**Calcula el área de un triángulo. Creando funciones**

Para escribir y ejecutar la mayoría del código que vamos a escribir puedes usar la consola de desarrollo de tu navegador.

Para calcular el area de un cuadrado podemos ejecutar:

console.log("El área de un triángulo de base 5 y altura 7 es: " + 5 \* 7 / 2)

¿Pero si queremos calcular el area de diferentes cuadrados? podemos automatizar esto utilizando una función

Una función encapsula una pieza de código que podemos ejecutar dependiendo de valores que les pasemos, para escribir nuestra funciones usamos el keyword function

Las funciones pueden tener un nombre, pero si no definimos este nombre son funciones anonimas, los valores que recibe se llaman parametros y devuelven un valor usando el keyword return

**function** **triangleArea**(base, height) {

**return** base \* height / 2

}

triangleArea(5,7)

Algo importante del lenguaje es que podemos asignar una función a una variable y usar un *arrow function* que nos hace la lectura del código más legible.

**let** triangleArea = (base, height) => base \* height / 2

***CONST, VAR, LET***

CONST: Es una constante la cual NO cambiara su valor en ningún momento en el futuro.

VAR: Es una variable que SI puede cambiar su valor y su scope es local.

LET: Es una variable que también podra cambiar su valor, pero solo vivirá(Funcionara) en el bloque donde fue declarada.

**¿Quiénes pueden pasar a ver una película? Ejercicio con condicionales, expresiones y booleanos**

Las condiciones nos permiten decidir el código a ejecutar dependiendo de que sucede. Por ejemplo una persona puede pasar a ver una película o no.

Las condicionales se definen con el keyword if la estructura es

**if** (condición) {

//código a ejecutar si se cumple la condición

} **else** {

//código a ejecutar si no se cumple la condición

}

Algunas de las cosas que podemos evaluar son:

* > es mayor
* < es menor
* >= es mayor e igual
* <= es menor e igual
* == es igual
* != es diferente

Si queremos concatenar varias condiciones lo podemos hacer usando el keyword else if

**Inventar un idioma manipulando strings**

Los strings son cadenas de texto como palabras, frases, etc.

cuando ejecutamos métodos sobre un string estos no se modifican, debemos asignarlo a otra variable.

Cada letra del texto tiene un indice y este indice comienza en 0, por ejemplo

**“platzi”** seria

0 = P

1 = l

2 = a

3 = t

4 = z

5 = i

También podemos comenzar a contar su indice desde el final,

-6 = P

-5 = l

-4 = a

-3 = t

-2 = z

-1 = i

**Concatenar**

Podemos unir dos string utilizando el operador +, por ejemplo:

**const** palabra = \'Pla\' + \'tzi\'

palabra == \'Platzi\'

**Convertir a arrays**

Podemos convertir los arrays a caracteres con el metodo split diciéndole por cual carácter dividirlo, por ejemplo

let **str** = \'hola\'

**str**.**split**(\'\') == [\'h\',\'o\',\'l\',\'a\']

También podemos unir un array y convertirlo en un array usando el metodo join

**let** arr = [\'h\',\'o\',\'l\',\'a\']

arr.**join**(\'\') == \'hola\'

**Metodos utiles.**

**str**.toUpperCase() // convierte el texto a mayúscula

**str**.toLowerCase() // convierte el texto en minúsculas

**str**.endsWith(\'\') // evalúa si el string termina con un texto

**str**.startsWith(\'\') // evalúa si un string comienza con un texto

**str**.slice(inicio, **final**) // partir un carácter

**str**.length // cuantos caracteres tiene el string

**¿Cuántos kms corre una persona en promedio? Entendiendo el ciclo for**

JavaScript nos permite ejecutar cierto código una cantidad de veces definida, por ejemplo podemos recorrer un array con un ciclo:

**const** dias = [\'lunes\',\'martes\',\'miércoles\',\'jueves\',\'sábado\',\'domingo\']

**for** (let i=0; i < dias.length; i++) {

console.log(\'dia: \', dias[i])

}

La construcción de for tiene tres valores

1. El iterador
2. La condición
3. Cómo cambia el iterador

“Hay 3 formas de crear objetos”:

1. Creando una funcion y usando prototipos
2. Creando un objeto y usando Createobject()
3. Creando una clase y creando una instancia de la misma

**Definiendo la clase Punto - Prototipos en JavaScript**

Cuando tenemos varios objetos que compartan la misma estructura podemos crear un prototipo del cual podemos crear los objetos que queramos.

Para nuestro ejercicio de los puntos podemos crear esta estructura como

**function** **Punto**(x,y) {

**this**.x = x

**this**.y = y

}

**const** p1 = **new** Punto(0,4)

Con el keyword new creamos un nuevo objeto

Los objetos se vinculan con otros objetos a través de prototype, en el navegador lo puedes ver como **proto** y mostrara el prototipo con el que fue creado

**Definiendo la clase Punto - Object.create en JavaScript**

En JavaScript tenemos varias formas para crear objetos, ahora vamos a crear un nuevo constructor del objeto punto

**const** Punto = {

init: **function** **init**(x,y) {

**this**.x = x

**this**.y = y

},

moverEnX: **function** **moverEnX**(x) {

**this**.x += x

},

moverEnY: **function** **moverEnY**(y) {

**this**.y += y

}

}

**const** p1 = Object.create(Punto)

p1.init(0,4)

**Definiendo la clase Punto - Class en JavaScript**

Una funcionalidad agregada en EcmaScript 2015, es poder escribir el constructor de los objetos con el keyword class, podemos definir nuestro constructor Punto como

**class** **Punto** {

**constructor**(x,y) {

**this**.x = x

**this**.y = y

}

moverEnX(x) {

**this**.x += x

}

moverEnY(y) {

**this**.y += y

}

**const** p1 = **new** Punto(0,4)

**const** p2 = **new** Punto(3,0)

**Entiende el scope de las variables**

Escope es el conjunto de variables y funciones que podemos llamar en una parte del código

Si definimos una variable en el scope global (window en el navegador) podemos acceder a ella dentro de cualquier lugar del código.

Si definimos una variable dentro de una función solo lo podemos utilizar dentro de esa función si lo declaramos con var, si lo declaramos con let solo podemos acceder a ella en el bloque de código.

**Scope** Conjunto de variables y funciones al cual podemos acceder en una determinada linea de código

**Closures** Funciones que recuerdan el scope (encotorno) en el que fueron creadas y pueden acceder a todas la variables y funciones que pertenecen a ese scope pero con los valores que tienen al momento en que se invoca ese closure

\*\*Let \*\* permite asignar valor a nuestras variables con visibilidad / validez únicamente dentro del contexto (llaves {}) en el que se ha definido.

**hoisting** Las variables JavaScript pueden hacer referencia a una variable declarada más tarde, sin obtener un error. Este concepto es conocido como hoisting, es decir, una variable declarada se “eleva” o “sube” a la parte superior de la función (closure) en la que se encuentra, independientemente de donde la declaremos.

El **scope**, también llamado contexto o alcance, es el ámbito que tienen las variables para poder ser utilizadas, depende mucho de con qué palabra reservada (let, var, const) se declaren las variables para que éstas pueden tener un comportamiento determinado en los scopes.

* **Var**: Con var existe algo llamado hoisting, que consiste en que las variables se “elevan” al principio de la función en donde fueron declaradas, incluso si fueron declaradas dentro de un scope, lo que hace var es declarar esa variable al principio de la función.
* **Let**: let es la forma correcta con la que debemos de declarar las variables, ya que respeta el scope y no genera información innecesaria
* **Const**: const es la palabra ideal que representa que un programador tiene el cuidado y el conocimiento necesario para poder declara una variable que no va a cambiar.

**Operaciones con arrays**

Los arrays son una forma de guardar nuestros datos muy importante en JavaScript, en este vídeo veremos como trabajar con Arrays.

Si queremos hacer una función que reciba N parámetros y devuelva la suma de estos, podemos escribirla como

**function** **suma**(...params) {

**return** params.reduce((acumulativo, actual) => {acumulativo + actual}, 0)

}

suma(4,5,6,23,26,7,8)

De esta forma utilizamos el método reduce con el cual cuentan los arrays, este nos permite ir iterando por todo el array y acumulando el valor en una variable.

Ahora queremos ejecutar una operación sobre todos los elementos del array, para esto tenemos un método llamadomap

**function** **doble**(...params) {

**return** params.map(x=>x\*2)

}

doble(2,3,4,5,6,7,5)

Si queremos filtrar ciertos valores de un array, tenemos otro método para poder hacer esto con keyword

**const** pares = (...numeros) => numeros.filter(x=> x % 2 == 0)

**Entiende los closures de JavaScript**

Los closure son funciones que recuerdan el entorno en el cual fuerón creadas, esto quiere decir que al llamar la función van a recordar las variables que tenian en ese momento. Por ejemplo, si queremos hacer una función para saludar determinada familia

**function** **saludarFamilia**(apellido) {

**return** **function**(nombre) {

console.log(`Hola ${nombre} ${apellido}`)

}

}

**let** saludarPerez = saludarFamilia(\'perez\');

``

De esta forma podemos crear nuevas funciones partiendo de funciones que recuerdan variables internas

**Estructura del lenguaje**

Algunas características del lenguaje son:

* Podemos asignarle cualquier valor a las variables, podemos asignarle strings, números, fechas, funciones, objetos, etc.
* Si definimos una variable con const no podemos reasignar el valor al que hacen referencia
* Las variables definidas con var y let pueden ser reasignadas
* El carácter punto y coma no es obligatorio,

**This, \_this y los arrow functions**

This hace referencia a un objeto y su valor depende de donde lo usemos, por ejemplo

**class** **Persona** {

**constructor**(nombre, amigos = []) {

**this**.nombre = nombre

**this**.amigos = amigos

}

}

**const** sacha = **new** Persona("Sacha", ["pedro", "juan", "pepe"])

**La función bind**

El método bind devuelve otra función que asigna el this a lo que nosotros le pasamos como parámetro.

**call y apply**

Con call y apply podemos definir el valor de this y ejecutar la función

* **Metodo bind():** Es un método de todas las funciones, el cual **nos devuelve una función para ser invocada en cualquier momento**. Se caracteriza por permitirnos definir el scope-contexto y por permitir establecer previamente los parametros

const newFunction = **fun**.**bind**(contexto, primerParametro)// Establece el scope y el-los parametros de fun

newFunction(segundoParametro) // Ejecuta fun pero con la caracteristica de que ya esta establecido el scope y los parametros. Igualmente nos permite enviarle más parametros a fun si es el caso

-**Métodos call() y apply():** A diferencia de bidn(), call() y apply() **ejecutan la función “padre” instantaneamente**. De igual forma nos permiten establecer el scope y los parametros con la pequeña diferencia de que apply, recibe los parametros en un array.

**fun**.**call**(contexto, primerParametro, segundoParametro)//Ejecuta fun en el scope establecido y con los parametros enviados

**fun**.**apply**(contexto, [primerParametro, segundoParametro)//Ejecuta fun en el scope establecido y con los parametros enviados en el array

**ECMAScript: El estándar en el que se basa JavaScript**

El estándar en el que se basa JavaScript.

Comenzó llamándose LiveScript para lo que se mostraba al cliente.

Java != JavaScript

Se llamó ECMAScript porque fue desarrollado por ECMA, una empresa dedicada a telecomunicaciones.

ECMA-262

Sus primeras versiones fueron lanzadas en los 90’s.

En este momento se está creando ECMAScript 2018.

Si escribimos código con ECMAScript tiene que ser compatible con el navegador en el que queremos que corra.

En el siguiente vamos a hablar sobre BABEL.

**Babel al rescate: logrando la compatibilidad buscada**

Babel

Usa JavaScript de la próxima generación hoy mismo.

Babel es una herramienta que te permite hacer compatible el código que usa funcionalidades de ECMAScript con los diferentes navegadores.

Para poder utilizar Babel debes tener instalado Node.

**Que es un modulo en javascript?**

Básicamente un modulo es un paquete de codigo. Un modulo es un pedazo de código encapsulado que se encarga de resolver alguna cosa en particular.

Las razones por las cuales nosotros vamos a querer escribir un modulo son dos, la primera es encapsulamiento, todo lo que pasa en un pedazo de codigo tiene que ser local a ese pedazo de codigo, de esa manera cada parte del codigo se hace responsable de cumplir una tarea. NO es que hacemos un modulo que hace un monton de cosas , sino que cada modulo debe ser pequeño y encargarse de una sola responsabilidad.  
La segunda razon por la cual nosotros queremos escribir un modulo son las dependencias.

**Como crear un modulo**

Existen varias manera de escribir un modulo, como por ejemplo con commonjs, amd, y el nuevo metodo de ecmascript 2015.  
La manera de crear modulos usando el metodo de ecmascript 2015 es la manera mas utilizada y mas facil de implementar. Basicamente hacemos import y export en cada modulo. Por ejemplo si tenemos un modulo que se llama cuadrado que sirve para calcular cosas de un cuadrado podemos utiliar este modulo en otro modulo importandolo

**Distintas formas de escribir módulos en JavaScript**

Sistemas de Módulos

¿Qué es un módulo?  
¿Por qué queremos escribir un módulo?

Encapsulamiento

Cada módulo debe ser pequeño y enfocado en una funcionalidad.

Capsulas de código que se pueden reutilizar.

Ejemplos de esto son Angular y Node.

Un módulo es un paquete de código.

**Los callbacks de JavaScript**

**JavaScript** cuando ejecuta código lo hace de manera **síncrona**, pero una forma que tiene de ejecutar código **asíncrono**es con las funciones llamadas *“Callbacks”*  
Éstas se ejecutan y nos devuelven el proceso, generalmente después de algún evento de tiempo, al completarse un request o al terminar de leer un archivo.

**Recuerda:**  
-El tiempo que le demos a los procesos es un tiempo mínimo. **JavaScript** puede no respetarlos, todo depende del peso de las tareas que tengamos en la cola de ejecución.  
-Para evitar los cuellos de botella en los procesos y delega tareas pesadas y con mucho tiempo de ejecución a otros microservicios.

**Callback a un servidor externo**

En este vídeo aprenderás a hacer *“Callbacks”* desde una API externa.

Notas:  
Existen otras formas diferentes de llamar a los *“Callbacks”* en el código de **JavaScript**.

* cb
* onfinish

**Promesas**

Las **Promesas** son objetos y están asociadas con alguna tarea *asincrónica*.

Existen tres estados internos de las **Promesas**:

1. Pending
2. Fullfiled
3. Rejected

El **constructor Promise** tiene dos parámetros:

**const** promise = **new** Promise(**function** (resolve, reject) {

)}

//Promise es un constructor que recibe una función con dos argumentos.

**Implementación de set interval en JavaScript**

Con *setInterval* puedes ejecutar código cada cierto tiempo, un buen ejemplo de setInterval es el contador que aparece entre video y video que te dice: El siguiente video se reproducirá en 5 segundos… 4 segundos…

En JS tenemos el método setInterval que nos permite hacer esto.

El reto que te voy a dejar en ésta clase se llama pluralización, lo que debes hacer es que cuando haya pasado un segundo el mensaje diga HA PASADO UN SEGUNDO, pero cuando corresponda a mas de un segundo diga HAN PASADO x SEGUNDOS.

**Funciones Recursivas**

Las funciones recursivas son funciones que se llaman a si mismas, si, el resultado de la función es el llamado de la misma función.

Este comportamiento es muy importarte para solucionar cosas como series o algoritmos que tomen en cuenta valores pasados.

La recursividad se compone de 2 partes indispensables:

A) Un caso base  
B) El llamado de la misma función dentro de la función

Sin un caso base, la función se va a llamar infinitamente. Esto quedará mas claro con el ejemplo de serie de fibonacci.

**Memoizacion**

Anteriormente hablamos de recursividad y de sus casos de uso, pero: ¿Qué significa esto a nivel de rendimiento?.

Por ejemplo, si tomamos el fibonacci del numero 20, la función se llama 13530 veces. Esto significa que sea un proceso que tardará mucho en ejecutarse y eso no es convenientemente, para solucionarlo existe algo llamado memoizacion.

Memoizacion es guardar temporalmente valores que ya hemos calculado anteriormente. Y tiene sentido ya que en la aplicación que estamos proponiendo se toma como referencia valores pasados, por ejemplo:

* El Fibonacci del numero 6 es la suma del fibonacci de 5 y del fibonacci de 4, y anteriormente el fibonacci de 4 esta dado por el fibonacci de 3 mas el fibonacci de 2.

Aplicando mamoizacion a nuestro proyecto, logramos pasar de 13530 llamados a tan solo 38.

**Iteradores en JavaScript**

**Solucionaremos el problema de Fibonacci con Iteradores**

**Los iteradores** nos permitirán hacer **listas infinitas de elementos** haciéndolos distintos a los arreglos (arrays) los cuales tienen un numero finito de elementos definidos

Para los iteradores podemos obtener lo siguiente:

* **next()** - iterará los datos
* **value** - nos devolverá el valor del dato
* **done** - será un indicador para cuando la lista se haya terminado

Con iteradores es muy sencillo realizar un for y obtener los datos.

**for(let value of fibo) {**  
console.log(value)  
**}**

**Generadores en JavaScript**

Para decirle a JavaScript que nuestra función es un **generador** debemos indicarlo con un asterisco de la siguiente forma:

## function(asterisco)

Si creamos un generador debemos colocar la palabra clave **yield** la cual indica que cuando llamemos la función después de la primera vez, **esta iniciará en la línea después de yield.**

**Estructuras de Datos Inmutables en JavaScript**

## La inmutabilidad de los objetos es algo muy importante para la programación funcional.

El operador **===** nos ayuda a comparar objetos, ejecutando la comparación no directamente a los datos del objeto sino, a la referencia del objeto.

**Cuando asignamos un objeto a otro** estamos haciendo que ambos apunten a la misma referencia, por eso **al modificar un objeto el otro también se verá afectado**, porque **ambos tienen la misma referencia** de memoria.

La **inmutabilidad tiene por objetivo** hacer que los parámetros de un objeto sean **no modificables** o **inmutables**

Si quieres aprender más sobre inmutabilidad de objetos en JavaScript en la sección de Enlaces dejamos el link de una **librería diseñada por Facebook** que nos permite crear distintos tipos de objetos inmutables

**Requisitos Técnicos**

**¿Qué es NPM?**

NPM es un package manager para JavaScript. Eso quiere decir que es un lugar donde están guardas aquellas librerías o dependencias que usaremos en nuestros proyectos, y de hecho, tu puedes crear y compartir las tuyas.

¿Qué tipo de dependencias?, muchas, tanto de backend y front-end que utilicen JavaScript, quizás unas ya las conoces, las puedes consultar las más populares en:

<https://www.npmjs.com/>

**¿Qué necesito para publicar un módulo de NPM?**

Lo primero es tener una cuenta en el sitio, solo necesitas correo y contraseña, lo obtienes en:

<https://www.npmjs.com/signup>

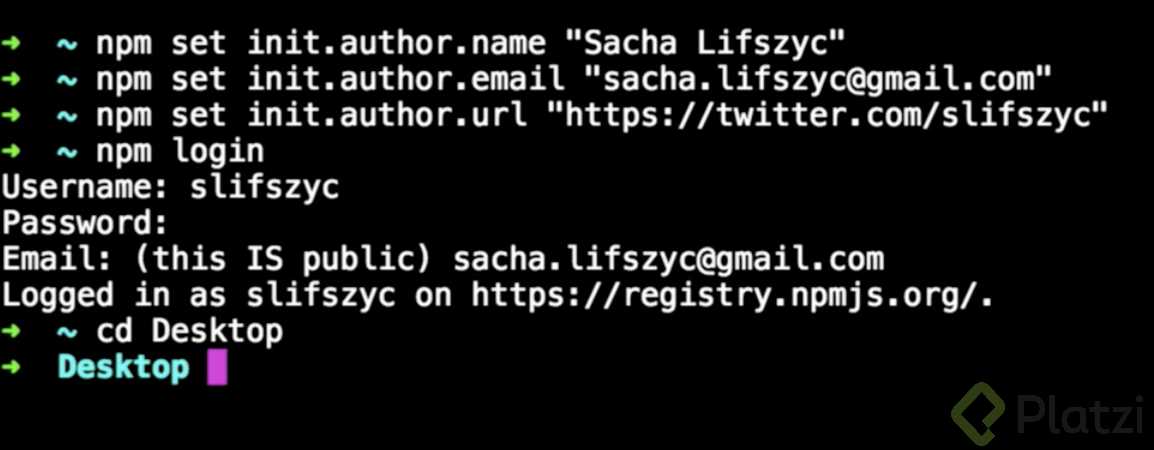
Lo siguiente es tener instalado Node.js, ya que entre otras cosas, es la plataforma que nos ayuda a tener instaladas las herramientas de NPM en nuestra terminal. Si aún no lo tienes, lo puedes hacer desde:

<https://nodejs.org/en/download/>

Teniendo cuenta lo anterior, tenemos que decirle a NPM quienes somos, para ello, necesitamos pasar unos datos desde la terminal:

* npm set [init.author.name](http://init.author.name/) “Tu nombre”
* npm set init.author.email “Tu correo”
* npm set init.author.url “Una url de tu sitio, inclusive puede ser una red social”

De inmediato ejecutamos npm login y agregaremos las credenciales que obtuvimos en el sitio. Si alguno de nuestros datos no es correcto, se va a mostrar en la terminal. Si todo está bien, se vera algo como lo siguiente:



Y con esto ya tenemos todo listo para publicar nuestro módulo.

**Creando nuestro paquete**

Para que NPM entienda que es nuestro proyecto, necesitamos configurar una serie de parámetros, para ello, en la carpeta donde vamos a iniciar nuestro proyecto, utilizamos el comando:

npm init

A partir de ese momento, estaremos en un tipo formulario en el cual se podrán agregar una serie de parámetros, los cuales son:

Nombre: “ Aquí pones el nombre del paquete “

Version: “ Aquí agregas la versión de tu librería, es recomendado que si es la primera iteración sea la versión 0.1.0, si el producto ya esta completo, tienes test y de ahi no va a cambiar algo usamos 1.0.0, si son cambios mas grandes que rompen lo que teníamos seria un cambio a la version 2.0.0, etc. Esto se llama en software semantic versions, puedes leer más en <http://semver.org/>"

Description: “Aquí es donde vamos a explicar que hace nuestro modulo, puedes ser tan extenso como sea necesario, esto es lo que va a leer la comunidad”

Entry point: “Es el archivo principal de nuestra librería, por defecto es index.js”

Test command: “Si nosotros tenemos test, aquí declaramos el comando en el cual podemos ejecutar este test”

Git repository: “Aqui agregamos la URL del repositorio de Github en donde estará el código de nuestro proyecto, no es obligatorio pero es recomendado para que otras personas puedan colaborar en tu proyecto de ser necesario”

License: Una de las partes mas importantes y que no deberías omitir, es básicamente donde indicamos que tenemos todos los derechos intelectuales del software y que pueden usar nuestra librería de acuerdo a ciertos términos y condiciones, acá la opción recomendada es MIT.

Una vez que pasamos por todos estos parámetros, notaras que en la carpeta tendremos un nuevo archivo llamado package.jsno, el cuál contiene toda la información que acabamos de llegar y que podemos modificar en caso de ser necesario.

El archivo Readme

Nosotros podemos documentar nuestro plugin haciendo uso de [README.md](http://readme.md/)

Este archivo será lo primero que leerán tus usuarios al no entender algo de tu proyecto. No olvides que sera necesario usar markdown, lo cual tienes una guía aquí mismo:

<https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet>

**Escribiendo el código de nuestro paquete**

Uno de los “Refactor” más importantes que debes de tener en cuenta antes de publicar nuestro modulo es la forma en que exportamos las funciones. Se tienen 2 opciones:

**export** **default** **function** **platzom**(str) {

}

De este modo, al momento de importar la librería en otro proyecto sera de la forma:

**import** platzom **from** ‘platzom\'

Y cuando tenemos varias funciones que queremos importar, usamos solo export:

**export** **function** **platzom**(str) {

}

Así para usar el import lo hacemos del modo:

**Import** {platzom} **from** **platzom**

Lo siguiente es preparar el código para que sea compatible con cualquier navegador, para lo cual usamos babel. En este caso, usamos la configuración por defecto:

<https://babeljs.io/docs/setup/#installation>

**Testeando el paquete**

El testing puede resultar algo confuso en un inicio: ¿Para qué quisiéramos probar algo que nosotros hicimos?. Es justo eso, los humanos no somos perfectos y necesitamos tener una serie de validaciones adicionales para asegurarnos que todo saldrá como lo esperamos.

Para hacer test se utilizan 2 herramientas muy populares: Mocha y Chai.

npm install --save-dev mocha chai

Los test prácticamente funcionan mediante resultado que esperamos, en la primer parte podemos describir que hace el test y lo segundo el valor esperado.

por ejemplo:

it(“Si **la** palabra termina **con** **ar**, **se** **le** quitan esas 2 letras.”,

const tanslation = platzom(“Programar”)

expect(translation).**to**.equal(“Program")

)

Y así con cada valor que esperamos.

**Publicando el paquete en NPM**

Lo primero que debes de tener en cuenta es que necesitas subir tu proyecto a Github. Esto no es obligatorio pero es importante para la comunidad que usara tu plugin.

Ahora para subir tu proyecto a NPM usas el comando

npm publish

**Inicializando el juego**

Con el teclado virtual que hicimos capturaremos el evento **keydown** llamada a través de la función ***document.addEventListener(‘keydown’)***

Cada tecla de nuestro teclado físico tiene un código asignado, con la función anterior obtendremos el keyCode de cada tecla para después asociarlo a nuestro teclado virtual con la propiedad **data-key** de la etiqueta *div*.

La función ***docment.querySelector(’[data-key]="${keyCode}"’)*** nos devolverá las etiquetas que poseen la propiedad **data-key**. Por lo tanto crearemos una función propia llamada *getElementByKeycode* que reciba el keycode asignado para que filtre y nos devuelva la etiqueta que tenga la propiedad data-key con ese keycode.

A continuación haremos una función que trabajará sobre los estilos css y mostrará activa, presionada o fallada la tecla presionada de nuestro teclado virtual.

**Y podemos llamarla así:**

activate(65);  
activate(65, { success: true});  
activate(65, { fail: true});

Con esto la tecla presionada adoptará el estilo css especificado.

**Para añadir un toque genial** a nuestro teclado pondremos un Time Out para desactivar la tecla y regresarla a su estilo inicial, esto nos dará un efecto de **keyDown** y **keyUp**.

Por último haremos dos funciones más que se enfocarán en la lógica de nuestro juego. Nos ayuden a definir aleatoriamente las teclas que el usuario debe presionar tras saber su nivel recuerda que esto debe suceder cada vez que se inicie un nuevo juego.

**A una ronda le sigue otra ronda**

## ****Crearemos la lógica de nuestro juego, validemos si el usuario adivina o no y actuemos sobre eso, todo esto sobre la primer ronda.****

Crearemos la función **siguienteNivel** la cual recibe el nivel actual esta **valida si el juego ha terminado, si va a comenzar o si esta en algún nivel específico.** Además llamando a nuestra función activate mostraremos las teclas que debe seguir el usuario añadiéndole un estilo particular.

Recuerda que **el procesador de nuestra computadora ejecuta muy rápido las instrucciones de código** por lo tanto pondremos un **Time Out para que las teclas a seguir se vayan mostrando con un Delay** o retardo.

Después **con nuestra función onkeydown** validaremos si el usuario presionó la tecla correcta o incorrecta, dependiendo del resultado asignaremos el estilo adecuado.