Julien Moumné Software Engineer, Paris, France

Building Trees using a JavaScript DSL

An instructional sequence to a hack-free internal DSL for building trees in JavaScript

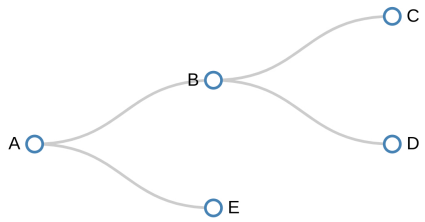
30 Jul 2016

In vanilla JavaScript, this is how you build a tree[1][2]:

JS Bin Save | HTML | CSS | ES6 / Babel | Console | Output | Help

ES6 / Babel ▾

```
// literally
let tree = {
  value: "A",
  forest: [{
      value: "E",
      forest: []
  }]
}

// programmatically
let b = {value: "B", forest: []}
let c = {value: "C", forest: []}
let d = {value: "D", forest: []}
b.forest.push(c)
b.forest.push(d)
tree.forest.unshift(b)

display(tree)
```



Auto-run JS ☑ Run with JS

The article describes a way to build trees using an internal DSL:

```
tree('A', () => {
  tree('B', () => {
    tree('C')
    tree('D')
  })
  tree('E')
  // add nodes from a dynamically created array
  moreNodes.forEach(e => tree(e))
})
```

This syntax mirrors the recursive structure of trees, reduces the noise and provides the ability to mixin arbitrary code.

This makes it convenient to:

- specify hierarchical configuration data as in Mocha or Hotshell,
- provide test data for algorithms on tree-like structures

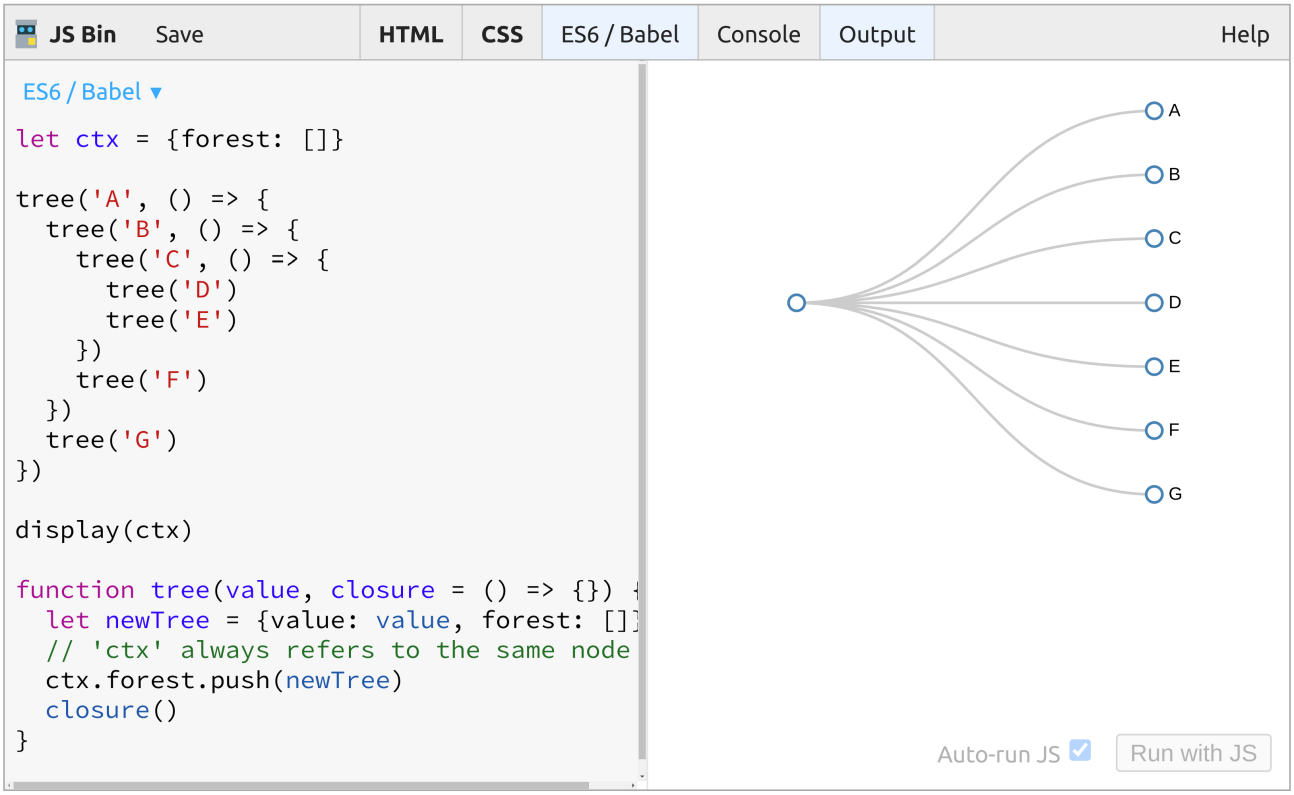
Concepts found in this article are neither new nor restrained to JavaScript[3].

This article aims to provide an instructional sequence to a hack-free solution in JavaScript.

Readers can jump to the final solution or step through this sequence:

Naive Solution

In this solution nodes are all children of the same parent node.

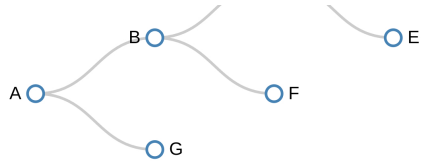


This helps us understand the key challenge, keeping track of the node under construction so children are inserted at the right location.

Forwarding a bare variable

The most straightforward solution is to explicitly pass along the parent node.

```
  // forwarded to each 'tree()' call
  tree(ctx, 'B', ctx => {
    tree(ctx, 'C', ctx => {
      tree(ctx, 'D')
      tree(ctx, 'E')
    })
    tree(ctx, 'F')
  })
  tree(ctx, 'G')
})

display(ctx.forest[0])

function tree(ctx, value, closure = () =>
  let newTree = {value: value, forest: []}
  ctx.forest.push(newTree)

  // 'closure()' is called with 'newTree'
  // and becomes the new parent node
  closure(newTree)
}
```



Auto-run JS Run with JS

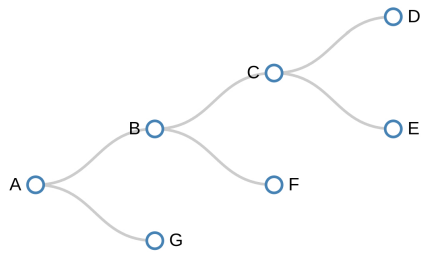
Coupling data and operations OOP style

An alternative solution in par with OOP promotes the global `tree()` function to an instance method.

JS Bin Save | HTML | CSS | ES6 / Babel | Console | Output | Help

ES6 / Babel ▾

```
let tree = {forest: [], add: add}

tree.add('A', tree => {
  tree.add('B', tree => {
    tree.add('C', tree => {
      tree.add('D')
      tree.add('E')
    })
    tree.add('F')
  })
  tree.add('G')
})

display(tree.forest[0])

function add(value, closure = () => {}) {
  let newTree = {
    value: value,
    forest: [],
    add: add
  }
  // 'this' refers to the parent node
  this.forest.push(newTree)

  // 'closure()' is called with
  // the new parent node 'newTree'
  closure(newTree)
}
```



Auto-run JS Run with JS

This approach is used by jbuilder. See jbuilder.js.

Because we have only one operation available on the tree we can remove the need to prefix calls to

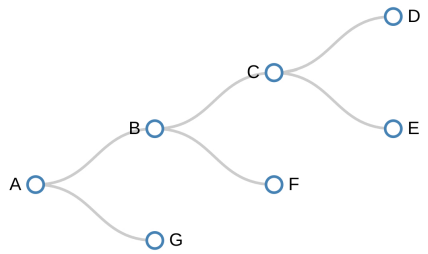
`add()`.

This is done by programmatically binding `this` using bind().

JS Bin Save | HTML | CSS | ES6 / Babel | Console | Output | Help

ES6 / Babel ▾

```
let forest = []

tree('A', tree => {
  tree('B', tree => {
    tree('C', tree => {
      tree('D')
      tree('E')
    })
    tree('F')
  })
  tree('G')
})

display(forest[0])

function tree(value, closure = () => {}) {
  let newTree = {value: value, forest: []}
  // 'this' refers to the parent node
  this.forest.push(newTree)

  // the 'tree' function is cloned and
  // the new parent node 'newTree'
  // is bound to 'this' using 'bind'
  closure(tree.bind(newTree))
}
```



Auto-run JS ☑ Run with JS

This is closer to the DSL we are looking for but we still have to pass around a parameter in each closure signature.

Implicit context using `this`

One approach to achieve parameterless closures is to use the implicit formal parameter this.

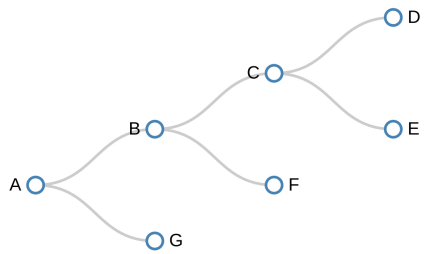
Forwarding a bare variable using `this`

Solution 'Forwarding a bare variable' can be adapted by programmatically binding `this` using apply() and forwarding it the same way `ctx` was being forwarded.

JS Bin Save | HTML | CSS | ES6 / Babel | Console | Output | Help

ES6 / Babel ▾

```
let ctx = {forest: []}

tree(ctx, 'A', function () {
  tree(this, 'B', function () {
    tree(this, 'C', function () {
      tree(this, 'D')
      tree(this, 'E')
    })
    tree(this, 'F')
  })
  tree(this, 'G')
})
```



```
display(ctx.forest[0])

function tree(ctx, value, closure = () =>
  let newTree = {value: value, forest: []}
  ctx.forest.push(newTree)

  // call 'closure()' and assign
  // the new parent node 'newTree' to 'thi
  closure.apply(newTree)
}
```

Auto-run JS Run with JS

In this case and the one that follows, setting `this` dynamically does not work with arrow functions. We must revert to standard function definitions `function () { body }`.

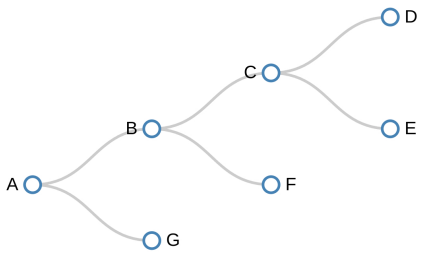
Coupling data and operations OOP style using `this`

Solution 'Coupling data and operations OOP style' can be rewritten the same way.

JS Bin Save | HTML | CSS | ES6 / Babel | Console | Output | Help

ES6 / Babel ▾

```
let forest = []

tree('A', function () {
  this.tree('B', function () {
    this.tree('C', function () {
      this.tree('D')
      this.tree('E')
    })
    this.tree('F')
  })
  this.tree('G')
})

display(forest[0])

function tree(value, closure = () => {}) {
  let newTree = {
    value: value,
    forest: [],
    // tree as a shared instance method
    tree: tree
  }
  // 'this' refers to the parent node
  this.forest.push(newTree)

  // call the closure and assign
  // the new parent node 'newTree' to 'th
  closure.apply(newTree)
}
```



Auto-run JS Run with JS

Coming from other OOP languages one could hope to directly call `tree()` without prefixing it with `this` thus achieving our target DSL.

In JavaScript, without hacks, it is mandatory to explicitly reference instance members using `this`.

Hacking a way through the forest

Solution 'Coupling data and operations OOP style using `this`' can be hacked to remove the need to

prefix calls to `tree()` with `this`.

The scope chain must be altered using with().

Usage of this method is usually not recommended. Whether it is to be considered bad practice or not, we will see that this solution presents serious limitations.

Ideally, we would apply `with()` this way:

```
function tree(value, closure = () => {}) {
  let newTree = {
    value: value,
    forest: [],
    tree: tree
  }
  this.forest.push(newTree)

  // in this block of code every property is first looked-up in 'newTree'
  with(newTree){
    // an example of a directly accessed property
    console.log(value)
    // call the closure wishing calls to function 'tree()' will resolve to 'newTree.tree
()'
    closure()
  };
}
```

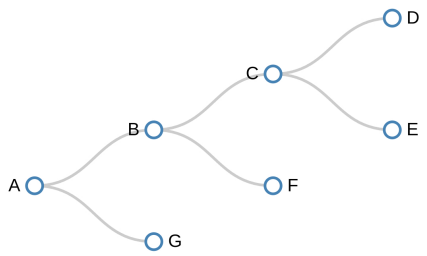
Calling the closure will not work as expected, properties will not be resolved on `newTree`. This is because JavaScript's closures are lexically scoped and `newTree` was not lexically present when the closure was defined.

We must reset the lexical scope by re-interpreting the closure using eval().

JS Bin Save | HTML | CSS | ES6 / Babel | Console | Output | Help

ES6 / Babel ▾

```
let forest = []

tree('A', () => {
  tree('B', () => {
    tree('C', () => {
      tree('D')
      tree('E')
    })
    tree('F')
  })
  tree('G')
})

display(forest[0])

function tree(value, closure = () => {}) {
  let newTree = {
    value: value,
    forest: [],
    tree: tree
  }
  this.forest.push(newTree)

  // this hack brings 'newTree' into the
  // lexical scope of 'closure' using 'eva
  // and provides a shortcut to its proper
  // using 'with'
```



```
  // using 'with'
  eval('with(newTree){(' + closure + ')()}')
}
```

Auto-run JS Run with JS

The DSL finally looks like what we had in mind.

However, there are two issues:

- functions called in the closure do not benefit from this hack, directly calling `tree()` within these functions will not work

```
tree('A', () => {
  twoNodesFail()
  twoNodes(newTree)
})

// does not work, nodes are added at the top of the tree as 'tree()' refers to the global function
function twoNodesFail() {
  tree('1')
  tree('2')
}

// works, but obscure, 'newTree' comes from 'with(newTree){...}'
function twoNodes(newTree) {
  newTree.tree('1')
  newTree.tree('2')
}
```

- the original lexical scope is lost, this breaks the principle of least astonishment

```
tree('A', () => {
  var commonChild = 'D'
  tree('B', () => {
    tree(commonChild) // ReferenceError: commonChild is not defined
  })
  tree('C', () => {
    tree(commonChild) // ReferenceError: commonChild is not defined
  })
})
```

This is how Hotshell was initially coded.

Forgo traditional calling convention

The trick to a hack-free solution lies in understanding how parameters are handled during a function call.

In stack-oriented programming languages, parameters are stored in the call stack and their life cycle is regulated by a calling convention.

We use solution 'Forwarding a bare variable' to illustrate the standard three step protocol.

```
function tree(ctx, value, closure = () => {}) {
  let newTree = {value: value, forest: []}
  ctx.forest.push(newTree)

  // 1. Push step
  //    Before calling 'closure()', a new call frame is pushed onto the stack and
```

```
  //    a reference of the local variable 'newTree' is copied in the parameter section of
the call frame
  closure(newTree) // 2. Peek step
                   //    Every reference to 'newTree' in the body of 'closure()' is resol
ved
                   //    by accessing the parameter section of the top of the call stack
  // 3. Pop step
  //    When 'closure()' returns, the parameter section of the call stack is discarded
  //    by removing the call frame created at step 1
}
```

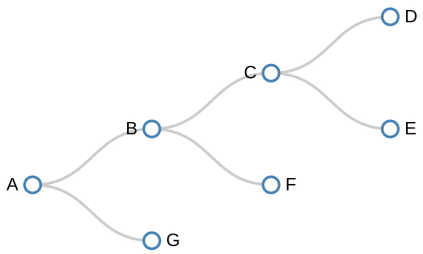
We can achieve the same result by using an explicit parameter stack and our own calling convention.

```
let stack = [{forest: []}]

function tree(ctx, value, closure = () => {}) {
  let newTree = {value: value, forest: []}

  // peak of stack holds the parent node
  stack.peek().forest.push(newTree)

  // simulates pushing the value to the argument section of the stack
  stack.push(newTree)

  // 'tree()' calls performed in 'closure()' will access the parent node
  // at the peak of the stack
  closure()

  // simulates popping the value from the argument section of the stack
  // to restore the argument to its previous value
  stack.pop()
}
```

JS Bin Save | HTML | CSS | ES6 / Babel | Console | Output | Help

ES6 / Babel ▾

```
let stack = [{forest: []}]

tree('A', () => {
  tree('B', () => {
    tree('C', () => {
      tree('D')
      tree('E')
    })
    tree('F')
  })
  tree('G')
})

display(stack.peek().forest[0])

function tree(value, closure = () => {}) {
  let newTree = {value: value, forest: []}
  stack.peek().forest.push(newTree)
  stack.push(newTree)
  closure()
  stack.pop()
}
```



Auto-run JS ☑ Run with JS

This approach is used in Mocha BDD interface to allow nested `describe()` calls. See bdd.js.

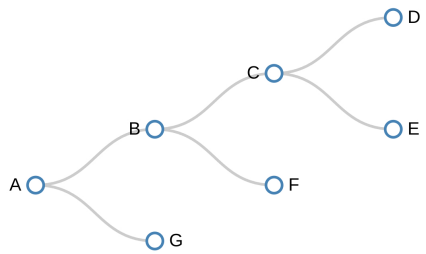
Reuse the existing stack

In a final simplification step we avoid allocating an explicit stack by using the local variable section of the existing stack.

JS Bin Save | HTML | CSS | ES6 / Babel | Console | Output | Help

ES6 / Babel ▾

```
let peak = {forest: []}

tree('A', () => {
  tree('B', () => {
    tree('C', () => {
      tree('D')
      tree('E')
    })
    tree('F')
  })
  tree('G')
})

display(peak.forest[0])

function tree(value, closure = () => {}) {
  let newTree = {value: value, forest: []}
  peak.forest.push(newTree)

  // keep a reference of the current parent
  // in the local variable section of the
  let prev = peak

  // update/push new parent node on top of
  peak = newTree

  closure()
  // restore/pop parent node so subsequent
  // 'tree()' calls operate on the right n
  peak = prev
}
```



Auto-run JS ☑ Run with JS

This approach is used in Groovy NodeBuilder and in Hotshell. See BuilderSupport.java and dslrunner.js.

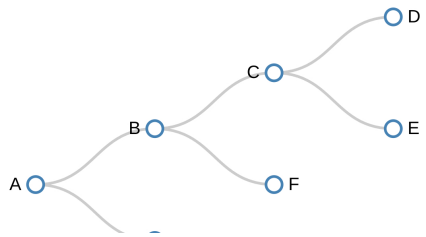
Package the solution in a JavaScript module

The same solution can be coded as a module to provide namespace control and isolation of state.

JS Bin Save | HTML | CSS | ES6 / Babel | Console | Output | Help

ES6 / Babel ▾

```
let tree = seed()

tree('A', () => {
  tree('B', () => {
    tree('C', () => {
      tree('D')
      tree('E')
    })
```



```
    tree('F')
  })
  tree('G')
})

display(tree.peek())

function seed() {
  let peak = {forest: []}
  function tree(value, closure = () => {}) {
    let newTree = {value: value, forest: [
    peak.forest.push(newTree)
    let prev = peak
    peak = newTree
    closure()
    peak = prev
  }
  tree.peek = () => peak.forest[0]
  return tree
}
```

Auto-run JS

Run with JS

Thoughts on internal DSLs

I find internal DSLs convenient to interact with tools I need on a daily basis.

They provide legibility as well as flexibility by being able to mixin arbitrary code.

Here are some examples to illustrate the versatility of this approach:

- Project builders (Gradle, Sbt)
- Testing frameworks (RSpec, Mocha)
- Virtual environment tools (Vagrant)
- Package Managers (Homebrew)

With the availability of embeddable interpreters I believe this approach can be generalized further.

This is the approach used in Hotshell by using Otto, an embeddable JavaScript interpreter for Go.

---

1. Property `forest` could have been named `trees`, `subtrees` or `children`. `Forest` is used in this article to echoe the mutually recursive definition found in Mutual recursion - Wikipedia. ↩
2. Tree visualization code found at https://bl.ocks.org/mbostock/4339184 ↩
3. NodeBuilder is one implementation in Groovy. Mocha is one in JavaScript. ↩