

# Agilizar los procesos de las aplicaciones web

*"Todos deberían aprender a programar, les enseña a estructurar el pensamiento."*

Steve Jobs— Cofundador de Apple y cofundador de Pixar

¿Con cuántos retos te encuentras día a día? A medida que avanzas en este camino como developer algo es seguro: ¡tendrás el desafío de desarrollar programas de mayor complejidad! [MDN](#), la plataforma de aprendizaje para las tecnologías web más consultada por developers en la actualidad, lo explica así: *"...cada vez más una página web hace más cosas que sólo mostrar información estática..."*. Hoy, por ejemplo, nos permite interactuar con mapas o animaciones gráficas 2D/3D.



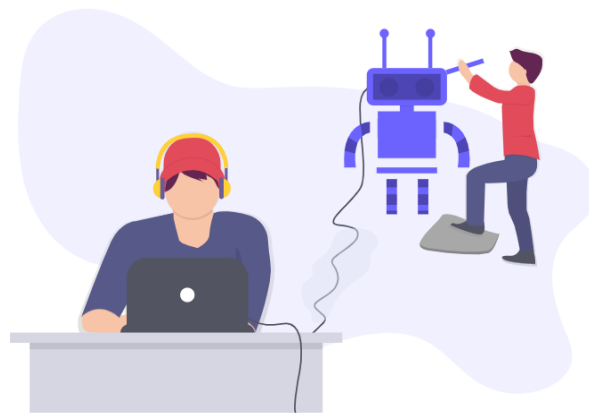
¿Cómo podemos hacer para abordar estos desafíos continuos propios de nuestro rol? Según Yeison Daza en su [artículo](#) *"...debemos dividir estos [desafíos complejos] en problemas pequeños que podamos resolver e implementar, luego componemos estas soluciones. Esto es lo que hacemos cada día como programadores..."*.

Esto significa que a medida que se complejizan las tareas, las vamos a simplificar en pequeñas porciones de código que podamos reutilizar, ¡Este es nuestro superpoder! Convertirlos en varios pasos más sencillos.

En esta toolbox ampliarás tus conocimientos sobre JavaScript e incorporarás un concepto nuevo: las **funciones**, que son la materia prima para crear programas complejos y garantizar su mantenimiento.

## ¡Que comience la función!

Una **función** es la definición del procedimiento de una o más sentencias que realizan una tarea, es decir, una serie de instrucciones ejecutadas en un mismo proceso que se definen una vez, y luego pueden ser invocadas una y otra vez sin necesidad de redefinir los pasos. ¿Muy técnico el concepto? Bien, tomemos un enfoque más práctico. Te presentamos a **chef-bot**.



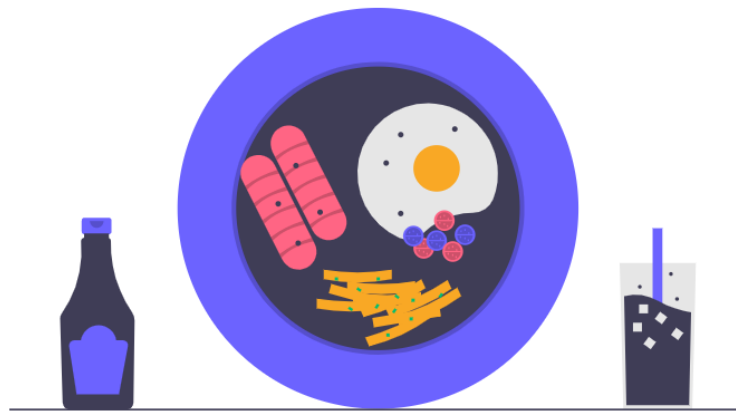
El **chef-bot** es un robot de última tecnología que puede ser programado para hacer tareas de cocina de forma muy veloz. No sabemos si este robot existe en la vida real, pero estamos de acuerdo que sería una idea fantástica, ¿cierto?

Imagina que podemos programar a este robot con Javascript, y para estrenarlo, queremos codear una receta básica: el pan tostado. Para alcanzar tal receta, debemos diseñar un algoritmo que pudiera describir los siguientes pasos:

- *Sacar una rebanada de la bolsa*
- *Prender el fuego de la cocina*
- *Buscar un sartén o asadera y colocarla encima del fuego*
- *Esperar que caliente el sartén*
- *Colocar la rebanada de pan sobre el sartén*
- *Pasado un tiempo, verificar si la cara del pan que da al sartén se tuesta*
- *Si se tuesta, darle la vuelta*
- *Si la superficie está tostada, servir el pan en un plato*
- *Si no se tuesta, esperar un rato y volver a revisar*

Con el algoritmo anterior, y mediante la ayuda de Javascript, pudiéramos programar a **chef-bot** para que realice la fácil tarea de tostar un pan, pero, ¿cómo podemos hacer para que nuestro robot tueste más de una rebanada de pan?, y también, ¿cómo podemos hacer para que pueda repetir esto varias veces al día? Quizás pudiéramos prenderlo y apagarlo para que ejecute la misma tarea de la misma forma, pero debe existir una forma más eficiente de hacer esto. La respuesta a estas preguntas: con **funciones**.

Una **función** de Javascript nos permite definir y guardar una serie de pasos para ejecutar tantas veces queramos durante la ejecución de una aplicación. Para el ejemplo anterior, pudiéramos **definir** la función `tostarPan()` la cual va a contener el algoritmo correspondiente, y ahora cuando queremos que **chef-bot** haga la tarea por nosotros, sólo tendríamos que **invocar** la función y esperar pacientemente por nuestro desayuno.



Entonces, **las funciones nos permiten ordenar, categorizar y evitar repetir una y otra vez código en nuestros scripts**. Sin ella, un software más complejo no sería posible de desarrollar ya que contaría con infinitas líneas de código desordenadas. ¡Por esto las queremos! (y queremos que aprendas a usarlas a tu favor).

Ahora veamos un ejemplo más cercano a lo que hemos venido estudiando en este sprint. Supongamos que queremos hacer un programa que muestre por consola si una persona es mayor de edad (considerando la mayoría de edad como un número mayor a 18), podemos empezar con esta aproximación:

...

```
let edad = 20;
if(edad > 18) {
  console.log("la persona es mayor de edad");
}
```



```

} else {
    console.log("la persona no es mayor de edad");
};
...

```

El código de arriba funciona perfecto y sin errores, pero ¿qué pasa si este código debe ser utilizado en otras partes de la aplicación? Naturalmente podemos copiar/pegar las líneas anteriores en los archivos que se necesite, sin embargo, esto causa duplicidad del código, lo cual se considera una mala práctica.

Así como en CSS se definen clases globales que agrupan un conjunto de propiedades en común para poder ser reutilizadas, con las funciones de JavaScript podemos crear procedimientos que se definen sólo **una vez** para luego invocarlas donde las necesitemos. Si queremos escribir el ejemplo anterior como una función de Javascript, podemos hacer lo siguiente:

```

...

let edad = 20;
// definimos la función
function verificarMayoria() {
    if(edad > 18) {
        console.log("la persona es mayor de edad");
    } else {
        console.log("la persona no es mayor de edad");
    };
};
// invocamos la función
verificarMayoria();
...

```

¡Perfecto! Hemos creado una función que verifica la mayoría de edad. Pero existe un problema aún, ¿cómo le decimos a la función que verifica un valor de edad en específico?

Podemos declarar este valor fuera de la función, pero imagina que la aplicación sigue creciendo y agregamos más variables. ¡Pronto nos encontraremos con que no podremos usar el nombre `edad` para ninguna otra variable porque esta función la estás utilizando ya! Si existiera otra variable con el mismo nombre, pero con otro valor, nuestra función quizás no imprimirá el resultado correcto.



Podemos colocar un comentario en el código o avisar a nuestrxs compañerxs que no usen el nombre de la variable, pero, ¿te parece una buena forma de construir una aplicación? En cuestión de días estaríamos dejando carteles por toda la oficina diciendo qué palabras no podemos usar.



Ante esa situación, ¡los **parámetros** han venido a salvarnos! ¿Notaste los paréntesis que se escriben luego del nombre de la función, en su **definición**? Las funciones ofrecen la posibilidad de definir parámetros, los cuales actúan como recipientes vacíos que se llenarán durante la **invocación** de la función con un valor real. Por ejemplo:

```
...
```

```
// definimos la función con un parámetro
```

```
function verificarMayoria(edad) {
```

```
  if(edad > 18) {
```

```
    console.log("la persona es mayor de edad");
```

```
  } else {
```

```
    console.log("la persona no es mayor de edad");
```

```
  }
```

```
};
```

```
// invocamos la función
```

```
let algunaEdad = 20;
```

```
verificarMayoria(algunaEdad);
```

```
...
```

En el ejemplo anterior, el parámetro `edad` es una variable especial que solo tiene validez dentro del cuerpo de la función, y no se mezcla con nada de lo que esté fuera de ella. Esto es gracias al concepto de **contextos**, el cual explicaremos más adelante en esta toolbox.

Las funciones hacen más que imprimir un mensaje por consola: también pueden **retornar** un valor, utilizando la palabra reservada `return`. Si aplicamos esto a la definición anterior, podemos tener lo siguiente:

```
...  
  
function verificarMayoria(edad) {  
  if(edad > 18) {  
    return true; // es mayor de edad  
  } else {  
    return false; // no es mayor de edad  
  }  
};  
  
// invocamos la función  
let algunaEdad;  
let esMayor = verificarMayoria(algunaEdad);  
...
```

En el ejemplo anterior, la variable `esMayor` ahora sostiene el resultado que retorna la función `verificarMayoria`, valor que podemos utilizar en cualquier lugar de nuestra aplicación, según lo permita el contexto. Este patrón nos da mucha flexibilidad a la hora de trabajar con funciones.

## Tres pasos con funciones

Lo anterior lo podemos resumir en tres pasos:

### 1. 📌 Definir

Para usar una función primero necesitas definirla en algún lugar desde el cual luego la vas a llamar. Existen varias formas de definir una función en JavaScript, y en esta toolbox te hemos contado de la **función declarada**. En próximas meetings te estaremos contando de otras formas de definir funciones.



## 2. 📦 Parametrizar (opcional)

A las funciones también les puedes enviar valores para que estén disponibles y puedan ser utilizados dentro de ellas. A estos valores los llamamos **parámetros**. **Se puede definir más de un parámetro ⚡** y de cualquier tipo: números, strings, booleanos, inclusive, ¡otras funciones! Aprenderás más de esto en las próximas meetingss.

## 3. ➡️ Invocar

Para invocar a una función hay que escribir en el código el nombre que definiste y opcionalmente, enviar los parámetros que indicaste en su definición. Con la palabra reservada ``return`` la función podrá devolver un valor.

# Contexto

El **contexto** puede definirse como el alcance que tendrá la variable dentro de la aplicación, y decide a qué variables tiene acceso en cada parte del código.

Existen dos tipos de contextos: local o global.

- Las variables con **contexto local** son las creadas dentro de una función que sólo pueden ser accedidas desde su propia función o dentro de funciones anidadas. Así como lo leíste, en JavaScript es posible declarar funciones dentro de funciones.
- Las variables con **contexto global** son a las que puedes acceder desde cualquier parte del código. ¡Utilízalas con cuidado!

# Resumiendo

¡Llevas hecho un gran recorrido! Te has metido de lleno en el concepto de funciones, y has aprendido sobre el funcionamiento de las variables ``const`` y ``let``. Poco a poco vamos conectando los puntos que involucra entender el arte de programar.

Recuerda que estos conceptos te acompañarán a lo largo de este sprint y son claves para el aprendizaje de Javascript, por lo que tener claro estos conceptos




tanto desde lo conceptual como desde lo práctico es absolutamente fundamental.

[Nos tomamos unos minutos para completar esta encuesta.](#) Queremos saber cómo valoran mi tarea hasta acá. ¡Les va a llevar solo un minuto!

## ¡Prepárate para la próxima meeting!


### Challenge

 1. Escribe la función `cualEsMayor()`, la cual muestra por consola cuál es el mayor de 2 números. **La función no debe retornar nada**, solo utilizar el infaltable ``console.log()`` para indicar cuál de los números es mayor. Se espera que invocando la función de esta forma:

```
cualEsMayor(8, 20)
```

Se muestre por consola:

```
"El número 20 es mayor que el número 8".
```

 2. Ahora, escribe la función `cualEsMenor()` la cual deberá **retornar** el menor de 3 números dados. Se espera esta invocación:

```
let numeroMenor = cualEsMenor(5, 7, 2)
```

Y al hacer `console.log` sobre la variable `numeroMenor`, se deberá mostrar:

```
2
```