Recursion I

Overview

```
/*
  - Definition of recursion
  - The call stack
  - countdown example
  - factorial example
  - Recursion and arrays, strings
  - Tips for approaching recursion problems
*/
```

Definition

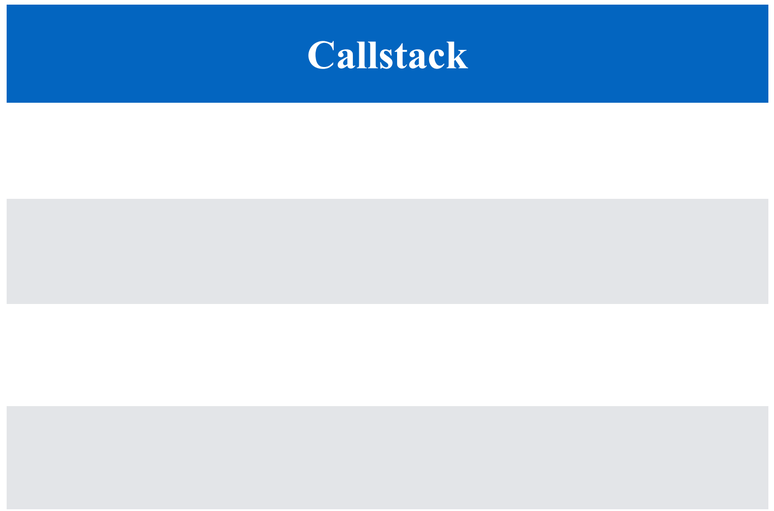
```
/* recursion occurs when a function calls itself!  */



/* recursion is an alternative to iteration (using a loop) */



/* in the real world, you may see recursion instead of iteration when a
   recursive solution is:
     - easier to reason about (recursion helps break big problems into
       small chunks)
     - easier to read than an iterative solution
     - won't negatively affect performance too much (recursion can be
       a memory hog)
Machine Learning
AI
Traversing Data Structures
*/
```

The call stack

```
1  /* before we talk about recursion, we have to talk about the call stack */
2
3
4
5  /* JS is "single threaded" - can only run one function at a time */
6
7
8
9  /* the call stack is the structure JS uses to figure out which function
10   it should be running at any point in time */
11
12
13
14
```

The call stack

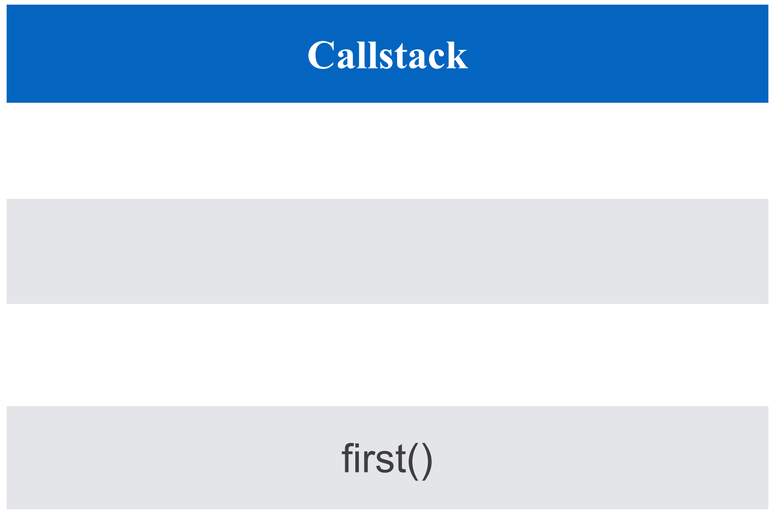
```
/* whenever we call a function, it's added to the top of the call stack */



/* JS will execute whatever function is on the top of the stack */



function first() {
  console.log('I am first!');
}



function second() {
  console.log('I am second!');
}
```



I am first

The call stack

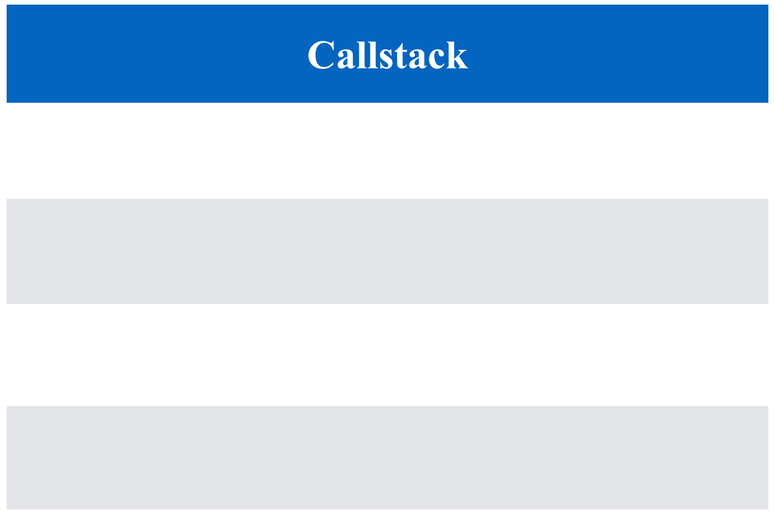
```
1   /* whenever we call a function, it's added to the top of the call stack */
2
3
4
5   /* JS will execute whatever function is on the top of the stack */
6
7
8
9   function first() {
10    console.log('I am first!');
11  }
12
13
14  function second() {
      console.log('I am second!');
    }
```



I am first

The call stack

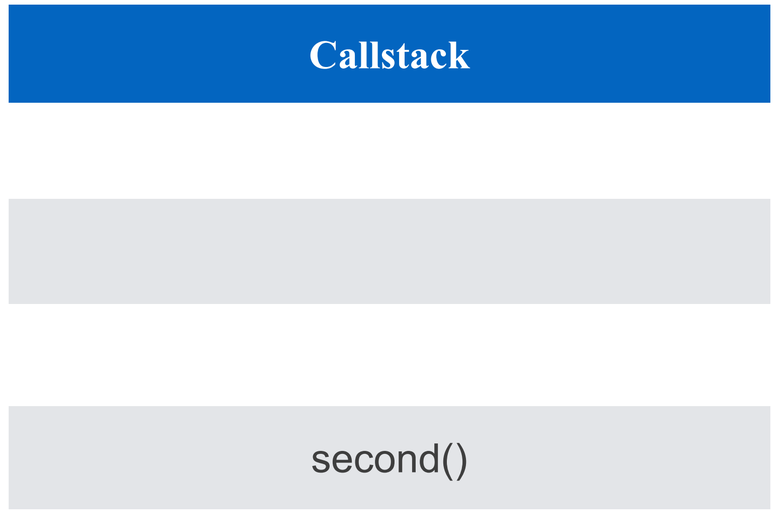
```
/* whenever we call a function, it's added to the top of the call stack */



/* JS will execute whatever function is on the top of the stack */



function first() {
  console.log('I am first!');
}



function second() {
  console.log('I am second!');
}
```



I am first
I am second

The call stack

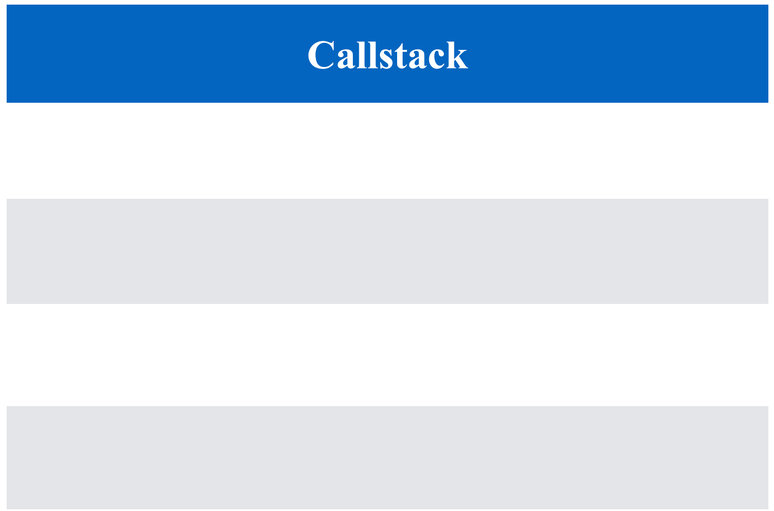
```
/* whenever we call a function, it's added to the top of the call stack */



/* JS will execute whatever function is on the top of the stack */



function first() {
  console.log('I am first!');
}



function second() {
  console.log('I am second!');
}
```



I am first
I am second

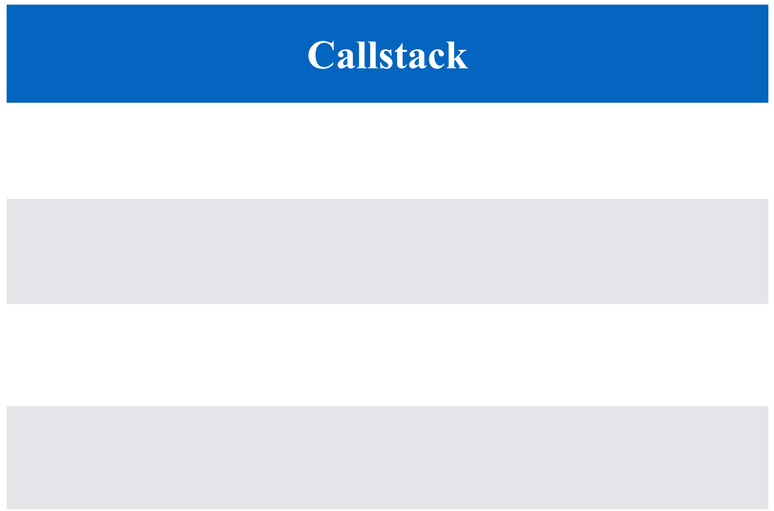
The call stack

```
/* whenever we call a function, it's added to the top of the call stack */



/* JS will execute whatever function is on the top of the stack */



function first() {
  console.log('I am first!');
}



function second() {
  console.log('I am second!');
}
```



The call stack

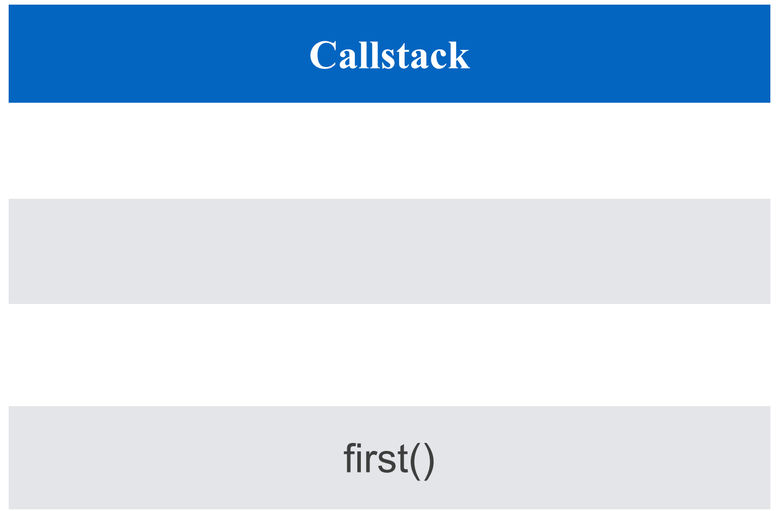
```
1   function first() {
2     console.log('I am first!');
3     second();
4     console.log('First is finished');
5   }
6
7
8
9   function second() {
10    console.log('I am second!');
11  }
12
13
14  first();
```



I am first

The call stack

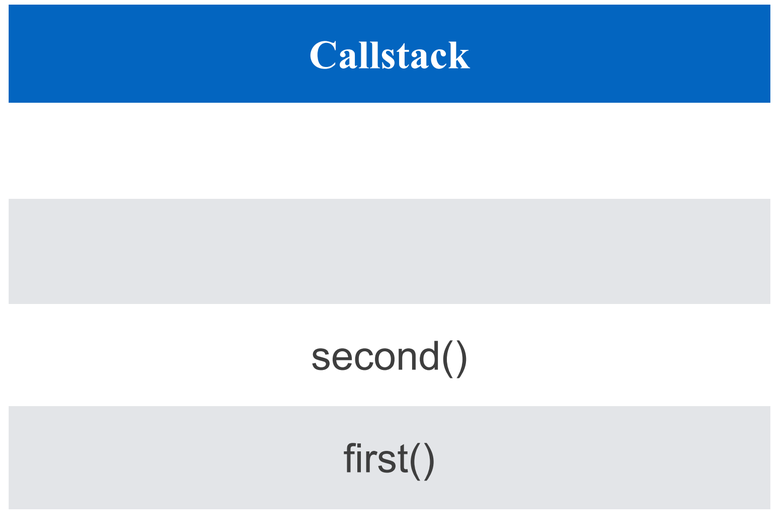
```
1   function first() {
2     console.log('I am first!');
3     second();
4     console.log('First is finished');
5   }
6
7
8
9   function second() {
10    console.log('I am second!');
11  }
12
13
14  first();
```



The call stack

```
I am first
I am second
```

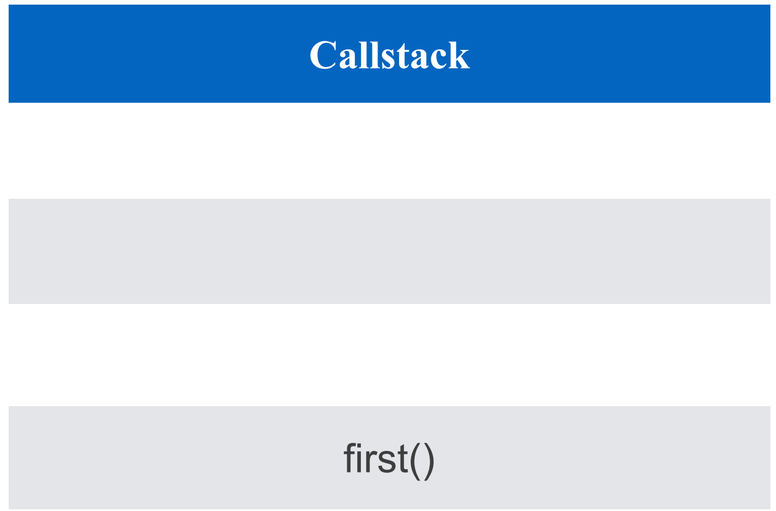
```
1   function first() {
2     console.log('I am first!');
3     second();
4     console.log('First is finished');
5   }
6
7
8
9   function second() {
10    console.log('I am second!');
11  }
12
13
14  first();
```



The call stack

```
I am first
I am second
First is finished
```

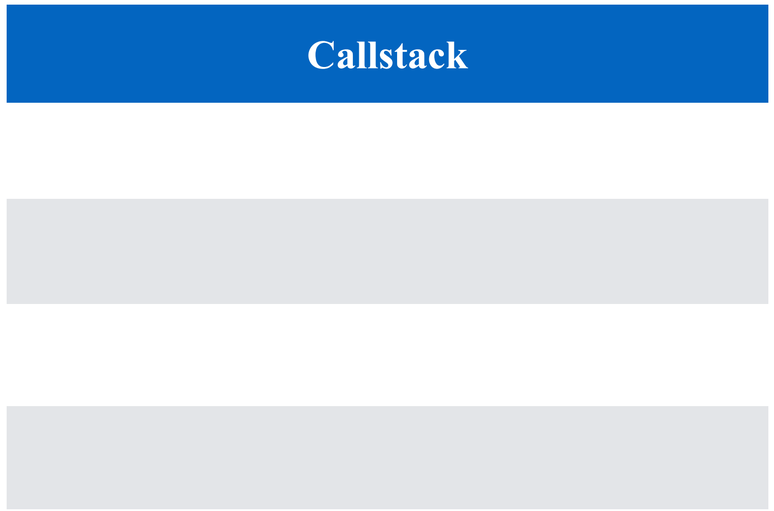
```
1   function first() {
2     console.log('I am first!');
3     second();
4     console.log('First is finished'); // first "paused" while second ran
5   }
6
7
8
9   function second() {
10    console.log('I am second!');
11  }
12
13
14  first();
```



The call stack

```
I am first
I am second
First is finished
```

```
1   function first() {
2     console.log('I am first!');
3     second();
4     console.log('First is finished'); // first "paused" while second ran
5   }
6
7
8
9   function second() {
10    console.log('I am second!');
11  }
12
13
14  first();
```



example: countdown

5
4
3
2
1

```
/* write a function that counts down to 1 */



function countdown(num) {
  for (let i = num; i >= 1; i--) {
    console.log(i);
  }
}



countdown(5);

```

example: countdown

5
4
3
2
1

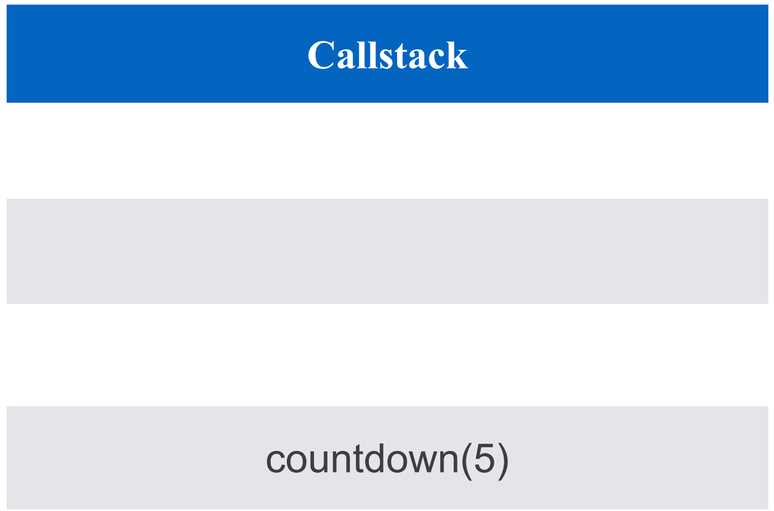
```
1   /* let's refactor our solution, writing a function that takes a number and
2      and logs it out */
3
4   function countdown(num) {
5     console.log(num);
6   }
7
8
9
10  countdown(5);
11  countdown(4);
12  countdown(3);
13  countdown(2); // notice, no loops!
14  countdown(1); // how do the arguments change between calls?
```

example: countdown

```
1   /* every time we called countdown, we subtracted one from the previous
2      num */
3
4   /* instead of manually calling countdown over and over, why not have
5      countdown call itself, subtracting one from num each time? */
6
7
8
9   function countdown(num) {
10    console.log(num);
11    countdown(num - 1);
12  }
13
14


    countdown(5);
```

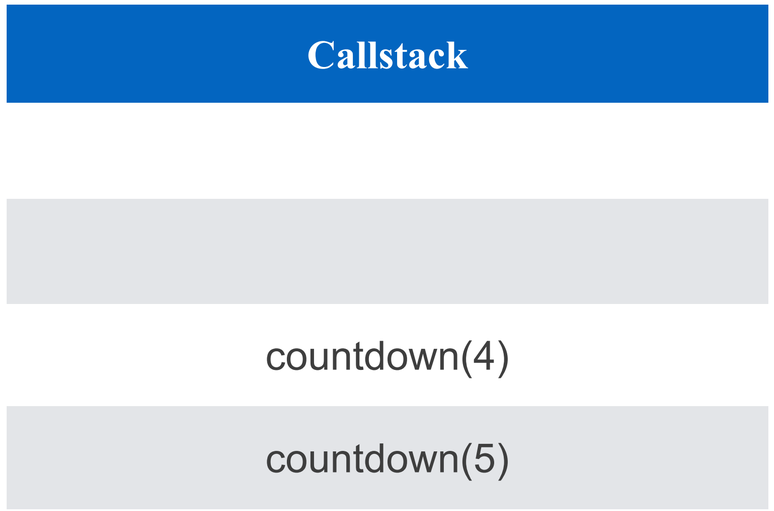
example: countdown

```
1   /* every time we called countdown, we subtracted one from the previous
2      num */
3
4   /* instead of manually calling countdown over and over, why not have
5      countdown call itself, subtracting one from num each time? */
6
7
8
9   function countdown(num) {
10    console.log(num);
11    countdown(num - 1);
12  }
13
14


    countdown(5);
```



example: countdown

```
1   /* every time we called countdown, we subtracted one from the previous
2      num */
3
4   /* instead of manually calling countdown over and over, why not have
5      countdown call itself, subtracting one from num each time? */
6
7
8
9   function countdown(num) {
10    console.log(num);
11    countdown(num - 1);
12  }
13
14


    countdown(5);
```



example: countdown

5
4
3

```
/* every time we called countdown, we subtracted one from the previous
   num */

/* instead of manually calling countdown over and over, why not have
   countdown call itself, subtracting one from num each time? */



function countdown(num) {
  console.log(num);
  countdown(num - 1);
}



countdown(5);
```

Callstack

countdown(3)

countdown(4)

countdown(5)

example: countdown

```
5
4
3
2
```

```
1   /* every time we called countdown, we subtracted one from the previous
2      num */
3
4   /* instead of manually calling countdown over and over, why not have
5      countdown call itself, subtracting one from num each time? */
6
7
8
9   function countdown(num) {
10    console.log(num);
11    countdown(num - 1);
12  }
13
14


    countdown(5);
```

Callstack
countdown(2)
countdown(3)
countdown(4)
countdown(5)

example: countdown

```
5
4
3
2
1
```

```
/* every time we called countdown, we subtracted one from the previous
   num */

/* instead of manually calling countdown over and over, why not have
   countdown call itself, subtracting one from num each time? */



function countdown(num) {
  console.log(num);
  countdown(num - 1);
}



countdown(5);
```

Callstack
countdown(1)
countdown(2)
countdown(3)
countdown(4)
countdown(5)

example: countdown

```
5
4
3
2
1
0
```

```
/* every time we called countdown, we subtracted one from the previous
   num */

/* instead of manually calling countdown over and over, why not have
   countdown call itself, subtracting one from num each time? */



function countdown(num) {
  console.log(num);
  countdown(num - 1);
}



countdown(5);
```

Callstack
countdown(0)
countdown(1)
countdown(2)
countdown(3)
countdown(4)
countdown(5)

example: countdown

```
5
4
3
2
1
0
-1
```

```
/* every time we called countdown, we subtracted one from the previous
   num */

/* instead of manually calling countdown over and over, why not have
   countdown call itself, subtracting one from num each time? */



function countdown(num) {
  console.log(num);
  countdown(num - 1);
}



countdown(5);
```

Callstack
countdown(-1)
countdown(0)
countdown(1)
countdown(2)
countdown(3)
countdown(4)
countdown(5)

example: countdown

```
5
4
3
2
1
0
-1
-2
```

```
/* every time we called countdown, we subtracted one from the previous
   num */

/* instead of manually calling countdown over and over, why not have
   countdown call itself, subtracting one from num each time? */



function countdown(num) {
  console.log(num);
  countdown(num - 1);
}



countdown(5);
```

Callstack
countdown(-2)
countdown(-1)
countdown(0)
countdown(1)
countdown(2)
countdown(3)
countdown(4)
countdown(5)

example: countdown

```
5
4
3
2
1
0
-1
-2
(and so on)
```

```
/* every time we called countdown, we subtracted one from the previous
   num */

/* instead of manually calling countdown over and over, why not have
   countdown call itself, subtracting one from num each time? */



function countdown(num) {
  console.log(num);
  countdown(num - 1);
}



countdown(5);
```

Callstack
(and so on)
countdown(-2)
countdown(-1)
countdown(0)
countdown(1)
countdown(2)
countdown(3)
countdown(4)
countdown(5)

example: countdown

```
1   /* every time we called countdown, we subtracted one from the previous
2      num */
3
4   /* instead of manually calling countdown over and over, why not have
5      countdown call itself, subtracting one from num each time? */
6
7
8
9   function countdown(num) {
10    console.log(num);
11    countdown(num - 1);
12  }
13
14


    countdown(5);
```

```
5
4
3
2
1
0
-1
-2
-3
-4
-5
-6
-7
-8
-9
-10
-11
RangeError:
Maximum call stack
size exceeded
```

example: countdown

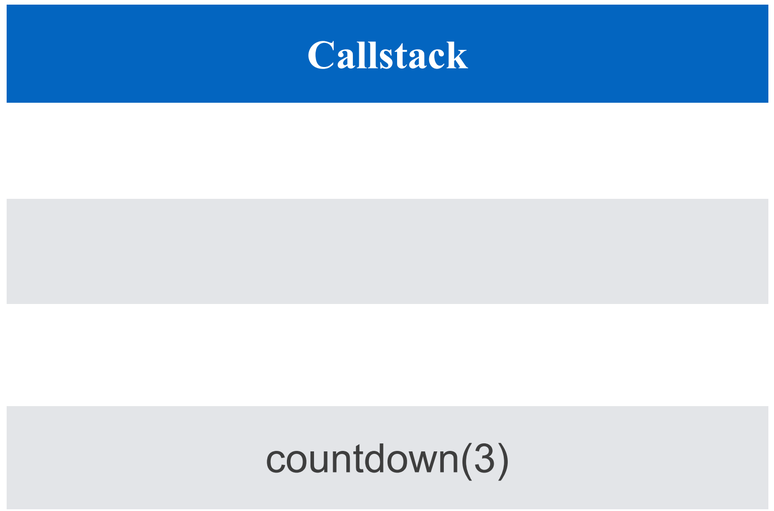
```
1   /* that started off so promisingly! */
2
3   /* because our function was instructed to call itself every time, the
4      function ends up calling itself forever until our computer runs out of
5      memory */
6
7
8
9   /* let's write in a stop condition so the function eventually stops
10     calling itself */
11
12
13
14
```

example: countdown

```
1   function countdown(num) {
2     // here's our stop condition, commonly known as the 'base case'
3     if (num < 1) {
4       console.log('done!');
5     }
6     // here's our 'recursive case'
7     else {
8       console.log(num);
9       countdown(num - 1);
10    }
11  }
12
13
14
    countdown(3);
```

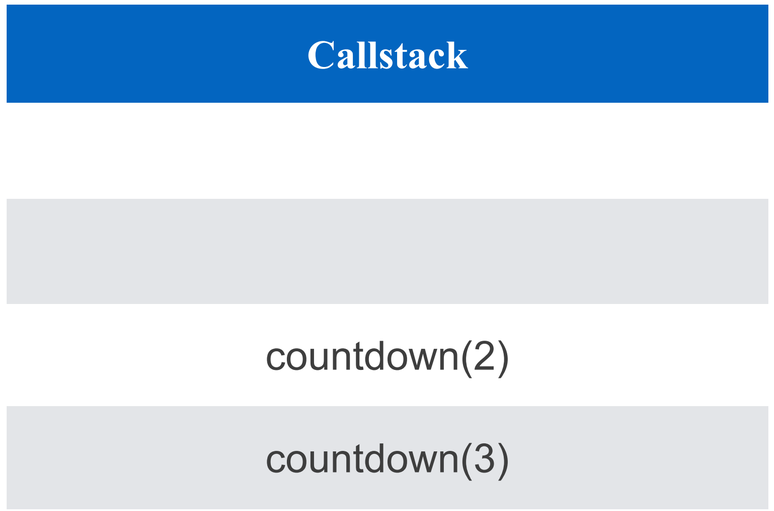
example: countdown

```
1   function countdown(num) {
2     // here's our stop condition, commonly known as the 'base case'
3     if (num < 1) {
4       console.log('done!');
5     }
6     // here's our 'recursive case'
7     else {
8       console.log(num);
9       countdown(num - 1);
10    }
11  }
12
13
14
    countdown(3);
```



example: countdown

```
1   function countdown(num) {
2     // here's our stop condition, commonly known as the 'base case'
3     if (num < 1) {
4       console.log('done!');
5     }
6     // here's our 'recursive case'
7     else {
8       console.log(num);
9       countdown(num - 1);
10    }
11  }
12
13
14
    countdown(3);
```



3
2
1

example: countdown

```
1   function countdown(num) {
2     // here's our stop condition, commonly known as the 'base case'
3     if (num < 1) {
4       console.log('done!');
5     }
6     // here's our 'recursive case'
7     else {
8       console.log(num);
9       countdown(num - 1);
10    }
11  }
12
13
14
    countdown(3);
```

Callstack

countdown(1)

countdown(2)

countdown(3)

example: countdown

```
3
2
1
done!
```

```
1   function countdown(num) {
2     // here's our stop condition, commonly known as the 'base case'
3     if (num < 1) {
4       console.log('done!');
5     }
6     // here's our 'recursive case'
7     else {
8       console.log(num);
9       countdown(num - 1);
10    }
11  }
12
13
14
    countdown(3);
```

Callstack
countdown(0)
countdown(1)
countdown(2)
countdown(3)

example: countdown

```
3
2
1
done!
```

```
1   function countdown(num) {
2     // here's our stop condition, commonly known as the 'base case'
3     if (num < 1) {
4       console.log('done!');
5     }
6     // here's our 'recursive case'
7     else {
8       console.log(num);
9       countdown(num - 1);
10    }
11  }
12
13
14
    countdown(3);
```

Callstack

countdown(1)

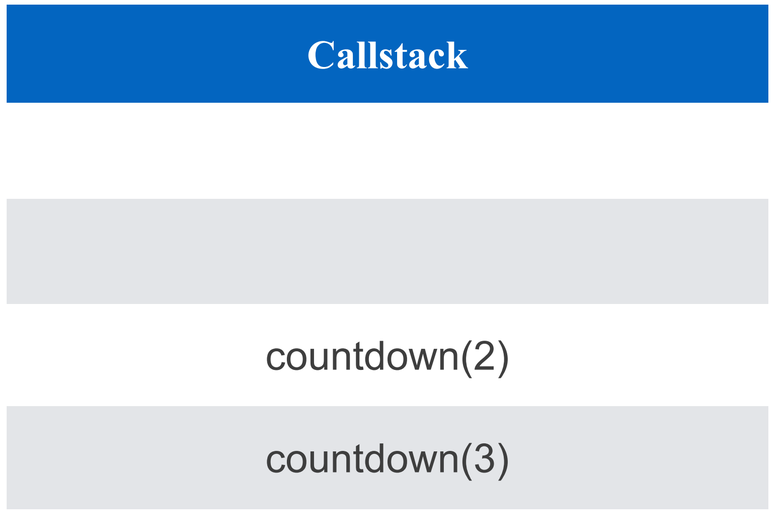
countdown(2)

countdown(3)

example: countdown

```
3
2
1
done!
```

```
1   function countdown(num) {
2     // here's our stop condition, commonly known as the 'base case'
3     if (num < 1) {
4       console.log('done!');
5     }
6     // here's our 'recursive case'
7     else {
8       console.log(num);
9       countdown(num - 1);
10    }
11  }
12
13
14
    countdown(3);
```



example: countdown

```
3
2
1
done!
```

```
1   function countdown(num) {
2     // here's our stop condition, commonly known as the 'base case'
3     if (num < 1) {
4       console.log('done!');
5     }
6     // here's our 'recursive case'
7     else {
8       console.log(num);
9       countdown(num - 1);
10    }
11  }
12
13
14
    countdown(3);
```

Callstack

countdown(3)

example: countdown

```
3
2
1
done!
```

```
1   function countdown(num) {
2     // here's our stop condition, commonly known as the 'base case'
3     if (num < 1) {
4       console.log('done!');
5     }
6     // here's our 'recursive case'
7     else {
8       console.log(num);
9       countdown(num - 1);
10    }
11  }
12
13
14
    countdown(3);
```

example: countdown

```
1   /* two takeaways from countdown: */
2
3   /* 1. you need to define a base case! */
4
5
6
7   /* 2. your recursive case must change the input to the function so that
8         you will eventually trigger the base case! */
9
10
11
12
13
14
```

Returning from recursive calls

```
/* recursion becomes more complicated when the function must return a
   value */

/* good practice is to start by defining a base case */




/* base cases are often occur when there is a simple input that expects a
   simple output (e.g., the sum of a single number is that number) */



/* test that the base case works before working with the recursive
   case! */
```

example: factorial

```
1   /* define a function, factorial, that take a number and returns the
2      factorial of that number */
3
4   /* as a reminder:
5      0! === 1
6      1! === 1
7      2! === 2 (2 * 1)
8      3! === 6 (3 * 2 * 1)
9      4! === 24 (4 * 3 * 2 * 1)
10     5! === 120 (5 * 4 * 3 * 2 * 1) */
11
12
13
14  /* what look like simple inputs/outputs we can use to build a base
       case? */
```

1
1

example: factorial

```
1   function factorial(num) {
2     // base case: num is 0 or 1
3     if (num === 0 || num === 1) {
4       return 1;
5     }
6   }
7
8
9
10  factorial(0);
11  factorial(1);
12
13
14
```

example: factorial

```
/* ok, base case is set, just need to remember that our recursive case
   has bring num closer and closer to 1 or 0 so we eventually
   hit our base case */



/* notice an interesting pattern!

   0! === 1
   1! === 1
   2! === 2 (2 * factorial(1))
   3! === 6 (3 * factorial(2))
   4! === 24 (4 * factorial(3))
   5! === 120 (5 * factorial(4)) */
```

undefined

example: factorial

```
1   function factorial(num) {
2     // base case: num is 0 or 1
3     if (num === 0 || num === 1) {
4       return 1;
5     }
6     // recursive case: num must get closer to 0 or 1
7     // TODO
8   }
9
10
11
12  /* it's best to write your recursive case using the simplest possible
13     input that will result in a recursive call */
14  let result = factorial(2);
    console.log(result);
```

1
undefined

example: factorial

```
1   function factorial(num) {
2     // base case: num is 0 or 1
3     if (num === 0 || num === 1) {
4       return 1;
5     }
6     // recursive case: num must get closer to 0 or 1
7     // we know we have to call factorial again in the recursive case
8     // if num === 2, what do we get if we call factorial again with num - 1?
9     console.log(factorial(num - 1));
10  }
11
12
13
14  let result = factorial(2);
    console.log(result);
```

2
undefined

example: factorial

```
1   function factorial(num) {
2     // base case: num is 0 or 1
3     if (num === 0 || num === 1) {
4       return 1;
5     }
6     // recursive case: num must get closer to 0 or 1
7     // from that pattern we noticed earlier, we know 2! === 2 * 1!
8     console.log(num * factorial(num - 1));
9   }
10
11
12
13  let result = factorial(2);
14  console.log(result);
```

example: factorial

```
1   function factorial(num) {
2     // base case: num is 0 or 1
3     if (num === 0 || num === 1) {
4       return 1;
5     }
6     // recursive case: num must get closer to 0 or 1
7     // just have to return the result now
8     let result = num * factorial(num - 1);
9     return result;
10  }
11
12
13
14  let result = factorial(2);
    console.log(result);
```

example: factorial

```
1   function factorial(num) {
2     // base case: num is 0 or 1
3     if (num === 0 || num === 1) {
4       return 1;
5     }
6     // recursive case: num must get closer to 0 or 1
7     let result = num * factorial(num - 1);
8     return result;
9   }
10
11
12
13  let result = factorial(5);
14  console.log(result);
```

call stack	return value

example: factorial

```
1   function factorial(num) {
2     // base case: num is 0 or 1
3     if (num === 0 || num === 1) {
4       return 1;
5     }
6     // recursive case: num must get closer to 0 or 1
7     let result = num * factorial(num - 1);
8     return result;
9   }
10
11
12
13  let result = factorial(5);
14  console.log(result);
```

call stack	return value
factorial(5)	5 * factorial(4)

example: factorial

```
1   function factorial(num) {
2     // base case: num is 0 or 1
3     if (num === 0 || num === 1) {
4       return 1;
5     }
6     // recursive case: num must get closer to 0 or 1
7     let result = num * factorial(num - 1);
8     return result;
9   }
10
11
12
13  let result = factorial(5);
14  console.log(result);
```

call stack	return value
factorial(4)	4 * factorial(3)
factorial(5)	5 * factorial(4)

example: factorial

```
1   function factorial(num) {
2     // base case: num is 0 or 1
3     if (num === 0 || num === 1) {
4       return 1;
5     }
6     // recursive case: num must get closer to 0 or 1
7     let result = num * factorial(num - 1);
8     return result;
9   }
10
11
12
13  let result = factorial(5);
14  console.log(result);
```

call stack	return value
factorial(3)	3 * factorial(2)
factorial(4)	4 * factorial(3)
factorial(5)	5 * factorial(4)

example: factorial

```
1   function factorial(num) {
2     // base case: num is 0 or 1
3     if (num === 0 || num === 1) {
4       return 1;
5     }
6     // recursive case: num must get closer to 0 or 1
7     let result = num * factorial(num - 1);
8     return result;
9   }
10
11
12
13  let result = factorial(5);
14  console.log(result);
```

call stack	return value
factorial(2)	2 * factorial(1)
factorial(3)	3 * factorial(2)
factorial(4)	4 * factorial(3)
factorial(5)	5 * factorial(4)

example: factorial

```
1   function factorial(num) {
2     // base case: num is 0 or 1
3     if (num === 0 || num === 1) {
4       return 1;
5     }
6     // recursive case: num must get closer to 0 or 1
7     let result = num * factorial(num - 1);
8     return result;
9   }
10
11
12
13  let result = factorial(5);
14  console.log(result);
```

call stack	return value
factorial(1)	=> 1
factorial(2)	2 * factorial(1)
factorial(3)	3 * factorial(2)
factorial(4)	4 * factorial(3)
factorial(5)	5 * factorial(4)

example: factorial

```
1   function factorial(num) {
2     // base case: num is 0 or 1
3     if (num === 0 || num === 1) {
4       return 1;
5     }
6     // recursive case: num must get closer to 0 or 1
7     let result = num * factorial(num - 1);
8     return result;
9   }
10
11
12
13  let result = factorial(5);
14  console.log(result);
```

call stack	return value
factorial(2)	2 * 1
factorial(3)	3 * factorial(2)
factorial(4)	4 * factorial(3)
factorial(5)	5 * factorial(4)

example: factorial

```
1   function factorial(num) {
2     // base case: num is 0 or 1
3     if (num === 0 || num === 1) {
4       return 1;
5     }
6     // recursive case: num must get closer to 0 or 1
7     let result = num * factorial(num - 1);
8     return result;
9   }
10
11
12
13  let result = factorial(5);
14  console.log(result);
```

call stack	return value
factorial(2)	=> 2
factorial(3)	3 * factorial(2)
factorial(4)	4 * factorial(3)
factorial(5)	5 * factorial(4)

example: factorial

```
1   function factorial(num) {
2     // base case: num is 0 or 1
3     if (num === 0 || num === 1) {
4       return 1;
5     }
6     // recursive case: num must get closer to 0 or 1
7     let result = num * factorial(num - 1);
8     return result;
9   }
10
11
12
13  let result = factorial(5);
14  console.log(result);
```

call stack	return value
factorial(3)	3 * 2
factorial(4)	4 * factorial(3)
factorial(5)	5 * factorial(4)

example: factorial

```
1   function factorial(num) {
2     // base case: num is 0 or 1
3     if (num === 0 || num === 1) {
4       return 1;
5     }
6     // recursive case: num must get closer to 0 or 1
7     let result = num * factorial(num - 1);
8     return result;
9   }
10
11
12
13  let result = factorial(5);
14  console.log(result);
```

call stack	return value
factorial(3)	=> 6
factorial(4)	4 * factorial(3)
factorial(5)	5 * factorial(4)

example: factorial

```
1   function factorial(num) {
2     // base case: num is 0 or 1
3     if (num === 0 || num === 1) {
4       return 1;
5     }
6     // recursive case: num must get closer to 0 or 1
7     let result = num * factorial(num - 1);
8     return result;
9   }
10
11
12
13  let result = factorial(5);
14  console.log(result);
```

call stack	return value
factorial(4)	4 * 6
factorial(5)	5 * factorial(4)

example: factorial

```
1   function factorial(num) {
2     // base case: num is 0 or 1
3     if (num === 0 || num === 1) {
4       return 1;
5     }
6     // recursive case: num must get closer to 0 or 1
7     let result = num * factorial(num - 1);
8     return result;
9   }
10
11
12
13  let result = factorial(5);
14  console.log(result);
```

call stack	return value
factorial(4)	=> 24
factorial(5)	5 * factorial(4)

example: factorial

```
1   function factorial(num) {
2     // base case: num is 0 or 1
3     if (num === 0 || num === 1) {
4       return 1;
5     }
6     // recursive case: num must get closer to 0 or 1
7     let result = num * factorial(num - 1);
8     return result;
9   }
10
11
12
13  let result = factorial(5);
14  console.log(result);
```

call stack	return value
factorial(5)	5 * 24

example: factorial

```
1   function factorial(num) {
2     // base case: num is 0 or 1
3     if (num === 0 || num === 1) {
4       return 1;
5     }
6     // recursive case: num must get closer to 0 or 1
7     let result = num * factorial(num - 1);
8     return result;
9   }
10
11
12
13  let result = factorial(5);
14  console.log(result);
```

call stack	return value
factorial(5)	=> 120

example: factorial

```
1   function factorial(num) {
2     // base case: num is 0 or 1
3     if (num === 0 || num === 1) {
4       return 1;
5     }
6     // recursive case: num must get closer to 0 or 1
7     let result = num * factorial(num - 1);
8     return result;
9   }
10
11
12
13  let result = factorial(5);
14  console.log(result);
```

call stack	return value

example: factorial

```
1   /* three takeaways from factorial: */
2
3   /* 1. write your base case first, and test it using simple
4      inputs/outputs */
5
6
7
8   /* 2. write your base case, and test it using the simplest possible
9         input that results in one recursive call to the base case */
10
11
12
13  /* 3. test your function against more-complex inputs */
14
    /* 4. You should try to use chromes debugger, it can help inspect recursive calls, because console.log will not
    be as useful with recursion. */
```

recursion and iterables

```
1   /* you can use recursion with any data type in JS */
2
3   /* if you're asked to recurse through arrays or strings, the base case
4      often occurs when the iterable is empty or has a length of one */
5
6
7
8   /* imagine finding the sum of numbers in an array */
9
10
11
12  sumArray([4]); // if array.length === 1, the sum is easy to calculate
13
14

    /* if the base case required the iterable to have a length of 1 or 0, it
       must mean that the recursive case has to reduce the length of the
       iterable with every recursive call */
```

other recursion hints

```
1   /* cannot emphasize enough: start with the base case! */
2
3
4
5   /* cannot emphasize enough: test recursive case with simplest possible
6      input that will result in one recursive call to the base case */
7
8
9
10  /* ask yourself: what type of thing should my function return? base case
11     and recursive case should return the same type of thing! */
12
13
14  /* use console.logs or debugger to debug */

    /* If you get stuck on a recursive problem, try writing it iteratively, e.g. how you normally would at this point. It
    can reveal the inner workings of recursion to just see how you normally would do it */
```

Recap

```
/*
  - Definition of recursion
  - The call stack
  - countdown example
  - factorial example
  - Recursion and arrays, strings
  - Tips for approaching recursion problems
*/
```