Recursion I

Overview

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

What is recursion?

```
1   /* recursion is when a function calls itself!  */
2
3    */ recursion helps break big problems into
4          small chunks */
5
6   /* recursion is an alternative to iteration (using a loop) */
7
8
9
10
11
12
13
14
```

How do you decide whether to use recursion or iteration?

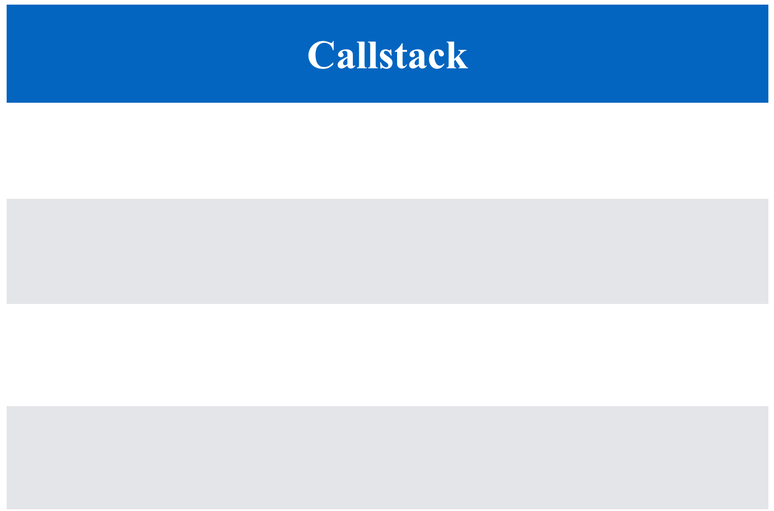
```
/* in the real world, you may see recursion instead of iteration when a
   recursive solution is:
     - easier to reason about
     - easier to read than an iterative solution
     - won't negatively affect performance too much (recursion can be
       a memory hog)
 */
```

The call stack

```
1  /* to understand recursion it is important that we 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
```

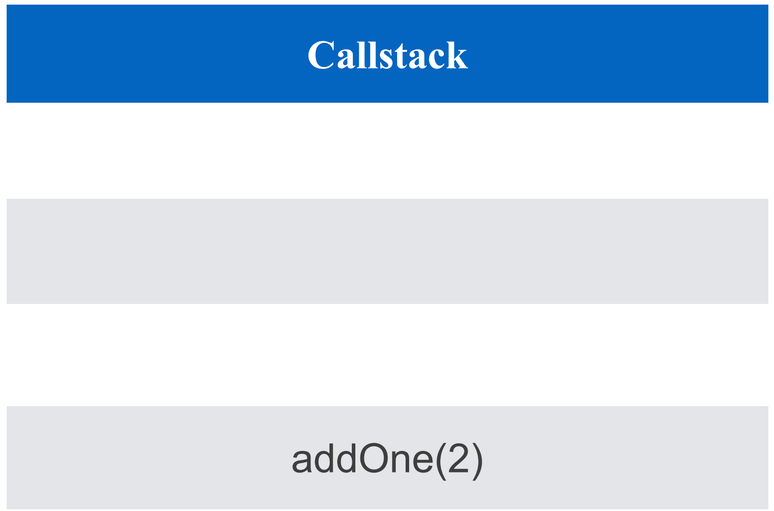
How does JS use the call stack to figure out which function to call?

```
/* 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 addOne(num) {
  return num + 1
}

function addTwo(num) {
  return num + 2
}
addOne(2);
addTwo(3);
```



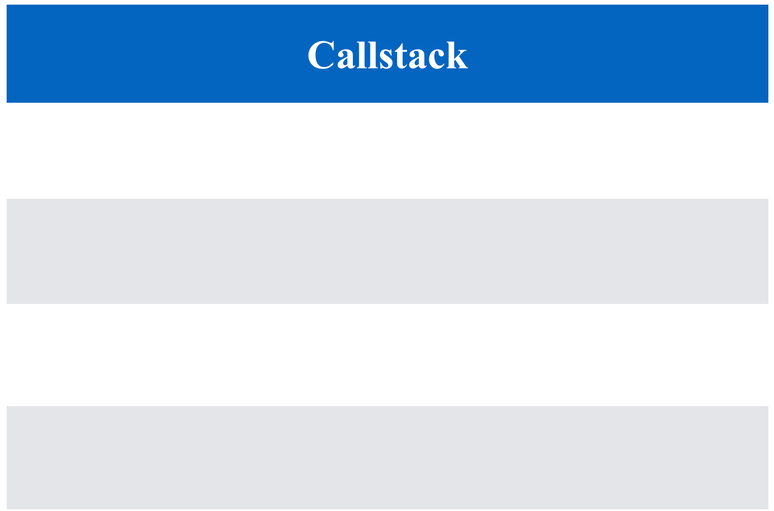
The call stack

```
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 addOne(num) {
10    return num + 1
11  }
12  function addTwo(num) {
13    return num + 2
14  }
15
16  addOne(2);
    addTwo(3);
```



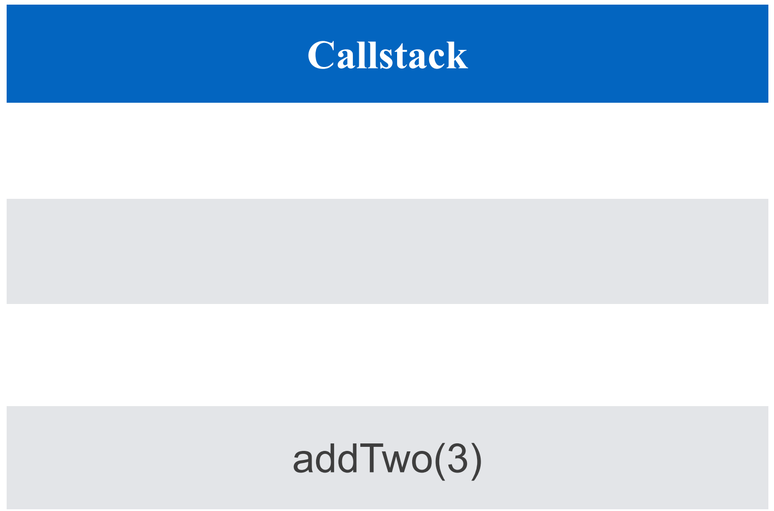
The call stack

```
/* 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 addOne(num) {
  return num + 1
}
function second(num) {
  return num + 2
}

addOne(2);
addTwo(3);
```



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 addOne(num) {
  return num + 1
}
function addTwo(num) {
  return num + 2
}

addOne(2);
addTwo(3);
```



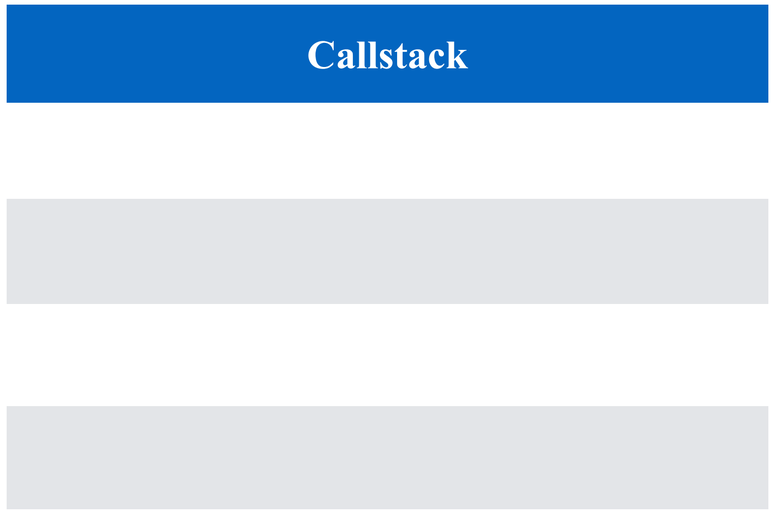
The call stack

```
/* 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 addOne(num) {
  return num + 1
}
function addTwo(num) {
  return num + 2
}

addOne(2);
addTwo(3);
```

Callstack

How does the call stack execute the following code?

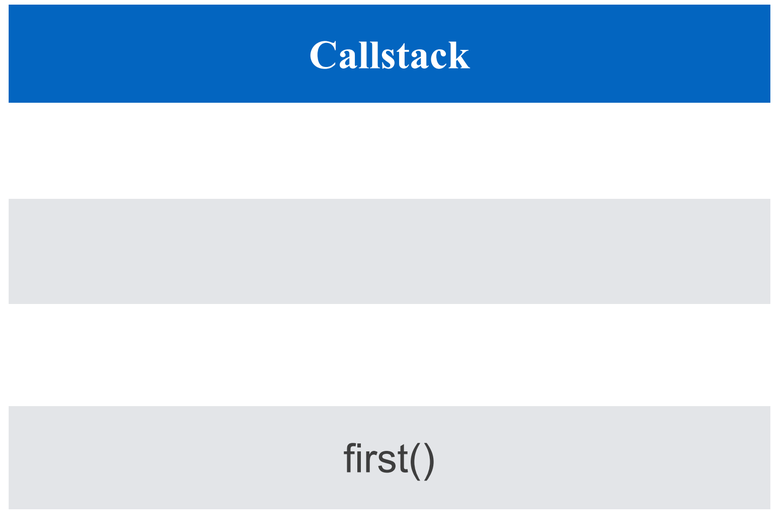
```
1   function first() {
2     console.log('I am first!');
3     second();
4     console.log('First is finished');
5   }
6
7
8
9   function second() {
10    console.log('I am second!');
11  }
12
13
14  first();
```



I am first

The call stack

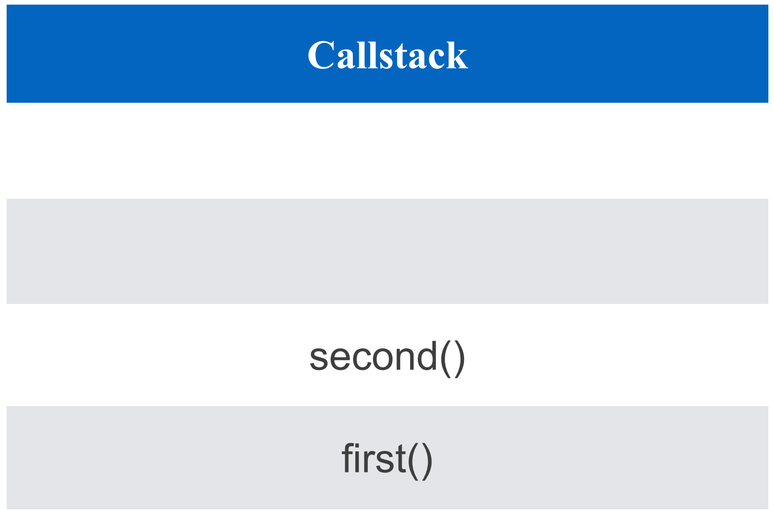
```
1   function first() {
2     console.log('I am first!');
3     second();
4     console.log('First is finished');
5   }
6
7
8
9   function second() {
10    console.log('I am second!');
11  }
12
13
14
15  first();
```



I am first
I am second

The call stack

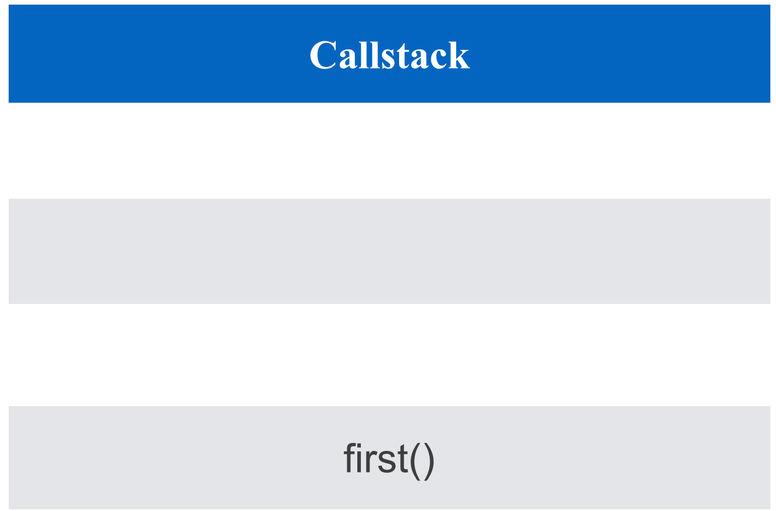
```
1   function first() {
2     console.log('I am first!');
3     second();
4     console.log('First is finished');// first "paused" while second ran
5   }
6
7
8
9   function second() {
10    console.log('I am second!');
11  }
12
13
14  first();
```



The call stack

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

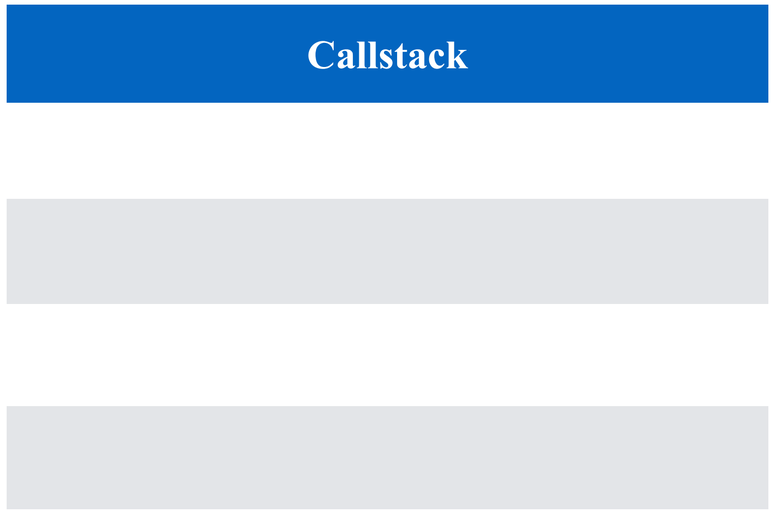
```
1   function first() {
2     console.log('I am first!');
3     second();
4     console.log('First is finished'); // first "paused" while second ran
5   }
6
7
8
9   function second() {
10    console.log('I am second!');
11  }
12
13
14  first();
```



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();
```



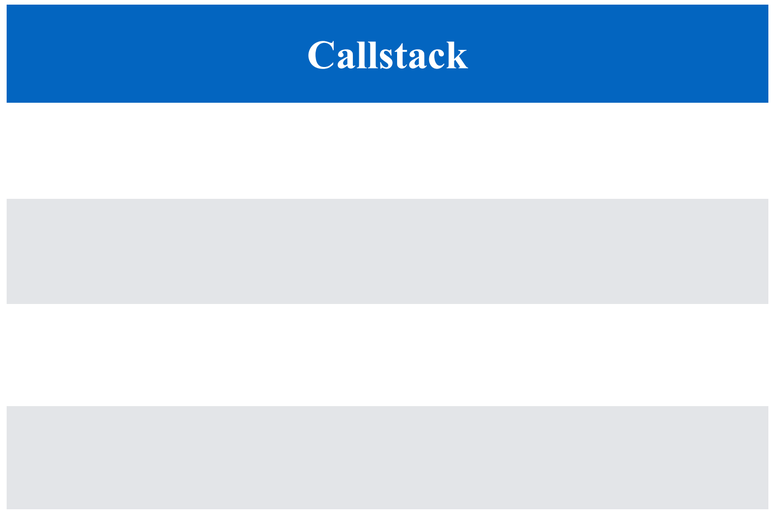
example: countdown

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);

```

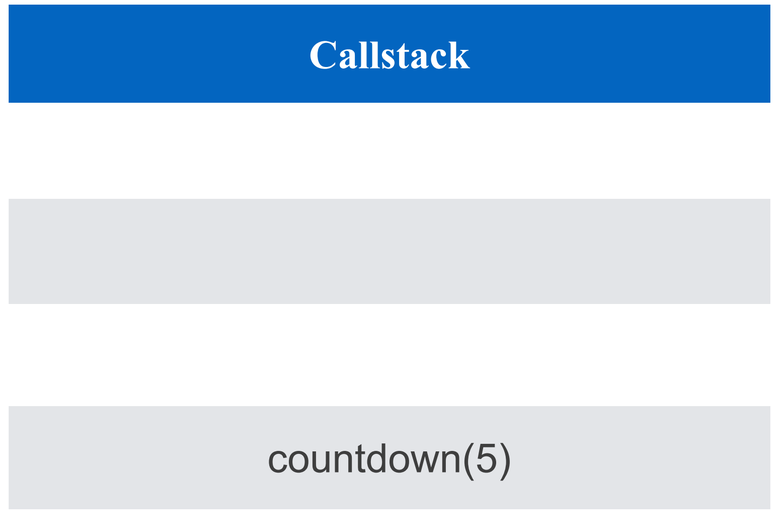
recursion property 1: the recursive case

```
1   function countdown(num) {
2     console.log(num);
3     countdown(num - 1);
4   }
5
6
7
8   countdown(5);
9
10
11
12
13
14
```



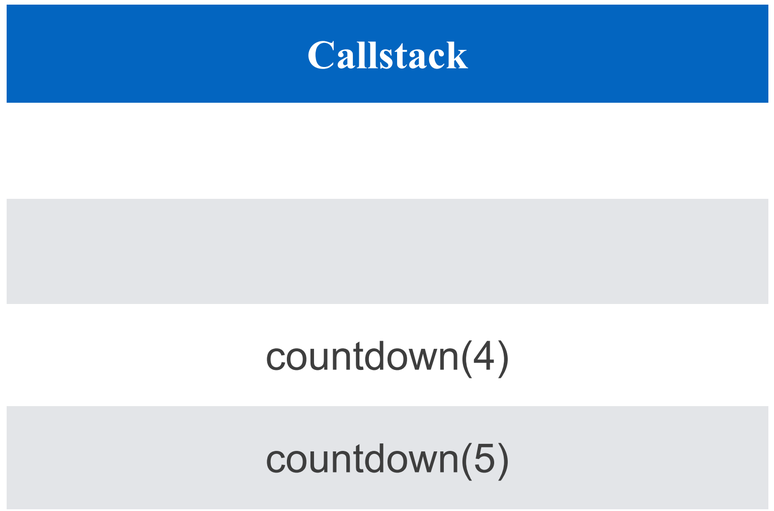
example: countdown

```
1   /* every time we called countdown, we subtracted one from the previous
2      num */
3
4
5
6   function countdown(num) {
7     console.log(num);
8     countdown(num - 1);
9   }
10
11
12
13  countdown(5);
14
```



example: countdown

```
1   /* every time we called countdown, we subtracted one from the previous
2      num */
3
4
5
6   function countdown(num) {
7     console.log(num);
8     countdown(num - 1);
9   }
10
11
12
13  countdown(5);
14
```



example: countdown

5
4
3

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



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



countdown(5);

```

Callstack

countdown(3)

countdown(4)

countdown(5)

example: countdown

```
5
4
3
2
```

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



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



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 */



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 */



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 */



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 */



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 */



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
5
6   function countdown(num) {
7     console.log(num);
8     countdown(num - 1);
9   }
10
11
12
13  countdown(5);
14
```

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

property 2 : the stop condition

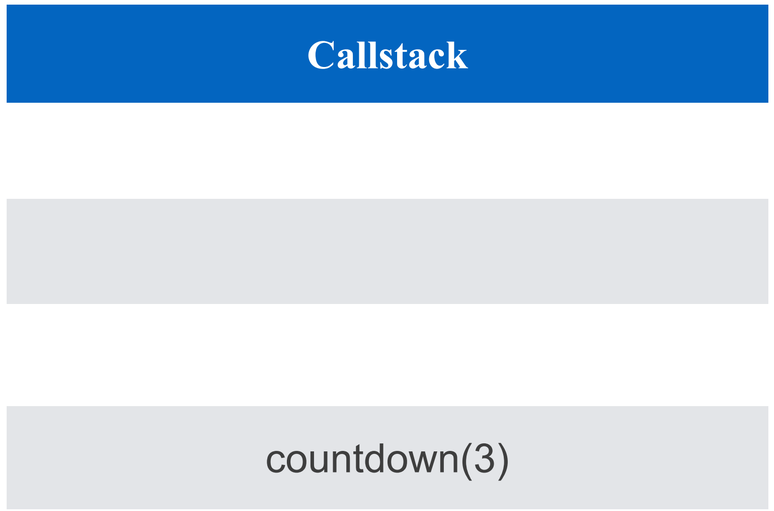
```
/* that started off so promisingly! */

/* because our function was instructed to call itself every time, the
   function ends up calling itself forever until our computer runs out of
   memory */



/* let's write in a stop condition so the function eventually stops
   calling itself */
```

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);
```

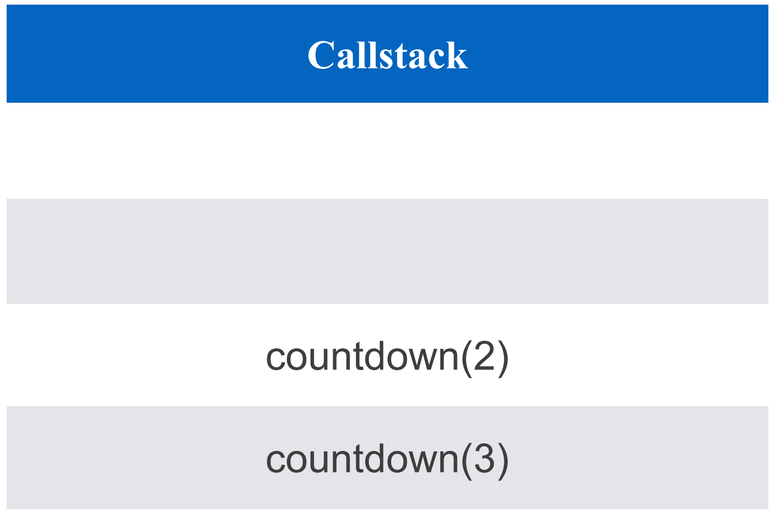
example: countdown

```
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)

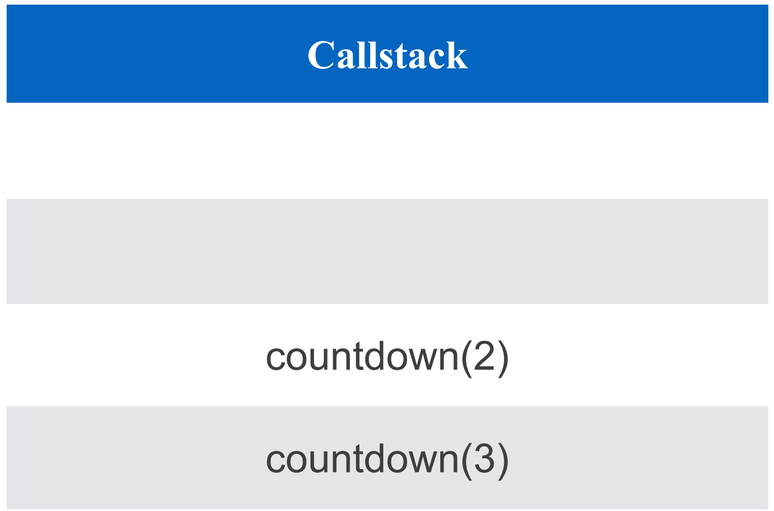
countdown(2)

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

```
1   /* recursion becomes more complicated when the function must return a
2      value */
3
4   /* good practice is to start by defining a base case */
5
6
7
8
9
10
11
12
13
14
```

example - raise x to the power n

if asked to raise x to the power n...that can happen by multiplying x n times

example - raise x to the power n

```
function pow(x, n) {
  let result = 1;

  // multiply result by x n times in the loop
  for (let i = 0; i < n; i++) {
    result *= x;
  }

  return result;
}

alert( pow(2, 3) ); // 8
```

example - raise x to the power n - pseudocode

base case - n === 1

example - raise x to the power n - pseudocode

if n === 1 return x, else multiply x n-1 times recursively

example - raise x to the power n - pseudocode

```
1  function pow(x, n) {
2    if (n == 1) {
3      return x;
4    } else {
5      return x * pow(x, n - 1);
6    }
7  }
8
9  alert( pow(2, 3) ); // 8
```

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

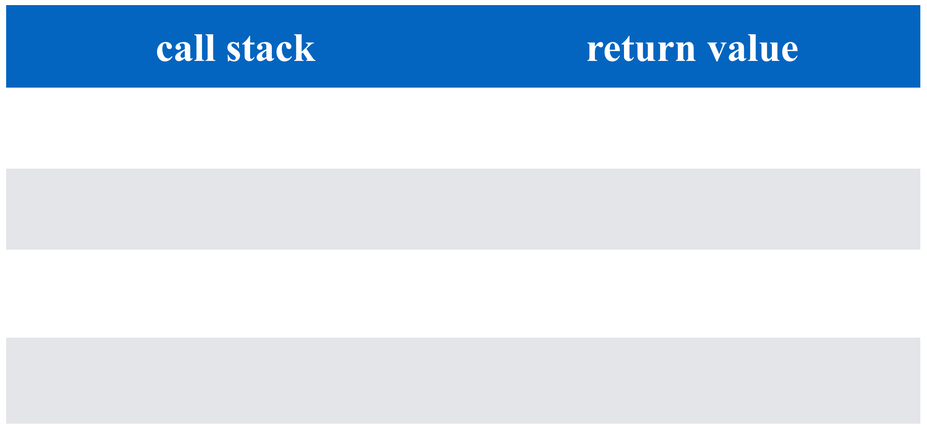
```
/* ok, base case is set, just need to remember that our recursive case
   has bring num closer and closer to 1 or 0 so we eventually
   hit our base case */



/* 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)) */
```

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);
```



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
```

recursion and iterables

```
/* you can use recursion with any data type in JS */

/* if you're asked to recurse through arrays or strings, the base case
   often occurs when the iterable is empty or has a length of one */
```

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
```

Recap

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