# Functions

### Function declaration syntax

- function name is built into the declaration of the function

```
function sayHello1() {
  console.log('hello');
  console.log('bye');
}
```

### Function expression syntax

- setting a variable to an anonymous function

```
let sayHello2 = function() {
  console.log('hello');
  console.log('bye');
};
```

sayHello2();
Functions are **first class objects**

- functions can be saved to variables

```
// examples of what can be saved to a variable

let name = 'Alvin'
let age = 1000;
let getAvg = function(num1, num2) {
  return (num1 + num2)/2
};
```

# Mutability

**Immutable** Types

- Cannot be mutated
- Examples
  - Number, e.g. `122`
  - `NaN`

- `undefined`
- `null`
- String, e.g. `abc`

**Mutable** Types

- Can be mutated
- Examples
  - Arrays, e.g. `['a', 'b', 'c']`
  - Objects (we will see this later)

# Array Functions

`Array.prototype.push`

- This is how it will show up on MDN. We also like to use the notation, `Array#push`
- takes a single argument and adds the argument passed in to the end of the array that it is called on
- mutates the array it is called on
- returns the length of the mutated array

```
let people = ['Gordon', 'Soon-Mi', 'Angela'];
people.push('Justin');

console.log(people); // ['Gordon', 'Soon-Mi', 'Angela', 'Justin']
```

`Array.prototype.pop` or `Array#pop`

- doesn't take any arguments and removes the last element in the array
- mutates the array it is called on
- returns the removed element

```
let dogs = ['Fido', 'Digby', 'Fluffy'];

const lastDog = dogs.pop();

console.log(lastDog); // 'Fluffy'
console.log(dogs); // ['Fido', 'Digby']
```

`Array.prototype.shift` or `Array#shift`

- doesn't take any arguments and removes the first element in the array

- mutates the array it is called on
- returns the removed element

```
let cats = ['Paprika', 'Whiskers', 'Garfield'];

let firstCat = cats.shift();

console.log(firstCat); // 'Paprika'
console.log(cats); // ['Whiskers', 'Garfield']
```

`Array.prototype.unshift` or `Array#unshift`

- takes a single argument and adds the argument to the beginning of the array
- mutates the array it is called on
- returns the length of the mutated array

```
let cats = ['Whiskers', 'Garfield']

cats.unshift('Sennacy');

console.log(cats); // ['Sennacy', 'Whiskers', 'Garfield']
console.log(beforeUnshift === cats);
```

# Nested Loops

```
for (let i = 0; i < 3; i++) {
  for (let j = 0; j < 4; j++) {
    console.log(i, j);
  }
}
```

# Pairs In Arrays

- Nested loops with arrays

## Pairs In Arrays


## Unique Pairs In Arrays


Unique Pairs In Arrays

```
// Closure
// When an inner function uses, or changes,
// variables defined in an outer scope.
// NOT for declaring a variable of the same name in an inner scope.
function sayHi() {
    let name = 'Bryan Guner';
    function greeting() {
        // here greeting function closes over, or captures, the name variable
        // to read it's value
        return "Hi there, " + name + "!";
    }
    // Here, we return the return value of the greeting function
    return greeting();
}
// console.log(sayHi());
function nameAndCity() {
    let person = { name: 'Sergey', city: 'Moscow' };
    function changeCity() {
        // here changeCity function closes over the person variable
        // and reassigns a value on an existing key
        person.city = 'Toronto';
    }
    changeCity();
    // the person variable will show the changes from the changeCity function
    return person;
}
// console.log(nameAndCity());
function smoothieMaker() {
    let ingredients = [];
    function addIngredient(ingredient) {
        // Here addIngredient function closes over the ingredients variable
        // to push new elements into the ingredients variable.
        // We have created a private state where we cannot access
        // the ingredients array from the outside and can only access
        // the variable from the inner function.
        ingredients.push(ingredient);
        return ingredients;
    }
    // Here the return value for smoothiemaker is the return value
    // is the function addIngredient, NOT addIngredient's return value
    return addIngredient;
}
// Here we initialize we return a new addIngredient function
// which has closed over the ingredients array
const makeSmoothie = smoothieMaker();
console.log(makeSmoothie);
console.log(makeSmoothie('spinach')); // prints [ spinach ]
console.log(makeSmoothie('turmeric')); // prints [ spinach, turmeric ]
// let mySmoothie = makeSmoothie();
// Here we return a new and different addIngredient function
// which has closed over a new a different ingredients array
```

```
const makeSmoothie2 = smoothieMaker();
console.log(makeSmoothie2('kale')); // prints [ kale ]  -- does not include spinach and turmeric
function createCounter() {
    let count = 0;
    return function () {
        count++;
        return count;
    }
}
let counter1 = createCounter();
let counter2 = createCounter();
// console.log(counter1());
// console.log(counter1());
// console.log(counter1());
// console.log(counter1());
// What will this print out?
// console.log(counter2());
// Brief talk of scope and redeclaring a const variable in a loop
for (let i = 0; i < 5; i++) {
    const num = i + 2;
}
[9:01 PM] // In the following examples we will predict what will
// be printed to the terminal
// 1
function dinerBreakfast(food) {
    let order = "I'd like cheesy scrambled eggs and ";
    function finishOrder() {
        return order + food;
    }
    return finishOrder();
}
// console.log(dinerBreakfast('green tea'));
// 2
function dinerBreakfast2(food) {
    let order = "I'd like a(n) " + food;
    function withEggs() {
        order = order + ' and cheesy scrambled eggs, please!'
    };
    withEggs();
    return order;
}
// console.log(dinerBreakfast2('avocado toast'));
// 3
function dinerBreakfast3() {
    let order = "I'd like cheesy scrambled eggs";
    return function (food) {
        order = order + " and " + food;
        return order;
    }
}
let breakfastOrder = dinerBreakfast3();
```

```
console.log(breakfastOrder);
console.log(breakfastOrder('cappuccino'));
console.log(breakfastOrder('pancakes'));
bryan-guner  [9:07 PM]
8:11
# Calculating Closures
What is a _closure_? This question is one of the _most frequent interview
questions_ where JavaScript is involved. If you answer this question quickly and
knowledgeably you'll look like a great candidate. We know you want to know it
all so let's dive right in!
The official definition of a closure from MDN is, "A closure is the combination
of a function and the lexical environment within which that function was
declared." The practicality of how a _closure_ is used it simple: a _closure_ is
when an inner function uses, or changes, variables in an outer function.
Closures in JavaScript are incredibly important in terms of the creativity,
flexibility and security of your code.
When you finish this reading you should be able to implement a closure and
explain how that closure effects scope.
## Closures and scope
Let's look at an example of a simple closure below:
```javascript
function climbTree(treeType) {
  let treeString = "You climbed a ";
  function sayClimbTree() {
    // this inner function has access to the variables in the outer scope
    // in which is was defined - including any defined parameters
    return treeString + treeType;
  }
  return sayClimbTree();
}
// We assign the result to a variable
const sayFunction = climbTree("Pine");
// So we can call it, and indeed the variables have been saved in the closure
// and the sayFunction prints out their values.
console.log(sayFunction); // You climbed a Pine
```

In the above snippet the `sayClimbTree` function captures and uses the `treeString` and `treeType` variables within its own inner scope.
Let's go over some basic closure rules:

1. Closures have access to any variables within its own, as well as any outer function's, scope when they are declared. This is where the *lexical environment* comes in - the *lexical environment* consists of any variables available within the scope in which the closure was declared (which are the local inner scope, outer function's scope, and global scope).
2. A closure will keep reference to all the variables when it was defined **even if the outer function has returned**.
```

Notice above that even though the above `climbTree` had run its `return` statement the inner function of `sayClimbTree` **still has access** to the variables( `treeString` and `treeType` ) from the outer scope where it was declared. So, even after an outer function has returned, an inner function will still have access to the outer function's variables.
Let's look at another example of a closure:

```
```js
function makeAdder(x) {
  return function(y) {
    return x + y;
  };
}
const add5 = makeAdder(5);
console.log(add5(2)); // prints 7
```

In the above example the function the anonymous function within the `makeAdder` function **closes over** the `x` variable and utilizes it within the inner anonymous function. This allows us to do some pretty cool stuff like creating the `add5` function above. Closures are your friend ♡.

# Applications of closures

Let's take a look at some of the common and practical applications of closures in JavaScript.

## Private State

Information hiding is incredibly important in the world of software engineering. JavaScript as a language does not have a way of declaring a function as exclusively private, as can be done in other programming languages. We can however, use *closures* to create private state within a function.
The following code illustrates how to use *closures* to define functions that can emulate private functions and variables:
```

```
function createCounter() {
  let count = 0;
  return function() {
    count++;
    return count;
  };
}
let counter = createCounter();
console.log(counter()); // => 1
console.log(counter()); // => 2
//we cannot reach the count variable!
counter.count; // undefined
let counter2 = createCounter();
console.log(counter2()); // => 1
```

In the above code we are storing the anonymous inner function inside the `createCounter` function onto the variable `counter`. The `counter` variable is now a *closure*. The `counter` variable **closes over** the inner `count` value inside `createCounter` even after `createCounter` has returned.
By **closing over** (or **capturing**) the `count` variable, each function that is return from `createCounter` has a **private**, mutable state that cannot be accessed externally. There is no way any outside function beside the closure itself can access the `count` state.
[pre-crement]:
https://stackoverflow.com/questions/3469885/somevariable-vs-somevariable-in-javascript

## Passing Arguments Implicitly

We can use closures to pass down arguments to helper functions without explicitly passing them into that helper function.

```
function isPalindrome(string) {
  function reverse() {
    return string
      .split("")
      .reverse()
      .join("");
  }
  return string === reverse();
}
```

# Problem-Solutions

> Write a function reverseString(str) that takes in a string. The function should return a new string where the order the characters is reversed.

Write a function reverseString(str) that takes in a string. The function should return a new string where the order the characters is reversed.

```
function reverseString(str) {
let newStr = "";
for (let i = str.length - 1; i > -1; i--) {
newStr += str[i];
}
return newStr;
}
```

Write a function range(min, max) that takes in two numbers. The function should return an array containing all numbers from min to max inclusive.

Define this function using function expression syntax.

```
function range(min, max) {
let result = [];
for (let i = min; i <= max; i++) {
result.push(i);
}
return result;
}
```

Write a function logBetweenStepper(min, max, step) that takes in 3 numbers as parameters. The function should print out numbers between min and max at step intervals. See the following examples.

Hint: this function only needs to print using console.log it does not need to return.

```
function logBetweenStepper(min, max, step) {
for (let i = min; i <= max; i += step) {
console.log(i);
}
}
```

Write a function reverseSentence(sentence) that takes in a sentence as an arg. The function should return a new sentence where the order of the words is reversed. Note that you should reverse the order among words, not the order among characters.

```
function reverseSentence(sentence) {
return sentence.split(" ").reverse().join(" ");
}
```

Write a function myIncludes(arr, target) that accepts an array and an target value as args. The function should return a boolean indicating whether the target is found in the array. Solve this without Array#includes or Array#indexOf.

```
function myIncludes(arr, target) {
for (let i = 0; i < arr.length; i++) {
if (arr[i] === target) {
return true;
}
}
return false;
}
```

Write a function initials(name) that accepts a full name as an arg. The function should return the initials for that name.

```
function initials(name) {
let arr = name.split(" ");
let initials = [];
for (let i = 0; i < arr.length; i++) {
initials.push(arr[i][0]);
}
return initials.join("").toUpperCase();
}
```

Write a function sumArray(array) that takes in an array of numbers and returns the total sum of all the numbers.

```
function sumArray(array) {
return array.reduce((a, b) => a + b);
}
```

Write a function factorsOf(num) that takes in a number as an arg. The method should return an array containing all positive numbers that are able to divide into num with no remainder.

Define this function using function expression syntax.

```
function factorsOf(num) {
let result = [];
if (num === 0) return [];
for (let i = 1; i <= num; i++) {
if (num % i === 0) {
result.push(i);
}
}
return result;
}
```

Write a function myIndexOf(arr, target) that takes in an array and target value as args. The function should return the first index where the target is found in the array. If the target is not found, it should return -1. Solve this without using Array#indexOf.

```
function myIndexOf(arr, target) {
for (let i = 0; i < arr.length; i++) {
if (arr[i] === target) {
return i;
}
}
return -1;
}
```

Write a function, countVowels(word), that takes in a string word and returns the number of vowels in the word.
Vowels are the letters "a", "e", "i", "o", "u".

```
function countVowels(word) {
const VOWEL = "aeiou";
let count = 0;
for (let i = 0; i < word.length; i++) {
if (VOWEL.indexOf(word[i]) > -1) {
count++;
}
}
return count;
}
```

Write a function hasVowel(str) that takes in a string. The function should return a boolean, true if the string contains at least one vowel, false otherwise. Vowels are the letters a, e, i, o, u.

```
function hasVowel(str) {
const VOWEL = "aeiou";
for (let i = 0; i < str.length; i++) {
if (VOWEL.indexOf(str[i]) > -1) {
return true;
}
}
return false;
}
```

Write a function oddNumbers(min, max) that takes in two numbers as args. The function should return an array containing all odd numbers between min and max, exclusive.

Define this function using function expression syntax.

```
let oddNumbers = function (min, max) {
let result = [];
for (let i = min + 1; i < max; i++) {
if (Math.abs(i % 2) === 1) {
```

```
result.push(i);
}
}
return result;
};
```

Write a function fizzBuzz(max) that accepts a number as an arg. The function should return an array containing all positive numbers less than max that are divisible by either 3 or 5, but not both.

```
function fizzBuzz(max) {
let result = [];
for (let i = 1; i < max; i++) {
if ((i % 3 === 0 || i % 5 === 0) && !(i % 3 === 0 && i % 5 === 0)) {
result.push(i);
}
}
return result;
}
```

Write a function firstVowel(str) that takes in a string and returns the first vowel that appears sequentially in the string.

```
function firstVowel(str) {
const VOWEL = "aeiou";
str = str.toLowerCase();
for (let i = 0; i < str.length; i++) {
if (VOWEL.indexOf(str[i]) > -1) {
return str[i];
}
}
return null;
}
```

Write a function evenNumbers(max) that takes in a number as an arg. The function should return an array containing all positive, even numbers that are less than max.

Define this function using function expression syntax.

```
let evenNumbers = function (max) {
let result = [];
for (let i = 2; i < max; i += 2) {
result.push(i);
}
return result;
};
```

Define a function isPrime(number) that returns true if number is prime. Otherwise, false. A number is prime if it is only divisible by 1 and itself.

```
function isPrime(number) {
for (let i = 2; i < number; i++) {
if (number % i === 0) {
return false;
}
}
return number > 1;
}
```

Write a function lastVowel(str) that takes in a string and returns the last vowel that appears sequentially in the string. Note that the string may contain capitalization.

Hint: You may find the String#toLowerCase or String#toUpperCase methods useful

```
function lastVowel(str) {
let newStr = str.toLowerCase();
const VOWEL = "aeiou";
for (let i = str.length - 1; i > -1; i--) {
if (VOWEL.indexOf(newStr[i]) > -1) {
return str[i];
}
```

```
}
return null;
}
```

Write a function pitPat(max) that accepts a number as an arg. The function should return an array containing all positive numbers less than or equal to max that are divisible by either 4 or 6, but not both.

```
function pitPat(max) {
let result = [];
for (let i = 1; i <= max; i++) {
if ((i % 4 === 0 || i % 6 === 0) && !(i % 4 === 0 && i % 6 === 0)) {
result.push(i);
}
}
return result;
}
```

Write a function removeLastVowel(word) that takes in a string and returns the string with its last vowel removed.

```
function removeLastVowel(word) {
const VOWEL = "aeiou";
let wordArr = word.split("");
for (let i = wordArr.length - 1; i > -1; i--) {
if (VOWEL.indexOf(wordArr[i]) > -1) {
wordArr.splice(i, 1);
return wordArr.join("");
}
}
return word;
}
```

Write a function pairsMaker(arr) that takes in a an array as an argument. The function should return a 2D array where the subarrays represent unique pairs of element from the input array.

```
function pairsMaker(arr) {
let result = [];
for (let i = 0; i < arr.length; i++) {
for (let j = i + 1; j < arr.length; j++) {
result.push([arr[i], arr[j]]);
}
}
return result;
}
```

Write a function minValue(nums) that takes in an array of numbers as an arg. The function should return the smallest number of the array. If the array is empty, the function should return null.

```
function minValue(nums) {
if (nums.length === 0) return null;
return Math.min(...nums);
}
```

Write a function twoSum(arr, target) that accepts an array and a target number as args. The function should a return a boolean indicating if two distinct numbers of the array add up to the target value. You can assume that input array contains only unique numbers.

```
function twoSum(arr, target) {
for (let i = 0; i < arr.length; i++) {
for (let j = i + 1; j < arr.length; j++) {
if (arr[i] + arr[j] === target) {
return true;
}
}
}
return false;
}
```

Write a function rotateRight(array, num) that takes in an array and a number as args. The function should return a new array where the elements of the array are rotated to the right num times. The function should not mutate the original array and instead return a new array.

Define this function using function expression syntax.

HINT: you can use Array#slice to create a copy of an array

```
function rotateRight(array, num) {
let result = array.slice(0);
for (let i = 0; i < num; i++) {
let ele = result.pop();
result.unshift(ele);
}
return result;
}
```

Write a function twoDimensionalSum(arr) that takes in a 2D array of numbers and returns the total sum of all numbers.

```
function twoDimensionalSum(arr) {
let sum = 0;
for (let i = 0; i < arr.length; i++) {
let subArr = arr[i];
for (let j = 0; j < subArr.length; j++) {
sum += subArr[j];
}
}
return sum;
}
```

Write a function rotateLeft(array, num) that takes in an array and a number as args. The function should rotate the elements of the array to the left num times, mutating the original array. The function should return undefined.

Define this function using function expression syntax.

```
function rotateLeft(array, num) {
for (let i = 0; i < num; i++) {
let ele = array.shift();
array.push(ele);
}
}
```

Pig Latin is a fun take on the English language where you move any consonant cluster from the start of the word to the end of the word; when words begin on a vowel, you simply add "-yay". Vowels are "aeiou".

Write a function pigLatinWord that takes in a word string and translates the word into Pig Latin. For this problem use the String#slice method. The slice() method extracts a section of a string and returns it as a new string, without modifying the original string.

Hint: Remember the String#includes method!

```
function pigLatinWord(word) {
const VOWEL = "aeiou";
if (VOWEL.indexOf(word[0]) > -1) return word + "yay";
for (let i = 0; i < word.length; i++) {
if (VOWEL.indexOf(word[i]) > -1) {
return word.slice(i) + word.slice(0, i) + "ay";
}
}
}
```

Write a function leastCommonMultiple(num1, num2) that accepts two numbers as arguments. The functions should return the smallest number that is divisible by both num1 and num2.

**Normal Solution**

```
function leastCommonMultiple(num1, num2) {
let upperBound = num1 * num2;
let lowerBound = Math.max(num1, num2);
for (let i = lowerBound; i <= upperBound; i++) {
if (i % num1 === 0 && i % num2 === 0) {
```

```
return i;
}
}
}
```