```
/*
1- reverse a string (without using str.reverse())
2- find out if a given word is a palindrome (without using str.reverse()). ie. kayak, mom, racecar
      BONUS: do the same for sentences. ie. "A Man, A Plan, A Canal, Panama!"
3- given an array of integers find the most repeated number
      ie. findMostRepeated([3,3,4,4,4,5,6,6,6,1,1,4,4,4,1,1,1,1,1,4,4,4,4,6]) = 4
4- given an array of integers find the number with the longest repeating sequence
      ie. findLongestSeq([3,3,4,4,4,5,6,6,6,1,1,4,4,4,1,1,1,1,1,4,4,4,4,6]) = 1
```

Class Inheritance:
instances inherit from classes (like a blueprint — a description of the class), and create sub-class relationships: hierarchical class taxonomies. Instances are typically instantiated via constructor functions with the `new` keyword. Class inheritance may or may not use the `class` keyword from ES6.
Prototypal Inheritance:
instances inherit directly from other objects. Instances are typically instantiated via factory functions or `Object.create()`. Instances may be composed from many different objects, allowing for easy selective inheritance.
In JavaScript, prototypal inheritance is simpler & more flexible than class inheritance.
https://www.w3schools.com/js/js_object_prototypes.asp

The tight coupling problem
The fragile base class problem
Inflexible hierarchy problem
The duplication by necessity problem
The Gorilla/banana problem
https://youtu.be/lKCCZTUx0sI

JS FUNDAMENTAL CONCEPTS : SCOPE, HOISTING, CLOSURES, CALLBACKS, PROMISES, Event Loop, Prototypes
https://medium.com/javascript-scene/the-two-pillars-of-javascript-ee6f3281e7f3
https://medium.com/javascript-scene/the-two-pillars-of-javascript-pt-2-functional-programming-a63aa53a41a4

COMPOSITION
https://youtu.be/wfMtDGfHWpA
https://alligator.io/js/class-composition
https://ui.dev/javascript-inheritance-vs-composition

DESIGN PATTERNS
Design patterns are documented solutions to commonly occurring problems in software engineering.
Engineers don't have to bang their heads on the problems that someone else has already solved.
https://www.telerik.com/blogs/design-patterns-in-javascript
singleton, builder, factory, observer

Concatenative inheritance:
The process of inheriting features directly from one object to another by copying the source objects properties. In JavaScript, source prototypes are commonly referred to as mixins. Since ES6, this feature has a convenience utility in JavaScript called `Object.assign()`. Prior to ES6, this was commonly done with Underscore/Lodash's `.extend()` jQuery's `$.extend()`, and so on… The composition example above uses concatenative inheritance.
Prototype delegation:
In JavaScript, an object may have a link to a prototype for delegation. If a property is not found on the object, the lookup is delegated to the delegate prototype, which may have a link to its own delegate prototype, and so on up the chain until you arrive at `Object.prototype`, which is the root delegate. This is the prototype that gets hooked up when you attach to a `Constructor.prototype` and instantiate with `new`. You can also use `Object.create()` for this purpose, and even mix this technique with concatenation in order to flatten multiple prototypes to a single delegate, or extend the object instance after creation.
Functional inheritance:
In JavaScript, any function can create an object. When that function is not a constructor (or `class`), it's called a factory function. Functional inheritance works by producing an object from a factory, and extending the produced object by assigning properties to it directly (using concatenative inheritance). Douglas Crockford coined the term, but functional inheritance has been in common use in JavaScript for a long time.
As you're probably starting to realize, concatenative inheritance is the secret sauce that enables object composition in JavaScript, which makes both prototype delegation and functional inheritance a lot more interesting.
When most people think of prototypal OO in JavaScript, they think of prototype delegation. By now you should see that they're missing out on a lot. Delegate prototypes aren't the great alternative to class inheritance — object composition is.
prototype chain image: https://miro.medium.com/max/2400/1*Lu-BaawSayDz1itKPk2u4w.png

REACT

ONE WAY DATA FLOW
One way data flow means that the model is the single source of truth. Changes in the UI trigger messages that signal user intent to the model (or "store" in React). Only the model has the access to change the app's state. The effect is that data always flows in a single direction, which makes it easier to understand.
One way data flows are deterministic, whereas two-way binding can cause side-effects which are harder to follow and understand.

STATE
States are the heart of React components. States are the source of data and must be kept as simple as possible.
Basically, states are the objects which determine components rendering and behavior.
They are mutable unlike the props and create dynamic and interactive components.

PROPS
Props is the shorthand for Properties in React. They are read-only components which must be kept pure i.e. immutable.
They are always passed down from the parent to the child components throughout the application.

A child component can never send a prop back to the parent component.
This help in maintaining the unidirectional data flow and are generally used to render the dynamically generated data.

PHASES OF COMPONENTS LIFECYCLE
Initial Rendering Phase:
This is the phase when the component is about to start its life journey and make its way to the DOM.
Updating Phase:
Once the component gets added to the DOM, it can potentially update and re-render only when a prop or state change occurs. That happens only in this phase.
Unmounting Phase:
This is the final phase of a component's life cycle in which the component is destroyed and removed from the DOM.

LIFECYCLE METHODS
componentWillMount() - Executed just before rendering takes place both on the client as well as server-side.
componentDidMount() - Executed on the client side only after the first render.
componentWillReceiveProps() - Invoked as soon as the props are received from the parent class and before another render is called.
shouldComponentUpdate() - Returns true or false value based on certain conditions. If you want your component to update, return true else return false. By default, it returns false.
componentWillUpdate() - Called just before rendering takes place in the DOM.
componentDidUpdate() - Called immediately after rendering takes place.
componentWillUnmount() - Called after the component is unmounted from the DOM. It is used to clear up the memory spaces.

HOOKS
Hooks are the new feature introduced in the React 16.8 version. It allows you to use state and other React features without writing a class.
Hooks are the functions which "hook into" React state and lifecycle features from function components.

Higher Order Components(HOC)
Higher Order Component is an advanced way of reusing the component logic.
Basically, it's a pattern that is derived from React's compositional nature. HOC are custom components which wrap another component within it.
They can accept any dynamically provided child component but they won't modify or copy any behavior from their input components.
You can say that HOC are 'pure' components.

HOC can be used for many tasks like:
Code reuse, logic and bootstrap abstraction
Render High jacking
State abstraction and manipulation
Props manipulation

What are Pure Components
Pure components are the simplest and fastest components which can be written.
They can replace any component which only has a render().
These components enhance the simplicity of the code and performance of the application.

REDUX
Redux is one of the most trending libraries for front-end development in today's marketplace.
It is a predictable state container for JavaScript applications and is used for the entire applications state management.
Applications developed with Redux are easy to test and can run in different environments showing consistent behavior.

Three principles that Redux follows
Single source of truth:
The state of the entire application is stored in an object/ state tree within a single store.
The single state tree makes it easier to keep track of changes over time and debug or inspect the application.
State is read-only:
The only way to change the state is to trigger an action.
An action is a plain JS object describing the change.
Just like state is the minimal representation of data, the action is the minimal representation of the change to that data.
Changes are made with pure functions:
In order to specify how the state tree is transformed by actions, you need pure functions.
Pure functions are those whose return value depends solely on the values of their arguments.

*/

```
input[type = "text"] {
   color: #000000;
}
```

```
function create() {
      let counter = 0
      let someVar
      return {
            increment: function() {
                  counter++
            },
```

```
            print: function() {
                  console.log(counter)
            },

            doSomething: function() {
            }
      }
}
let c = create()
c.increment()
c.print()




const palindrome = (string) => {
      for ( let i = 0; i < string.length/2; i ++) {
            If (string[i] !== string[string.length - 1 - i) {
                  Return false
            }
      }
      Return True
}


const reverseString = (string) => {
      let arr = string.split('')
      for ( let i = 0; i < string.length/2; i ++) {
            [arr[i], arr [string.length - 1 - i]] = [arr [string.length - 1 - i], arr[i]]
      }
      return arr.join('')
}



3- given an array of integers find the most repeated number
      ie. findMostRepeated([3,3,4,4,4,5,6,6,6,1,1,4,4,4,1,1,1,1,1,4,4,4,4,6]) = 4


Const findMostRepeated = (array) => {
      numberHash = {}

      For (let number of array) {
            If (number in numberHash) {
                  numberHash[number] += 1;
            } else {
                  numberHash[number] = 1;
            }
      }

      let max = {
            value : 0,
            key: null
      }

      Object.keys(numberHash).forEach(key => {
            let value = numberHash[key]
            If (numberHash[key] >  max.value) {
                  max.value = value
                  max.key = key
            }

      )
      return max.key
}

// array.reduce(function(total, currentValue, currentIndex, arr), initialValue)
// given an array of integers find the max (biggest number)
```