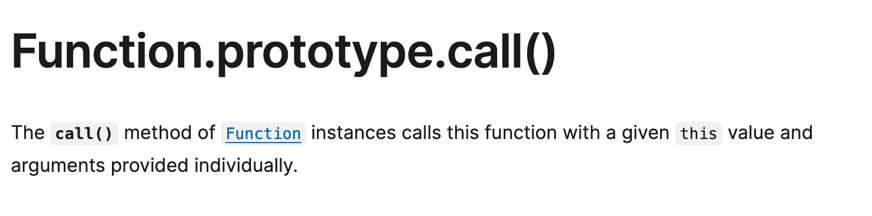
functionBind

1. We (mitch) had the basic idea – we can either bind methods with identical names from one object to another (providing the correct context for “this”
   1. Or we can add the method to the passed in object
   2. And return a function which returns the evaluated result of invoking the method that was defined on the context object.



1. rockPaperScissors
2. function rockPaperScissors(num, roundArr = [], currRound = 1, output = []) {
3. *if* (num <= 0) *return*;
4. const throws = ['rock', 'paper', 'scissors'];
5. *for* (let i = 0; i < throws.length; i++) {
6. roundArr.push(throws[i])
7. *if* (currRound === num) {
8. output.push(roundArr.slice())
9. } *else* {
10. rockPaperScissors(num, roundArr, currRound + 1, output)
11. }
12. roundArr.pop()
13. }
14. *return* output;
15. }

* To sort of summarize what this is happening, we’re first on our call stack 1 for rps(2). Then we take the first el of the possible throws.
* We use a “roundArr” as a temporary cache which can be pushed into the output array.
* As we iterate through the throws, we make a recursive call after pushing the first element.
* This stores the first element in the cache and allows us to add to it from within the recursive call.
  + Even though the second call pushes a new stack frame onto the call stack and opens a new execution context,
  + We can still access the same temporary cache because the cache is passed by reference.
* We make a recursive call, increasing the “round” number so that we can keep track of the number of throws.
* We make as many recursive calls as there are “rounds,” and each time, we add the first element of the throw to the cache before iterating.
* And we push the cache into the output before popping off the last element.

mergeSort

* Big picture strategy – we recursively divide the array
* And then we use a helper function to merge the array halves based on comparisons.

*// below is a helper function which merges two sorted arrays.*

*// "left" is a sorted array on the "Left" side and "right" is a sorted array on the 'right" side.*

function merge(left, right) {

const output = []

let i = j = 0 *// two pointers for i and j*

*while* (i < left.length && j < right.length) {

*if* (left[i] <= right[j]) {

output.push(left[i])

i++;

} *else* {

output.push(right[j]);

j++;

}

}

*// once this array is iterated through, we can assume that the rest of the values in the other are higher.*

*return* [...output, ...left.slice[i], ...right.slice[j]]

}

function mergeSort(array) {

*// base case:*

*if* (array.length <= 1) {

*return* array;

}

const mid = Math.floor(array.length / 2)

const left = array.slice(0, mid)

const right = array.slice(mid)

*// recursive case:*

*return* merge(mergeSort(left), mergeSort(right))

}