Divergence Improved

Spencer Tipping

August 31, 2010

Contents

1 Introduction **2**

2 Instances **3**
2.1 Return value of `new` . . . 3
2.2 Roles . . . 4

3 Building Functions **6**
3.1 Numbers . . . 6
3.1.1 Large integers . . . 6

Chapter 1

Introduction

The original Divergence[1] 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`

[2] Not that is 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`

```
new divergence (function (d) {
  // Code in here can access d, which is a copy of the global divergence.
  // To create a copy of d:
  new d (function (new_d) {
    // new_d is a copy of d, and will inherit any d-specific customizations
    // specified earlier.
  });

  // Another way to do it:
  d.clone (function (new_d) {
    // This is exactly the same as above, except that its return value is
    // intact.
  });

  // To grab the copy for later:
  var new_d = d.clone();
});
```

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

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

```
var result = divergence.clone (function (d) {
  d.foo = 'bar';
  return 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

```
divergence.role.create ('assert', function (d) {
  d.assert = function (what, message) {
    if (! what) throw new Error ('Assertion failed: ' + message);
    return what;
  };
});

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

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

```

```
new divergence ('assert', function (d) {
  // d is a clone of divergence, but also with 'assert'
  d.assert (true, 'should pass');         // => true
});

divergence.role.use ('assert');           // Not a great idea; see next paragraph
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 integers

Integers larger than 5 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:

`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.

Because the digits 5-9 aren't used otherwise, they are also commands:

- `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` Rotate the top three stack entries – that is, the top two are moved down, and the third is moved to the top. ("7" looks kind of like a knight's jump in chess, which is two down and one over. You can think of the two down as shifts, and the one over as a pull.)
- `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.)

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. This solves the "leading-zero" problem – you will never need a leading zero, since `arguments[0]` begins at the stack top.

Here are some examples:

Listing 3.1 `examples/number-functions.js`

```
d(0xa)    // => function (x, y) {return x + y}         (if numeric)
d(0xa)    // => function (x, y) {return x.concat([y])}  (if x is an array)
d(0xb)    // => function (x, y) {return x - y}         (if numeric)
d(0xb)    // => function (x, y) {return x || y}        (if non-numeric)
d(0xc)    // => function (x, y) {return x * y}         (if numeric)
d(0xc)    // => function (x, y) {return x && y}        (if non-numeric)
d(0xd)    // => function (x, y) {return x / y}         (if numeric)
d(0xd)    // => function (x, y) {return x[y]}          (if x is non-numeric)

d(0x8a)   // => function (x)    {return x + x}         (if numeric)
d(0x8aa)  // => function (x, y) {return x + x + y}     (if numeric)

d(0x65b)  // => function (x, y, z) {return z - y}      (if numeric)
d(0xdd)   // => function (x, y, z) {return x[y][z]}    (if non-numeric)
d(0xdd)   // => function (x, y, z) {return (x / y) / z} (if numeric)
```