**¿Qué significa ser un profesional de JavaScript?**

‌

El camino para llegar a ser profesional es largo y duro, no es fácil. Todos necesitamos que nos guíen para saber qué hacer y qué no. Este camino es conocido como la ruta de pasar de **Junior** a **Senior**, *este es un duro camino lleno de experiencia*.

‌

**¿Qué forma a un profesional?**

‌

Te presento una lista de estas cosas que lo forman:

‌

1. Conocimiento del lenguaje.
2. Conocimiento de entornos de programación.
3. Mejores prácticas.
4. Versado en código.
5. Herramientas.
6. Ética / Profesionalismo.
7. Experiencia.

‌

**El lenguaje: JavaScript**

‌

Debemos tener muy claro cuales son los **fundamentos** de JavaScript antes de comenzar con esto. Existen *features* muy raros y hay que estudiarlos. Tenemos que saber cómo **funcionan las cosas** en JavaScript.

‌

**No fundamentos**

‌

Los **no fundamentos**" representan las siguientes características del lenguaje:

‌

1. **Promesas (nivel pro)**.
2. **Getters, setters**: son formas de obtener valor de una variable sin tener que poner this.name.
3. **Proxies**: es un *feature* muy raro, pero que más adelante veremos a profundidad. Sirve para interceptar a una función antes de que se ejecute.
4. **Generadores**: esto es raro, pero vamos a ver que sí es eficiente.

‌

**¿Cómo funciona?**

‌

Este lenguaje corre sobre un motor. JavaScript no contiene clases como otros lenguajes de programación, esto es algo que vuela mucho la cabeza, es muy difícil de entender. Otro *feature* muy *cool* que vamos a aprender es event loop, es lo que permite que pueda correr muchos procesos a la vez.

‌

**Entornos de programación**

‌

Cuando estamos desarrollando lo hacemos para la **WEB**, para un celular, para seguidores. Existen diferentes entornos que nos ofrecen **APIS**, tenemos que conocer todo esto. V

‌

**Versado en código**

‌

Esto quiere decir que tenemos que leer mucho código, un lugar hermoso para ponernos a leer código es **GitHub**. Debemos leer mucho y hacerlo de forma muy constante.

‌

**Mejores prácticas**

‌

No vamos a reinventar la rueda, hay muchas personas que ya han solucionado los problemas más comunes, tenemos que usar estas soluciones, a estas soluciones se les llama: **patrones de diseño**.

‌

**Ética**

‌

Esta es la parte más importante de ser un profesional. Un buen profesional cumple con los siguientes valores:

‌

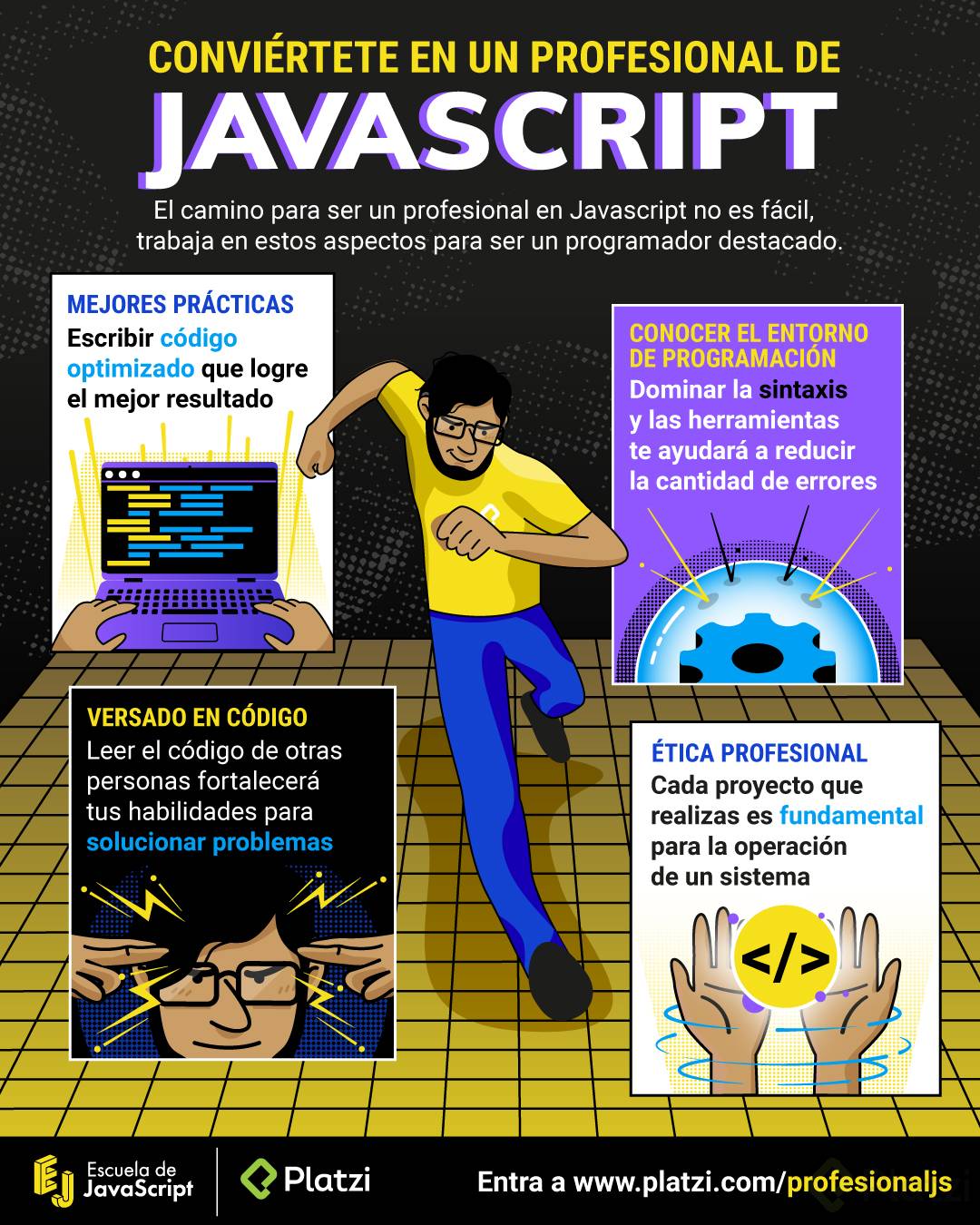
1. **Es responsable**.
2. **Entrega a tiempo sus trabajos**.
3. **Sabe decir que no**.
4. **No hace daño**.

‌

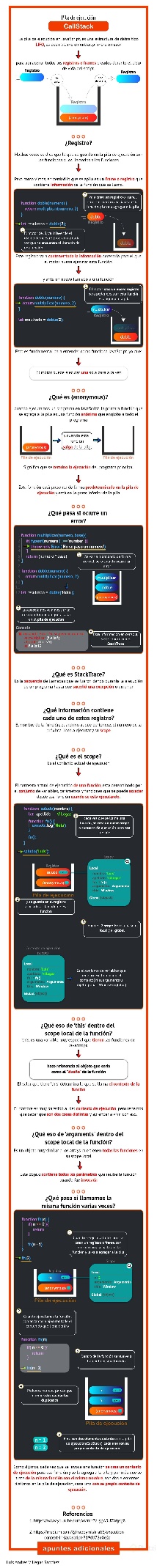
**Experiencia**

‌

La experiencia no es algo que se pueda enseñar, tenemos que encontrarla nosotros mismos en el camino a ser profesionales. Todo está en nosotros, tenemos que estudiar y practicar mucho.







El **Scope** o ámbito es lo que define el tiempo de vida de una variable, en que partes de nuestro código pueden ser usadas.

### Global Scope

Variables disponibles de forma global se usa la palabra var, son accesibles por todos los scripts que se cargan en la página. Aquí hay mucho riesgo de sobreescritura.

### Function Scope

Variables declaradas dentro de una función sólo visibles dentro de ella misma (incluyendo los argumentos que se pasan a la función).

### Block Scope

Variables definidas dentro de un bloque, por ejemplo variables declaradas dentro un loop while o for. Se usa let y const para declarar este tipo de variables.

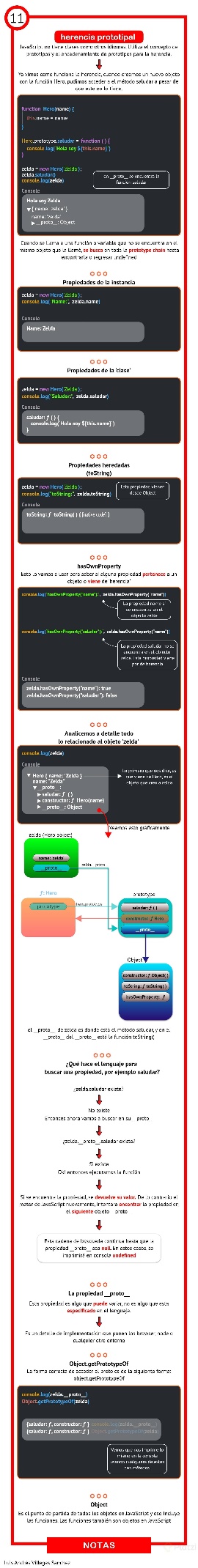
### Module Scope

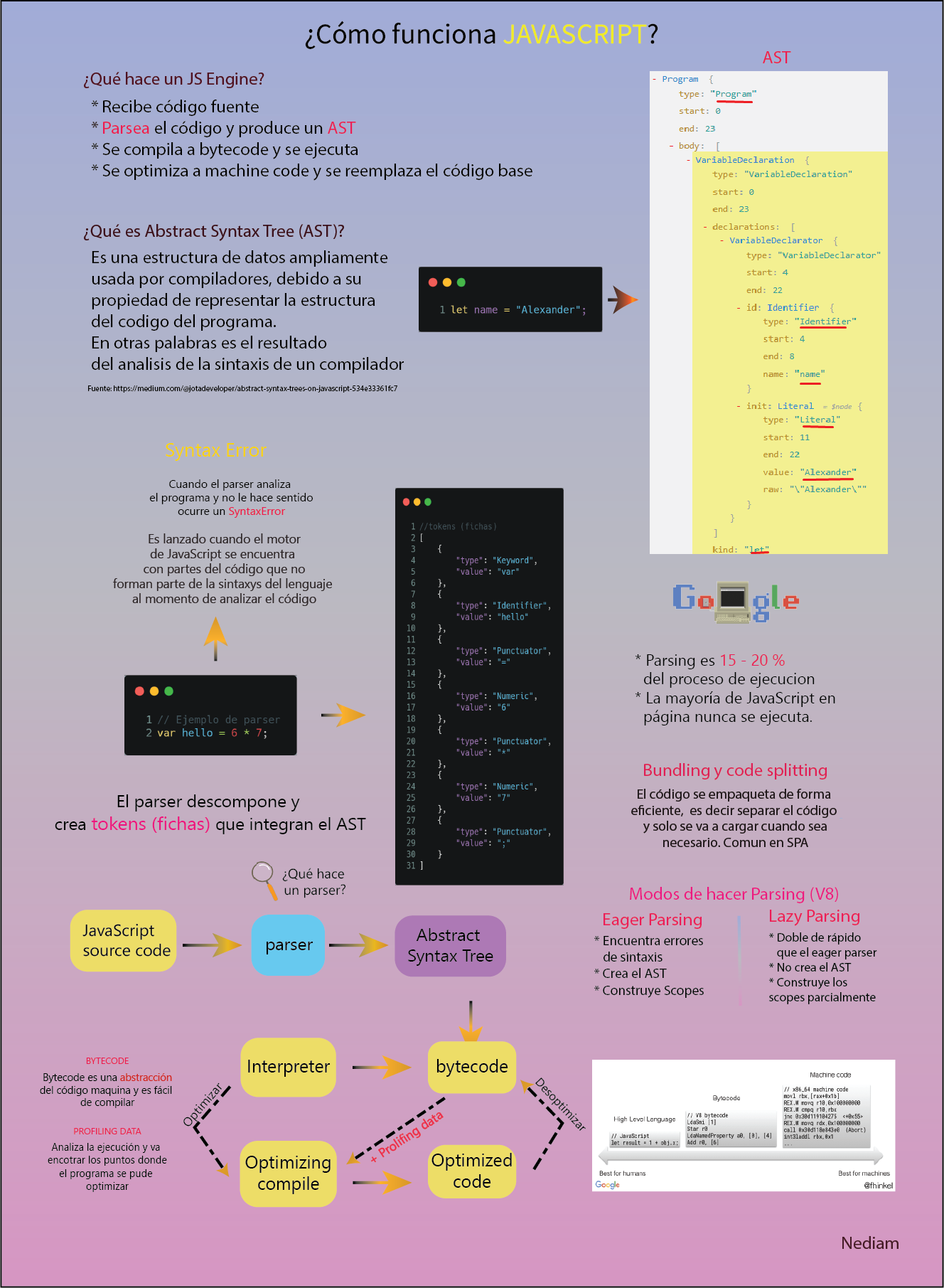
Cuando se denota un script de tipo module con el atributo type="module las variables son limitadas al archivo en el que están declaradas.

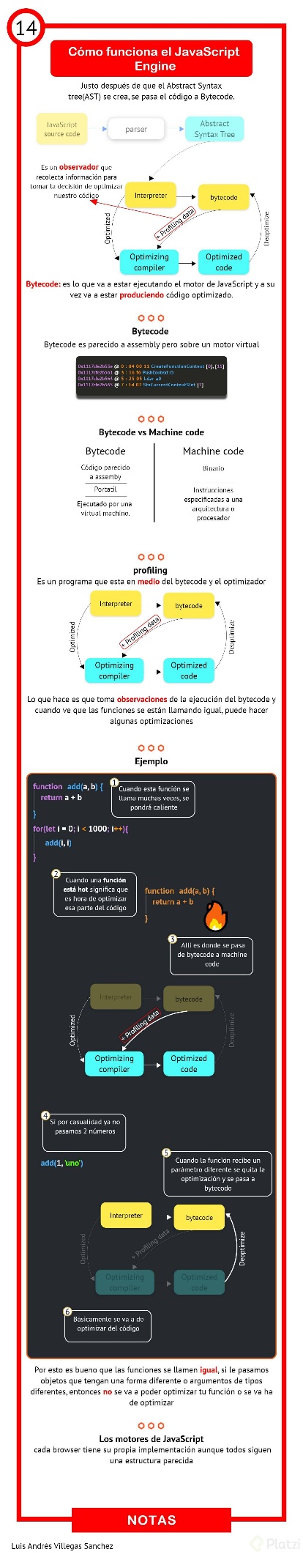
Una función IFEE es una función que se ejecuta inmediatamente y contienen una función anónima con alcance léxico encerrado por el operador de operación () y la expresión de función cuya ejecución es inmediata. Una función anónima es una función que se ha definido sin un nombre. La función anónima se almacena en la memoria, pero el tiempo de ejecución no crea automáticamente.

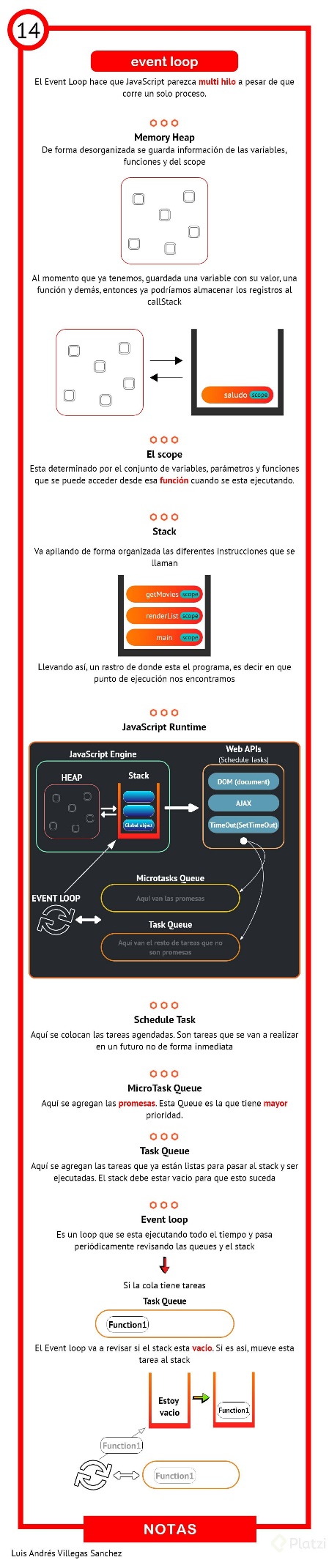
Son funciones que regresan una función o un objeto con funciones que mantienen las variables que fueron declaradas fuera de su scope.

Los **closures** nos sirven para tener algo parecido a variables privadas, característica que no tiene JavaScript por *default*. Es decir encapsulan variables que no pueden ser modificadas directamente por otros objetos, sólo por funciones pertenecientes al mismo.

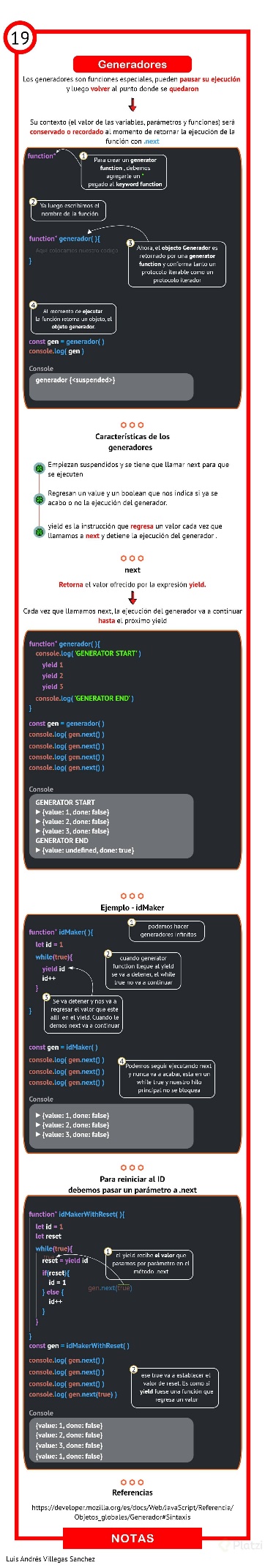


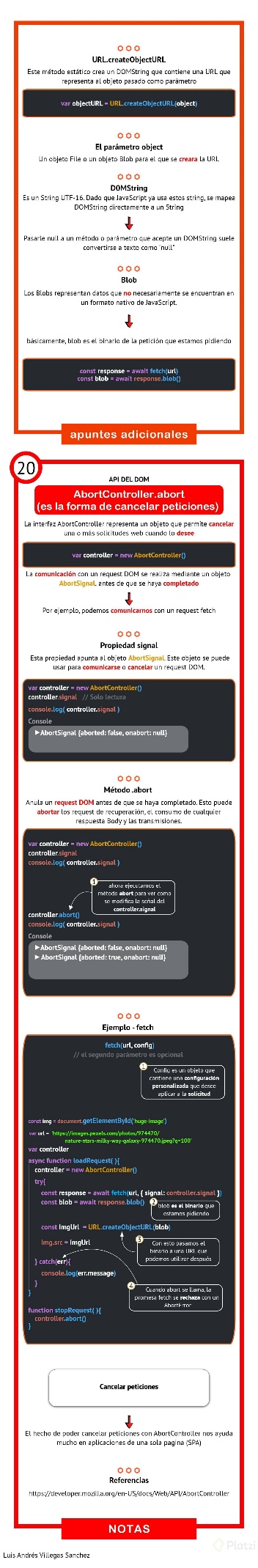


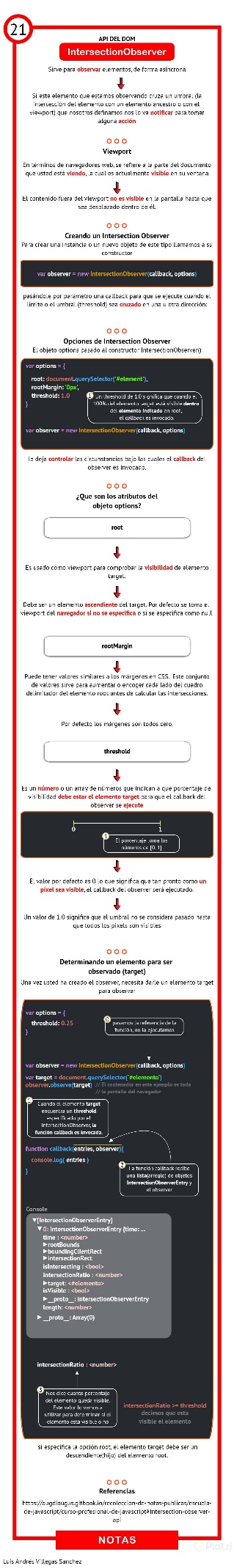












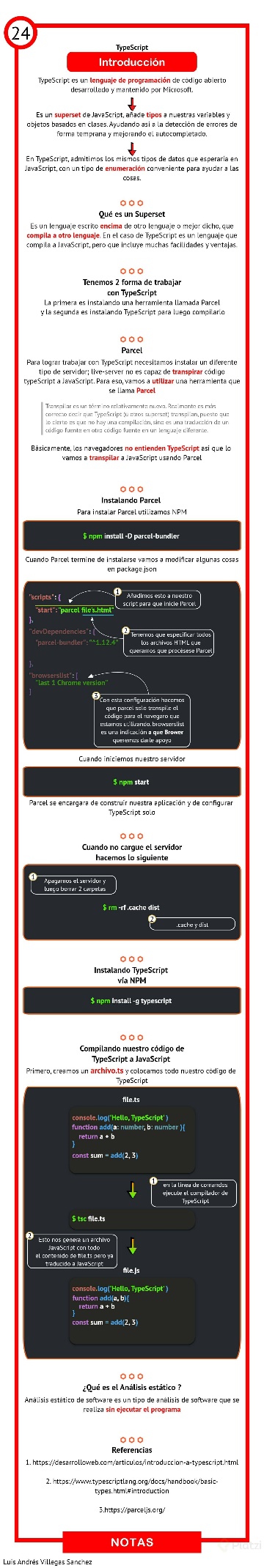


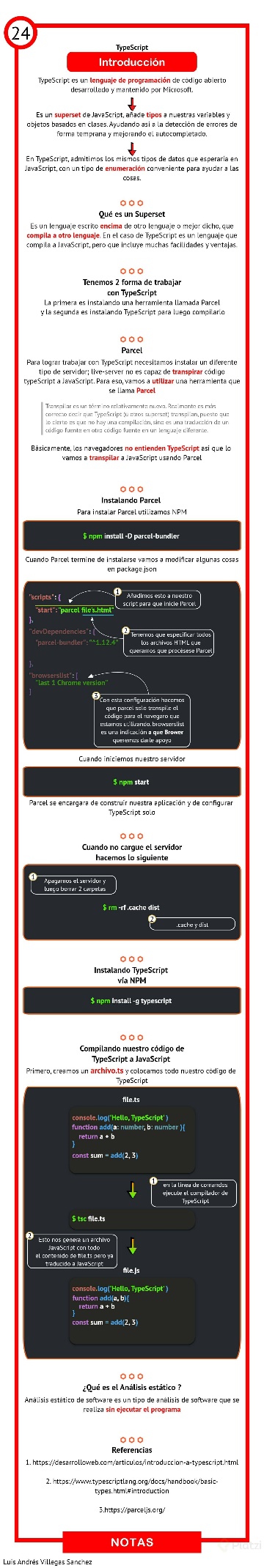
# Service Workers

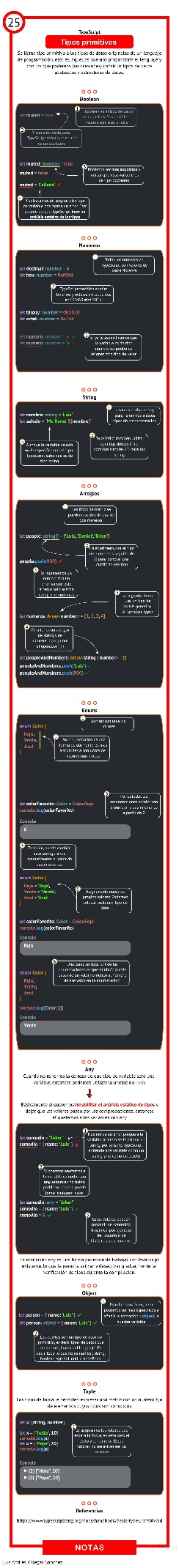
Sirven para hacer que nuestras aplicaciones funcionen Offline.

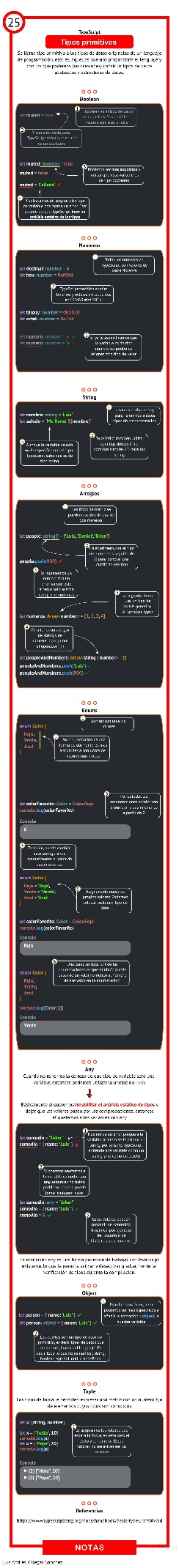
Muy usados en las **Progressive Web Apps** (PWA) los ServiceWorkers son una capa que vive entre el navegador y el Internet.

Parecido a como lo hacen los proxys van a interceptar peticiones para guardar el resultado en cache y la próxima vez que se haga la petición tomar del cache ese resultado.









**Patrón de diseño**

[Ir a la navegación](https://es.wikipedia.org/wiki/Patr%C3%B3n_de_dise%C3%B1o#mw-head)[Ir a la búsqueda](https://es.wikipedia.org/wiki/Patr%C3%B3n_de_dise%C3%B1o#p-search)

Los **patrones de diseño** son unas técnicas para resolver problemas comunes en el [desarrollo de *software*](https://es.wikipedia.org/wiki/Desarrollo_de_software) y otros ámbitos referentes al diseño de interacción o interfaces.

Un patrón de diseño resulta ser una solución a un problema de diseño. Para que una solución sea considerada un patrón debe poseer ciertas características. Una de ellas es que debe haber comprobado su efectividad resolviendo problemas similares en ocasiones anteriores. Otra es que debe ser reutilizable, lo que significa que es aplicable a diferentes problemas de diseño en distintas circunstancias.[1](https://es.wikipedia.org/wiki/Patr%C3%B3n_de_dise%C3%B1o#cite_note-1)​

**Objetivos de los patrones**

Los patrones de diseño pretenden:

* Proporcionar catálogos de elementos reusables en el diseño de sistemas software.
* Evitar la reiteración en la búsqueda de soluciones a problemas ya conocidos y solucionados anteriormente.
* Formalizar un vocabulario común entre diseñadores.
* Estandarizar el modo en que se realiza el diseño.
* Facilitar el aprendizaje de las nuevas generaciones de diseñadores condensando conocimiento ya existente.

Asimismo, no pretenden:

* Imponer ciertas alternativas de diseño frente a otras.
* Eliminar la creatividad inherente al proceso de diseño.

No es obligatorio utilizar los patrones, solo es aconsejable en el caso de tener el mismo problema o similar que soluciona el patrón, siempre teniendo en cuenta que en un caso particular puede no ser aplicable. “Abusar o forzar el uso de los patrones puede ser un error”.

**Volver a aprender el diseño CSS**

Si te encuentras luchando con el diseño CSS, es probable que estés tomando decisiones sobre los navegadores que deberían tomar ellos mismos. A través de una serie de diseños simples y [componibles](https://every-layout.dev/rudiments/composition) , **cada diseño** le enseñará cómo aprovechar mejor los algoritmos integrados que potencian los navegadores y CSS.

Emplear un diseño de diseño algorítmico significa eliminar los @mediapuntos de interrupción, los " [números mágicos](https://css-tricks.com/magic-numbers-in-css/) " y otros trucos para crear componentes de diseño independientes del contexto. Sus futuros sistemas de diseño serán más consistentes, tendrán menos código y serán más maleables en manos de sus usuarios y sus dispositivos.

**Historia[**[**editar**](https://es.wikipedia.org/w/index.php?title=Patr%C3%B3n_de_dise%C3%B1o&action=edit&section=1)**]**

En 1979 el [arquitecto](https://es.wikipedia.org/wiki/Arquitecto) [Christopher Alexander](https://es.wikipedia.org/wiki/Christopher_Alexander) aportó al mundo de la arquitectura el libro *The Timeless Way of Building*; en él proponía el aprendizaje y uso de una serie de patrones para la construcción de edificios de una mayor calidad, en la que esa mayor calidad se refería a la arquitectura antigua y la menor calidad correspondía a la arquitectura moderna, que el romper con la arquitectura antigua había perdido esa conexión con lo que las personas consideraban que era calidad.

En palabras de este autor, "Cada patrón describe un problema que ocurre infinidad de veces en nuestro entorno, así como la solución al mismo, de tal modo que podemos utilizar esta solución un millón de veces más adelante sin tener que volver a pensarla otra vez."  
Los patrones que Christopher Alexander y sus colegas definieron, publicados en un volumen denominado *A Pattern Language*, son un intento de formalizar y plasmar de una forma práctica generaciones de conocimiento arquitectónico. Los patrones no son principios abstractos que requieran su redescubrimiento para obtener una aplicación satisfactoria, ni son específicos a una situación particular o cultural; son algo intermedio. Un patrón define una posible solución correcta para un problema de diseño dentro de un contexto dado, describiendo las cualidades invariantes de todas las soluciones. Dentro de las soluciones de Christopher Alexander se encuentran cómo se deben diseñar ciudades y dónde deben ir las perillas de las puertas.

Más tarde, en 1987, [Ward Cunningham](https://es.wikipedia.org/wiki/Ward_Cunningham) y [Kent Beck](https://es.wikipedia.org/wiki/Kent_Beck), sobrepasados por el pobre entrenamiento que recibían los nuevos programadores en orientación a objetos, se preguntaban cómo se podían capturar las buenas ideas para, luego de alguna manera, traspasarlas a los nuevos programadores recién instruidos en herencia y polimorfismo. Leyendo a Alexander se dieron cuenta del paralelo que existía entre la buena arquitectura propuesta por Alexander y la buena arquitectura OO, de modo que usaron varias ideas de Alexander para desarrollar cinco patrones de interacción hombre-ordenador (HCI) y publicaron un artículo en OOPSLA-87 titulado **Using Pattern Languages for OO Programs**.

No obstante, no fue hasta principios de la década de 1990 cuando los patrones de diseño tuvieron un gran éxito en el mundo de la informática a partir de la publicación del libro [*Design Patterns*](https://es.wikipedia.org/w/index.php?title=Design_Patterns&action=edit&redlink=1) escrito por el grupo [Gang of Four](https://es.wikipedia.org/w/index.php?title=Gang_of_Four_(dise%C3%B1o)&action=edit&redlink=1" \t "_blank" \o "Gang of Four (diseño) (aún no redactado)) (**GoF**) compuesto por [Erich Gamma](https://es.wikipedia.org/wiki/Erich_Gamma), [Richard Helm](https://es.wikipedia.org/w/index.php?title=Richard_Helm&action=edit&redlink=1), [Ralph Johnson](https://es.wikipedia.org/w/index.php?title=Ralph_Johnson_(inform%C3%A1tica)&action=edit&redlink=1) y [John Vlissides](https://es.wikipedia.org/w/index.php?title=John_Vlissides&action=edit&redlink=1), en el que se recogían 23 patrones de diseño comunes.

# Categorías de patrones de diseño

➡️**Creacionales**  
Proveen diferentes mecanismos para crear objetos.

* Abstract Factory
* Builder
* Factory Method
* Prototype
* Singleton  
     
  ➡️**Estructurales**  
  Describen formas de componer objetos para formar nuevas estructuras flexibles y eficientes.
* Adapter
* Bridge
* Composite
* Decorator
* Facade
* Flyweight
* Proxy  
     
  ➡️**Comportamiento**  
  Gestionan algoritmos y responsabilidades entre objetos.
* Chain of Responsibility
* Command
* Interpreter
* Iterator
* Mediator
* Memento
* Observer
* State
* Strategy
* Template Method
* Visitor

# Patrón Singleton y Casos de Uso

Es un patrón que te asegura que una clase solo tiene una instancia. Esta única instancia puede ser consumida por cualquier otro objeto.

# 

# ¿Cómo funciona el Patrón Observer?

En esta clase Richard Kaufman, tu profesor en el Curso profesional de JavaScript, nos explica el funcionamiento del patrón observer y como implementarlo.

El patrón observer se compone de un sujeto que ofrece mecanismos de suscripción y desuscripción a múltiples observadores que quieren ser notificados de los cambios en dicho sujeto. Cada observador expone un método de update que es usado por el sujeto para notificar cualquier cambio a todos los suscritos.

Es uno de los patrones más utilizados, algunos ejemplos típicos son:

* Newsletter
* Sockets
* Listeners en páginas web

## Decorator (patrón de diseño)

El [patrón](https://es.wikipedia.org/wiki/Patr%C3%B3n_de_dise%C3%B1o) **Decorator** responde a la necesidad de añadir dinámicamente funcionalidad a un Objeto. Esto nos permite no tener que crear sucesivas clases que hereden de la primera incorporando la nueva funcionalidad, sino otras que la implementan y se asocian a la primera.

## Motivación

Un ejemplo para poder ver la aplicabilidad del patrón decorador podría ser el siguiente:

* Disponemos de una herramienta para crear interfaces gráﬁcas, que permite añadir funcionalidades como bordes o barras de desplazamiento a cualquier componente de la interfaz.
* Una posible solución sería utilizar la herencia para extender las responsabilidades de la clase. Si optamos por esta solución, estaríamos haciendo un diseño inflexible (estático), ya que el cliente no puede controlar cuándo y cómo decorar el componente con esa propiedad.
* La solución está en encapsular dentro de otro objeto, llamado Decorador, las nuevas responsabilidades. El decorador redirige las peticiones al componente y, además, puede realizar acciones adicionales antes y después de la redirección. De este modo, se pueden añadir decoradores con cualidades añadidas recursivamente.