# layerCompose

Related work
https://straits.github.io/introduction/

https://github.com/traitsjs/traits.js
https://github.com/traitsjs/traits.js#stateful-traits

**Data Encapsulation**
Moving fast is one of JavaScripts main advantages as an enterprise language. And it is, let's be honest, a lot easier to find oneself in trouble when moving fast. JavaScript is notorious.

There's room for improvement in any case, and this proposal (and your challenge Ramses, shall you accept it) is to begin writing a JS library that would be a cornerstone of Harbr's frontend and ML development.

Although the origins of this proposal are rooted in the need for strong data encapsulation while maintaining easy access to the data, the solution is much more versatile. It resembles Scala's `traits`.

**Layering**

The one feature that is unique to `layerCompose` (as far as the author knows) is "Layering". While composition is something familiar, "Layering" is not inheritance. For example, with inheritance a parent class might call a function as defined/implemented/overridden by a child class. That is strictly forbidden with "Layering". Each layer can only know of the layers underneath and even that is strongly discouraged.

Eg.: {a: f(), b: f()} + {c: f()} = {a,b,c}
(in cases of overlapping method lists, different overriding strategies are available, both manual and automatic)

Layering promotes testability by constricting the pathways by which outside code can influence a layer. In what other ways does `layerCompose` help us build enterprise software?

**Goals**

- Compatibility with regular JS objects

Plain JS objects are used everywhere for passing bits of data. While we want to extend the functionality of a piece of data (eg, restricting writes) we don't want `JSON.stringify()` to return anything different than it would from plain data.

- Safety

  While it's convenient to pass a data object around, keeping it free from inappropriate write access is hard. We want to keep the convenience, and remove the danger. We do this by allowing one, and only one layer to edit a property on an object -- full read access across all layers and the external environment, but only a single layer can write to a particular property (reminiscent of Redux)

- Simplicity

  Shouldn't break your brain to use it and to reason about your code. In fact, it should make it easier to reason about your code.

- Performance

  `layerCompose` will see use in Machine Learning. This is what makes the construction of this library tricky; eg. JS Proxies could be quite handy to achieve our goals, but the performance could only be beared in development and not in production.

- Return On Investment

  This is the most important metric. The library should be extremely simple and light weight to promote maintainability (thus reducing costs). It must be widely applicable and save significant development effort (thus producing profit). Additionally, anything that doesn't grow dies so new features should be added at a slow but steady pace, such as automatic TypeScript definition generators.

**Sample code**

Code is worth a 1000 words. Below is a prototype that should bring clarity to how `layerCompose` could be implemented. Suggested to read from the bottom.

```
const isFunction = function (obj) {
  return !!(obj && obj.constructor && obj.call && obj.apply)
}

function _composeChain(chain, composition = []) {
  // const layers = []
  const methods = chain.reduceRight((acc, layerSpec, i) => {
    const layer = _compose(layerSpec, composition)
    composition.unshift(layer)
    // layers[i] = layer
    return Object.assign(acc, layer)
```

```
  }, {})

  // override functionality
  Object.keys(methods).forEach(name => methods[name].override = function () {
    throw new Error('Not Implemented')
  })

  return methods
}

function _composeServices(services, composition) {
  Object.keys(services).forEach(key => {
    if (services[key]._spec) services[key] = _compose(services[key], composition)
  })
  return services
}

function _compose(spec, composition = []) {
  if (Array.isArray(spec)) {
    return _composeChain(spec, composition)
  } else if (isFunction(spec)) {
    if (spec._spec) {
      return _compose(spec._spec, composition)
    } else {
      return spec(composition[0])
    }
  } else {
    return _composeServices(spec, composition)
  }
}

function _bind(methodsAndServices, contents) {
  const ms = methodsAndServices
  Object.keys(ms).forEach(key => {
    if (ms[key].bind) {
      ms[key] = ms[key].bind(contents)
    } else {
      _bind(ms[key], contents)
    }
  })
}

function layerCompose(spec) {
  const methods = _compose(spec)

  const composed = (contents) => {
    _bind(methods, contents)

    // proxy the contents for write protection
```

```
    // return Object.setPrototypeOf(contents, Object.create(methods))
    return Object.setPrototypeOf(methods, contents)
  }
  composed._spec = spec

  return composed
}


const A = layerCompose([{
  multiply: function () {
    return this.x * this.y
  }
}, {
  divide: function () {
    return this.x / this.y
  }
}])

const a = A({x: 1, y: 2})
console.log("Simple layering", a.x, a.y, "|", a.multiply(), a.divide())


const B = layerCompose([{
  getRemainder() {
    return this.x % this.y
  }
},
  A
])

const b = B({x: 10, y: 3})
console.log("Extension", b.x, b.y, "|", b.multiply(), b.divide(), b.getRemainder())


const C = layerCompose([(services) => {

  // just for fun
  let runCount = 0

  return {
    remainderRatio() {
      runCount++
      return services.B.getRemainder() / services.B.multiply() * runCount
    }
  }
},
  {B}
])
```

```
const c = C({x: 10, y: 3})
console.log("Extension", c.x, c.y, "|", c.remainderRatio(), c.remainderRatio())

// logs
// Simple layering 1 2 | 2 0.5
// Extension 10 3 | 30 3.3333333333333335 1
// Extension 10 3 | 0.03333333333333333 0.06666666666666667
```