Not a great idea; see next paragraph
12 divergence.assert (1, 'truthy 1');         // => 1

```

Roles can't be "un-used", so generally the best approach is to add a role to a copy of your divergence function.

Chapter 3

Building Functions

Just like the original Divergence, this version is all about building functions. Also just like the old version, it specifies conversions to promote any built-in data type into a function, and lets you use your own data types if they provide `.fn()` methods.

3.1 Numbers

Numbers are now much more expressive. Just like before, 0, 1, 2, 3, and 4 map to the first five positional parameters. However, there are some new cases:

3.1.1 Large positive integers

Integers larger than 4 are converted into hexadecimal and interpreted, where each digit is a command in a stack-based language. The stack's initial contents are the positional parameters, where `arguments[0]` is at the top and `arguments[arguments.length - 1]` is at the bottom. Digits are interpreted from left-to-right; so, for example, the number `0xab` is interpreted as the command `a` followed by the command `b`. The following commands are understood:

- 5 Swap the top two stack entries. ("`5`" looks kind of like "`S`", which stands for Swap.)
- 6 Drop the top stack entry. ("`6`" is a backwards "`d`", which stands for Drop.)
- 7 If the next digit is a 0, 1, 2, 3, or 4, then push that digit as a number onto the stack. Otherwise, push the stack depth onto the stack and process the next digit normally.
- 8 Duplicate the top stack entry. ("`8`" looks like two "`0`"s.)
- 9 Drop the second entry. ("`9`" is "`6`" upside-down, and `6` drops the top entry.)

- a Add the two arguments on the top of the stack, and push the result. This also works on strings. If the top of the stack is an array, then push a new array consisting of the stack top concatenated with the second stack element; that is, `stack[0].concat([stack[1]])`.
- b Subtract `stack[1]` from `stack[0]`, pop both, and push the result. If either argument is non-numeric, then this operator applies `||` to the top two stack entries instead; that is, `stack[0] || stack[1]`.
- c Pop twice, multiply, and push. If either argument is non-numeric, then this operator applies `&&` to the top two stack entries instead; that is, `stack[0] && stack[1]`.
- d Pop twice, divide, and push. Operands are ordered the same way as they are for subtraction. If either argument is non-numeric, then this operator dereferences the stack top by the stack second instead of performing division; that is, `stack[0][stack[1]]`. If the stack top is undefined or null, then the second argument is dropped silently instead of being used for dereferencing.
- e Negate the top stack entry if it's a number. If it's not a number, then apply logical negation.
- f Invoke the top stack entry on the next one, and return the result. If the top of the stack isn't a function, then the current `d()` (that is, the one being used to convert this number to a function in the first place) is used to convert the stack top to a function first.

The digits 0-4, of course, push those positional parameters onto the top of the stack. For example 0 pushes arguments[0], 1 pushes arguments[1], etc.

Here are some examples:

Listing 3.1 examples/large-integer-functions.js

```

1 d(0xa)    // => function (x, y) {return x + y}           (if numeric)
2 d(0xa)    // => function (x, y) {return x.concat([y])}    (if x is an array)
3 d(0xb)    // => function (x, y) {return x - y}           (if numeric)
4 d(0xb)    // => function (x, y) {return x || y}           (if non-numeric)
5 d(0xc)    // => function (x, y) {return x * y}           (if numeric)
6 d(0xc)    // => function (x, y) {return x && y}          (if non-numeric)
7 d(0xd)    // => function (x, y) {return x / y}           (if numeric)
8 d(0xd)    // => function (x, y) {return x[y]}            (if x is non-numeric)
9
10 d(0x8a)   // => function (x)    {return x + x}           (if numeric)
11 d(0x8aa)  // => function (x, y) {return x + x + y}       (if numeric)
12
13 d(0x65b)  // => function (x, y, z) {return z - y}        (if numeric)
14 d(0xdd)   // => function (x, y, z) {return x[y][z]}      (if non-numeric)
15 d(0xdd)   // => function (x, y, z) {return (x / y) / z}  (if numeric)

```



```

16
17 d(0x88cc) // => function (x) {return x * x * x}           (if numeric)
18 d(0xee)   // => function (x) {return !!x}                 (if non-numeric)
19 d(0x7a)   // => function (x) {return x + 1}               (if numeric)
20 d(0x74a)  // => function (x) {return x + 4}               (if numeric)
21 d(0x748cc) // => function (x) {return x * 16}             (if numeric)
22 d(0x2f1f) // => function (f, x, y) {return f(y)(x)}

```

To be portable, you should use at most seven hex digits. Some browsers have integer math that can change the sign if the 32-bit is set.

- Positive floating-point numbers — [sec:building-functions-positive-floating-point-numbers](#)