tecnicas de programación

Santiago Chiale-santiagochiale@abc.gob.ar

Contenido

[Introducción 4](#_Toc194222972)

[Enfoque formativo y profesional 4](#_Toc194222973)

[Aprender a programar como técnica fundamental 4](#_Toc194222974)

[Unidad 1: Estructura de Datos (Primera Parte) 5](#_Toc194222975)

[Concepto de algoritmo, resolución algorítmica de problemas y estrategias de diseño 5](#_Toc194222976)

[¿Por qué son importantes? 5](#_Toc194222977)

[Algoritmos fundamentales y algoritmos numéricos simples 5](#_Toc194222978)

[Estructuras fundamentales: variables, constantes, tipos, expresiones y entrada/salida 6](#_Toc194222979)

[Variables y constantes 6](#_Toc194222980)

[Expresiones y asignaciones 8](#_Toc194222981)

[Entrada y salida 8](#_Toc194222982)

[Estructuras de control condicionales e iterativas 9](#_Toc194222983)

[Funciones y pasaje de parámetros 12](#_Toc194222984)

[¿Qué es una función? 12](#_Toc194222985)

[Tipos de funciones 15](#_Toc194222986)

[Descomposición estructurada 16](#_Toc194222987)

[Ejercicios prácticos para resolver 18](#_Toc194222988)

[Unidad 2: Estructura de Datos (Segunda Parte) 19](#_Toc194222989)

[Máquinas virtuales (MV), jerarquía y lenguajes intermedios 19](#_Toc194222990)

[¿Para qué sirve una máquina virtual? 19](#_Toc194222991)

[Jerarquía de máquinas virtuales 19](#_Toc194222992)

[Asuntos de seguridad al ejecutar código en máquinas virtuales 20](#_Toc194222993)

[Representación de datos numéricos (rango, precisión y errores de redondeo) 21](#_Toc194222994)

[Arreglos, listas y procesamiento de datos de caracteres 22](#_Toc194222995)

[Arreglos (arrays) 22](#_Toc194222996)

[Elección de una estructura de datos adecuada 24](#_Toc194222997)

[Ejercicios prácticos para resolver 24](#_Toc194222998)

[Unidad 3: Metodología de Resolución de Problemas 25](#_Toc194222999)

[Programación modular: concepto y aplicación 25](#_Toc194223000)

[Clases, objetos y programación orientada a objetos 26](#_Toc194223001)

[Reglas para escribir algoritmos eficientes 32](#_Toc194223002)

[Complejidad algorítmica 33](#_Toc194223003)

[Recomendaciones clave para escribir algoritmos eficientes 33](#_Toc194223004)

[Eliminar cálculos repetidos o redundantes 33](#_Toc194223005)

[Usar bucles simples siempre que sea posible 34](#_Toc194223006)

[Emplear estructuras de datos adecuadas 34](#_Toc194223007)

[Modularizar el código 34](#_Toc194223008)

[Anticipar el crecimiento de los datos 35](#_Toc194223009)

[*Herramientas para analizar eficiencia* 35](#_Toc194223010)

[Conclusión 35](#_Toc194223011)

[Elaboración de algoritmos tipo 36](#_Toc194223012)

[Algoritmos de camino mínimo 38](#_Toc194223013)

[Elementos de complejidad de algoritmos 38](#_Toc194223014)

[Pruebas de escritorio 39](#_Toc194223015)

[Ejercicios prácticos 40](#_Toc194223016)

[Unidad 4: Desarrollo de Programas 42](#_Toc194223017)

[Ambientes de programación 42](#_Toc194223018)

[Ambientes de desarrollo: local, staging y producción 42](#_Toc194223019)

[Comparación de ambientes 43](#_Toc194223020)

[¿Cómo funciona una aplicación web en Internet? 43](#_Toc194223021)

[APIs (Interfaz de Programación de Aplicaciones) 43](#_Toc194223022)

[Tipos de APIs 43](#_Toc194223023)

[¿Qué es una API? 44](#_Toc194223024)

[Cómo se utiliza una API en JavaScript 44](#_Toc194223025)

[*APIs públicas gratuitas para practicar* 46](#_Toc194223026)

[APIs Web integradas en JavaScript (nativas del navegador) 46](#_Toc194223027)

[*Buenas prácticas al usar APIs* 47](#_Toc194223028)

[Ejercicios prácticos 47](#_Toc194223029)

# Introducción

El presente material fue diseñado como guía de estudio y consulta para acompañar el desarrollo de competencias fundamentales en el marco de la carrera Tecnicatura Superior en Ciencia de Datos e Inteligencia Artificial. Su contenido responde a una necesidad concreta: que el/la estudiante no solo aprenda a programar, sino que lo haga desde una perspectiva crítica, aplicando buenas prácticas y desarrollando soluciones eficientes, sostenibles y con impacto real.

El plan de estudio se articula en torno a cuatro unidades temáticas que acompañan el crecimiento progresivo de las habilidades de análisis, diseño, codificación y evaluación de soluciones computacionales. Estas unidades no solo abordan aspectos técnicos, sino también metodológicos y estratégicos del desarrollo de software, con énfasis en la resolución de problemas y el uso profesional de herramientas modernas.

## Enfoque formativo y profesional

La formación en Ciencia de Datos e Inteligencia Artificial exige un dominio sólido de la programación estructurada y modular, el diseño algorítmico, el uso adecuado de estructuras de datos, y la capacidad de construir soluciones eficientes tanto en términos computacionales como organizacionales. Todo esto debe articularse con una visión integral que contemple:

La comprensión de los datos: saber analizarlos, limpiarlos y transformarlos para que puedan ser utilizados en modelos y sistemas inteligentes.

El diseño e implementación de algoritmos: desde los más simples hasta aquellos con implicancias en la toma de decisiones automatizada.

La comunicación clara de resultados: mediante visualizaciones efectivas, documentación comprensible y estructuras de código limpias.

A lo largo de este material se trabajan ejemplos, ejercicios y conceptos que conectan directamente con las funciones y capacidades que se esperan de un/a profesional técnico/a en esta disciplina. Entre ellas:

Diseñar e implementar soluciones algorítmicas para distintos problemas.

Utilizar estructuras de datos adecuadas para optimizar el procesamiento de la información.

Aplicar principios de modularidad y reutilización en el desarrollo de código.

Interactuar con APIs y servicios externos para integrar datos y funcionalidades a los proyectos.

Realizar pruebas, validaciones y análisis de eficiencia algorítmica.

Aprender a programar como técnica fundamental

El eje transversal de este documento es el pensamiento algorítmico aplicado a problemas reales. Se parte del análisis del problema, se continúa con el diseño lógico de la solución, se implementa con herramientas de programación modernas (como JavaScript), y finalmente se evalúan los resultados y el desempeño del programa.

En este sentido, aprender a programar no se reduce a escribir instrucciones para la computadora: implica aprender a pensar en términos de procesos, estructuras, abstracciones y eficiencia, algo que es clave en todas las etapas del ciclo de Ciencia de Datos y en la construcción de modelos de Inteligencia Artificial.

# Unidad 1: Estructura de Datos (Primera Parte)

## Concepto de algoritmo, resolución algorítmica de problemas y estrategias de diseño

Antes de aprender a programar, es fundamental comprender qué es un algoritmo.

Un algoritmo es como una receta: una serie de pasos claros, finitos y ordenados que permiten resolver un problema o realizar una tarea. Un algoritmo no necesariamente está escrito en un lenguaje de programación; puede expresarse en lenguaje natural, pseudocódigo o diagramas de flujo.

### ¿Por qué son importantes?

Los algoritmos son la base de la informática.

Nos permiten:

* Pensar de manera ordenada y lógica
* Resolver problemas complejos paso a paso
* Comunicar nuestras ideas a una computadora para que pueda ejecutarlas

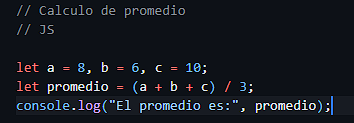
Características de un buen algoritmo:

* Claridad: cada paso debe estar definido sin ambigüedades
* Finitud: debe tener un fin (no puede ser infinito)
* Eficiencia: debe utilizar los recursos de forma óptima
* Generalidad: debe resolver una clase de problemas, no solo uno específico

### Algoritmos fundamentales y algoritmos numéricos simples

Estos son algoritmos que permiten resolver problemas básicos y frecuentes, como operaciones matemáticas, búsqueda de valores, ordenamientos, etc.

Ejemplo:



Ejemplo:

Texto

El contenido generado por IA puede ser incorrecto.

## Estructuras fundamentales: variables, constantes, tipos, expresiones y entrada/salida

### Variables y constantes

Una variable es un espacio de memoria donde almacenamos información que puede cambiar durante la ejecución del programa. En JavaScript, se puede declarar una variable con let o var. Sin embargo, en los programas modernos se prefiere **let** por sobre **var** debido a su comportamiento más predecible y seguro.

Para declarar constantes, usamos **const**. Una constante es un valor que no puede ser reasignado una vez definido.

#### Alcance de las variables (scope)

El alcance determina desde qué partes del programa se puede acceder a una variable.

* **let** y **const** tienen alcance de bloque (solo existen dentro del bloque {} donde fueron definidas).
* **var** tiene alcance de función (puede ser accedida en toda la función donde fue declarada).

Recomendación: evita usar var para evitar errores de alcance inesperados

¿Cuándo usar **let** o **const**?

Usá **let** si el valor puede cambiar (por ejemplo: contador, saldo, temperatura, etc.).

Usá **cons**t si el valor debe mantenerse fijo (por ejemplo: tasa fija, textos predeterminados, límites, configuraciones).

#### Tipos de datos

En JavaScript, los valores tienen distintos tipos de datos. Los más comunes son:

* *number*: representa números, tanto enteros como decimales.
* *string*: cadenas de texto, se escriben entre comillas simples (') o dobles (").
* *boolean*: representa valores lógicos: true (verdadero) o false (falso).

Otros tipos importantes son:

* *undefined*: una variable que ha sido declarada pero no tiene valor.
* *null*: representa la ausencia intencional de un valor.
* *object*: estructuras más complejas que agrupan datos y funcionalidades.
* *array*: listas ordenadas de elementos.
* *symbol* y *bigint*: tipos avanzados, menos comunes en los primeros niveles.

#### Ejemplos

Texto

El contenido generado por IA puede ser incorrecto.

### Expresiones y asignaciones

Una expresión es cualquier fragmento de código que produce un valor. Puede incluir operaciones aritméticas, comparaciones, llamadas a funciones, etc.

Una asignación se realiza con el operador =, y sirve para guardar un valor en una variable.

Una captura de pantalla de un celular

El contenido generado por IA puede ser incorrecto.

### Entrada y salida

JavaScript ofrece formas básicas de entrada y salida (I/O) cuando se ejecuta en un navegador:

Texto

El contenido generado por IA puede ser incorrecto.

prompt() y alert() son útiles para pruebas simples. En aplicaciones reales se usan formularios HTML o bibliotecas más avanzadas.

### Estructuras de control condicionales e iterativas

#### ¿Qué es una estructura de control?

Es una herramienta del lenguaje que permite modificar el flujo natural del programa, ya sea tomando decisiones o repitiendo instrucciones. Gracias a estas estructuras, los programas pueden adaptarse a diferentes situaciones y comportarse de forma dinámica.

##### Condicionales (if / else)

Permiten ejecutar diferentes bloques de código dependiendo de si se cumple una condición lógica.

Pantalla de computadora con letras

El contenido generado por IA puede ser incorrecto.

##### Switch case

La estructura switch es útil cuando se quiere comparar una misma variable con varios valores posibles. Funciona como una serie de if/else if, pero más ordenado.

Texto

El contenido generado por IA puede ser incorrecto.

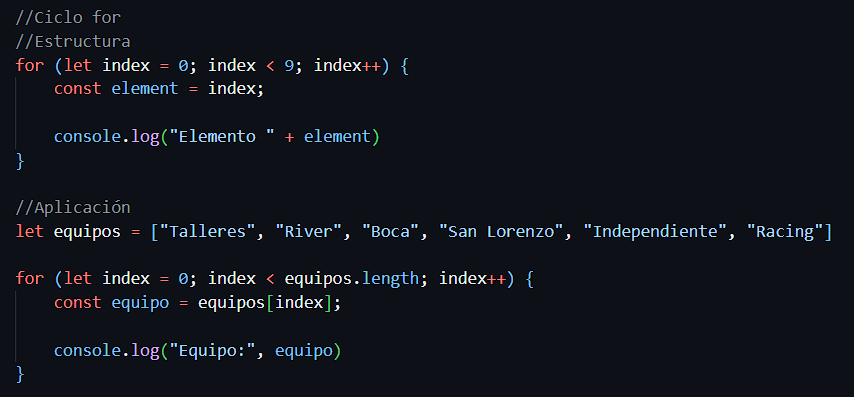
No olvides el break para evitar que el programa ejecute todos los casos siguientes.

#### Estructuras iterativas (bucles)

Permiten repetir instrucciones mientras se cumpla una condición. Los bucles más comunes son:

##### for

Se usa cuando sabemos cuántas veces queremos repetir algo:



##### while

Se ejecuta mientras una condición sea verdadera. La condición se evalúa antes de cada iteración:

Texto

El contenido generado por IA puede ser incorrecto.

##### do...while

Similar al while, pero la condición se evalúa después de ejecutar el bloque, por lo que siempre se ejecuta al menos una vez:

Una captura de pantalla de un celular con texto e imagen

El contenido generado por IA puede ser incorrecto.

##### Diferencias entre while y do...while

|  |  |  |
| --- | --- | --- |
| Característica | while | do...while |
| Evalúa la condición | Antes de la iteración | Después de la iteración |
| Puede no ejecutarse nunca | Sí | No, siempre ejecuta al menos una vez |

##### map vs for

Ambos permiten recorrer elementos de una lista (array), pero con propósitos diferentes:

###### for

Es más flexible. Permite realizar cualquier operación durante el recorrido:

Captura de pantalla de un celular

El contenido generado por IA puede ser incorrecto.

###### map

Se usa cuando queremos transformar un array en otro. Devuelve un nuevo array con los resultados:

Una captura de pantalla de un celular con texto e imagen

El contenido generado por IA puede ser incorrecto.

Usá map si necesitás crear un nuevo array a partir de uno existente.  
Usá for si necesitás más control o ejecutar acciones más complejas.

## Funciones y pasaje de parámetros

### ¿Qué es una función?

Una función es un bloque de código que agrupa una o más instrucciones, con el objetivo de que puedan reutilizarse cuantas veces sea necesario. Las funciones permiten estructurar un programa en partes más pequeñas, con nombres significativos que explican lo que hace cada sección.

Podemos pensar una función como una "máquina" que recibe una entrada (parámetros), hace un procesamiento interno, y devuelve una salida (valor de retorno).

#### ¿Por qué son importantes las funciones en programación?

Las funciones son fundamentales por varias razones:

* Reutilización de código. Una vez que definís una función, podés usarla todas las veces que quieras sin volver a escribir el mismo código.
* Organización y claridad. Ayudan a dividir un problema grande en partes más pequeñas y manejables, lo que hace que el código sea más fácil de leer y mantener.
* Evita la duplicación. Centralizan la lógica que se repite, reduciendo errores y simplificando futuras modificaciones.
* Facilitan el testing. Es más fácil probar y depurar una función individual que todo un programa a la vez.
* Abstracción. Permiten centrarse en el "qué hace" una parte del programa, sin necesidad de preocuparse por el "cómo lo hace".

Texto

El contenido generado por IA puede ser incorrecto.

Este código repite la misma lógica para distintos valores. Esto es poco eficiente y propenso a errores si necesitamos cambiar la fórmula en varios lugares.

Texto

El contenido generado por IA puede ser incorrecto.

Ventajas:

* El código es más corto, claro y reutilizable.
* Si la fórmula cambia, solo hay que modificarla en un lugar.
* Podemos reutilizar user\_register en cualquier parte del programa.

### Tipos de funciones

#### Funciones con parámetros

Pantalla de un celular con texto e imagen

El contenido generado por IA puede ser incorrecto.

#### Funciones que devuelven un valor

Texto

El contenido generado por IA puede ser incorrecto.

#### Funciones sin parámetros

Texto

El contenido generado por IA puede ser incorrecto.

#### Funciones flecha (arrow functions)

##### Forma moderna y concisa:



Las funciones flecha son muy útiles cuando queremos escribir funciones cortas, especialmente como argumentos en métodos como map, filter, reduce.

#### Funciones que llaman a otras funciones

Texto

El contenido generado por IA puede ser incorrecto.

Esto demuestra cómo se pueden combinar funciones para construir programas más complejos y modulares.

#### Buenas prácticas con funciones

* Usá nombres claros y descriptivos (calcularPromedio, esMayorDeEdad).
* Nombres con camel Case pueden ser una buena practica (camelCase)
* Una función debe realizar una sola tarea específica.
* Si una función crece demasiado, considera dividirla en funciones más pequeñas.
* Evita que las funciones modifiquen variables externas (efectos secundarios), salvo que sea necesario.

## Descomposición estructurada

La descomposición estructurada es una técnica fundamental en programación que consiste en dividir un problema en partes más pequeñas y específicas, resolviendo cada una por separado a través de funciones.

La idea es abordar problemas complejos paso a paso, resolviendo tareas más simples y conectándolas para formar una solución completa.

Este enfoque permite:

* Organizar el código en bloques lógicos y comprensibles.
* Reutilizar funciones cuando sean necesarias en otras partes del programa.
* Facilitar el mantenimiento y la detección de errores.
* Colaborar mejor en proyectos donde diferentes personas trabajan sobre distintas partes.

Ejemplo simple: sistema de login

Supongamos que queremos registrar un usuario, verificar edad, y mostrar diferentes mensajes según el caso. Podemos usar descomposición para hacerlo más claro:

Texto

El contenido generado por IA puede ser incorrecto.

Este código está dividido en funciones pequeñas y específicas:

* *iniciarRegistro* coordina el proceso.
* *pedirNombre* se encarga de pedir mediante interfaz de usuario el nombre
* *pedirEdad se* encarga de pedir mediante interfaz de usuario la edad
* *esMayorDeEdad se encarga de evaluar si la edad es mayor a 18 años*
* *registrar* y *rechazarRegistro* manejan la respuesta visual.

Cada función tiene una misión clara, lo que facilita futuros cambios (por ejemplo, cambiar el criterio de edad mínima o la forma de pedir los datos).

#### Relación con programación estructurada y modular

La descomposición estructurada es parte de la programación estructurada, un paradigma que promueve el uso de estructuras de control y funciones para evitar el código desordenado y repetitivo.

Además, sienta las bases de la programación modular, donde los programas se organizan en módulos reutilizables y escalables.

##### Recomendaciones para aplicar descomposición

* Identifica las tareas repetidas o independientes: cada una puede ser una función.
* Usá nombres descriptivos para que el código sea auto explicativo.
* Mantener las funciones cortas y específicas: si una hace muchas cosas, dividida.
* Proba las funciones de forma individual para asegurarte de que hacen lo correcto.

## Ejercicios prácticos para resolver

A continuación, se proponen 10 ejercicios que podés resolver en plataformas online como [CodePen](https://codepen.io/) o en tu propio entorno (por ejemplo, VS Code):

1. Área de un triángulo. Usá la fórmula (base \* altura) / 2.
2. Convertidor de temperaturas. Convertí de grados Celsius a Fahrenheit.
3. Verificador de mayoría de edad. Mostrá si el usuario es mayor de 18.
4. Suma de pares del 1 al 100. Sumá todos los números pares usando un bucle.
5. Tabla de multiplicar. Mostrá la tabla del número ingresado.
6. Contador de vocales. Contá cuántas vocales hay en una palabra.
7. Login simple. Usuario: admin – Clave: 1234
8. Números primos del 1 al 100
9. Cálculo de promedio con validación
10. Simulador de cajero automático

# Unidad 2: Estructura de Datos (Segunda Parte)

## Máquinas virtuales (MV), jerarquía y lenguajes intermedios

Una máquina virtual (MV) es un software que simula el funcionamiento de una computadora física. Ejecuta programas en un entorno controlado y aislado del hardware real. Gracias a esto, un mismo programa puede funcionar en distintas plataformas sin necesidad de cambiar el código fuente.

### ¿Para qué sirve una máquina virtual?

* Permite independencia del sistema operativo: el mismo código puede correr en Windows, Linux o macOS.
* Ofrece un entorno seguro para pruebas o ejecución de código.
* Mejora la portabilidad del software.
* Facilita la implementación de lenguajes de alto nivel a través de lenguajes intermedios.

### Jerarquía de máquinas virtuales

* Hardware real (CPU, memoria, disco)
* Sistema operativo (Windows, Linux, etc.)
* Máquina virtual (V8, JVM, Python VM)
* Lenguaje intermedio (bytecode, AST)
* Programa del usuario
* Lenguajes intermedios

Muchos lenguajes modernos no se ejecutan directamente en el hardware. Primero se traducen a un lenguaje intermedio, que la máquina virtual interpreta o compila en tiempo real:

* Java → bytecode → JVM
* TypeScript → JavaScript → navegador
* Python → bytecode .pyc → Python VM

Esto permite que una misma aplicación sea ejecutada en diferentes entornos sin reescribir el código.

### Asuntos de seguridad al ejecutar código en máquinas virtuales

Aunque las máquinas virtuales brindan aislamiento y seguridad, su mal uso o el descuido en la programación pueden provocar vulnerabilidades.

Uno de los errores más comunes es ejecutar código dinámico sin validarlo adecuadamente. El ejemplo más riesgoso es el uso de la función eval().

#### Peligro: uso de eval()

Texto

El contenido generado por IA puede ser incorrecto.

Una captura de pantalla de un celular con texto e imagen

El contenido generado por IA puede ser incorrecto.

El código será ejecutado por el navegador, lo cual representa una grave vulnerabilidad.

Texto

El contenido generado por IA puede ser incorrecto.

Este enfoque evita la ejecución arbitraria de código y es mucho más seguro.

Buenas prácticas

* Nunca usar eval() con datos del usuario
* Validar y sanear cualquier entrada antes de usarla
* Preferir funciones nativas seguras como Number(), parseInt()
* Usar entornos de prueba y revisión para analizar entradas complejas

## Representación de datos numéricos (rango, precisión y errores de redondeo)

Las computadoras representan números usando una cantidad limitada de bits. Esto produce limitaciones en rango, precisión y exactitud.

#### Rango y precisión

En JavaScript, los números flotantes siguen el estándar IEEE 754.

Esto significa que hay una precisión máxima de unos 15 dígitos.

Los decimales no siempre se representan con exactitud.

Texto

El contenido generado por IA puede ser incorrecto.

También existen librerías especializadas para manejo de dinero y precisión decimal (como Decimal.js).

## Arreglos, listas y procesamiento de datos de caracteres

### Arreglos (arrays)

Un arreglo es una estructura que permite almacenar múltiples valores bajo una misma variable. Los elementos del arreglo se acceden por índice:

Texto

El contenido generado por IA puede ser incorrecto.

Los arreglos son muy útiles para recorrer listas, almacenar múltiples entradas del usuario o manejar datos dinámicamente.

Métodos frecuentes de recorrido de arreglos en JavaScript

#### for tradicional

Texto

El contenido generado por IA puede ser incorrecto.

Características

* Acceso total al índice
* Mayor control del bucle
* Más rápido en situaciones críticas

#### for...of

Una captura de pantalla de un celular con texto e imagen

El contenido generado por IA puede ser incorrecto.

Características

* Sintaxis simple
* Menor control sobre el índice
* Muy legible y útil para lectura

#### forEach()

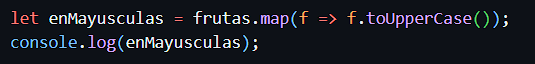
Una captura de pantalla de un celular con texto e imagen

El contenido generado por IA puede ser incorrecto.

Características

* Función de orden superior
* Ideal para aplicar operaciones simples a cada elemento

#### map()



Características

* Devuelve un nuevo arreglo transformado
* Muy útil para transformar datos

#### filter()

Texto

El contenido generado por IA puede ser incorrecto.

Características

* Devuelve un nuevo arreglo con los elementos que cumplen una condición

#### reduce()



Características

* Acumula valores en una sola salida
* Comparación de consumo y rendimiento

|  |  |  |  |
| --- | --- | --- | --- |
| Método | Velocidad | Control del índice | Ideal para |
| for | Alta (rápido) | ✅ Sí | Búsquedas, performance crítica |
| for...of | Media | ❌ No | Lectura simple de elementos |
| forEach() | Media | ❌ No | Acciones simples sobre arrays |
| map() | Media | ❌ No | Transformaciones funcionales |
| filter() | Media | ❌ No | Selección de subconjuntos |
| reduce() | Baja | ❌ No | Acumulación compleja de valores |

## Elección de una estructura de datos adecuada

Elegir bien una estructura de datos mejora el rendimiento, facilita la programación y evita errores.

Comparación de estructuras

|  |  |
| --- | --- |
| Estructura | Ideal para... |
| Array | Listas ordenadas, acceso por índice |
| Object | Agrupar datos con nombre |
| Set | Listas sin duplicados |
| Map | Claves personalizadas, búsqueda rápida |
|  |  |

###### ¿Cómo decidir?

¿Necesito acceder por posición? → Array

¿Quiero agrupar atributos? → Object

¿No deben repetirse valores? → Set

¿Quiero usar claves como objetos? → Map

## Ejercicios prácticos para resolver

1. Mostrar los primeros 10 números con dos decimales exactos
2. Detectar si un número decimal tiene error de redondeo (ej.: 0.1 + 0.2)
3. Crear un array de 5 elementos y mostrarlos en orden inverso
4. Contar cuántas veces aparece una letra en una palabra
5. Pedir una lista de nombres y ordenarlos alfabéticamente
6. Simular una agenda de contactos con objetos (nombre, teléfono, email)
7. Usar un Set para eliminar duplicados de una lista de palabras
8. Mostrar el índice de la primera vocal en una palabra
9. Mostrar la suma de los elementos de un array numérico
10. Comparar los tiempos de búsqueda en array vs. objeto (simple)

# Unidad 3: Metodología de Resolución de Problemas

## Programación modular: concepto y aplicación

A medida que los programas crecen en complejidad, se vuelve imprescindible organizar el código de forma clara, flexible y mantenible. La programación modular propone una solución a este problema: dividir el software en pequeñas unidades funcionales llamadas módulos, cada una con una responsabilidad específica.

Este enfoque permite trabajar sobre partes individuales del programa, favorece la reutilización del código y facilita la colaboración entre múltiples programadores.

En JavaScript, los módulos suelen implementarse como funciones o clases, que encapsulan lógica y pueden ser llamadas desde distintas partes del programa.

Ejemplo básico de programación modular:

Pantalla de un celular con texto e imagen

El contenido generado por IA puede ser incorrecto.

### Clases, objetos y programación orientada a objetos

La **Programación Orientada a Objetos (POO)** es un paradigma de programación que organiza el software en torno a *objetos*, que representan entidades del mundo real o conceptual. Cada objeto combina *estado* (propiedades) y *comportamiento* (métodos), y es creado a partir de una *clase*, que actúa como un molde o plantilla.

Este enfoque permite representar sistemas complejos de forma modular, reutilizable e intuitiva, y es ampliamente utilizado en el desarrollo de aplicaciones modernas, tanto del lado del cliente (front-end) como del servidor (back-end).

#### ¿Qué es una clase?

Una clase es una **plantilla para crear objetos**. Se define usando la palabra clave class y contiene un **constructor** (una función especial que se ejecuta al crear una instancia) y uno o más **métodos** que definen el comportamiento del objeto.

Texto

El contenido generado por IA puede ser incorrecto.

#### El método constructor()

Es un método especial para inicializar el objeto cuando se crea una nueva instancia con new.

Se llama automáticamente y suele usarse para asignar valores iniciales a las propiedades.

#### ¿Qué es this?

En el contexto de las clases, la palabra clave ***this*** hace referencia a la **instancia actual del objeto**. Se utiliza dentro del constructor o de los métodos para acceder a las propiedades y métodos del objeto en cuestión.

En el ejemplo anterior **this.nombre** se refiere a la propiedad nombre de la instancia actual.

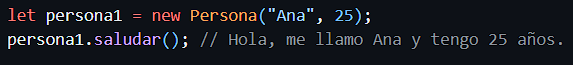
Cuando se ejecuta persona.saludar(), el valor de this dentro del método saludar() apunta a persona.

**Importante:**

El valor de this puede variar dependiendo del contexto de ejecución. Pero dentro de una clase, usado correctamente, this siempre se refiere al objeto que fue instanciado con new.

#### ¿Qué es una instancia?

Una **instancia** es un objeto específico creado a partir de una clase. Se utiliza la palabra clave new para crear una instancia.



Cada vez que se usa new Persona(...), se crea un nuevo objeto con sus propias propiedades.

#### Propiedades y métodos

**Propiedades**: Son los datos que describen al objeto. Se definen en el constructor con this.

Ejemplo: this.nombre, this.edad

**Métodos**: Son funciones que pertenecen a la clase y definen su comportamiento.

Ejemplo: saludar()

Los métodos se escriben dentro del cuerpo de la clase, pero fuera del constructor.

#### Herencia de clases

La **herencia** permite que una clase (llamada clase hija o subclase) herede las propiedades y métodos de otra clase (llamada clase padre o superclase). En JavaScript se usa la palabra clave extends.

Captura de pantalla de computadora

El contenido generado por IA puede ser incorrecto.

#### Palabra clave super

super(...) se utiliza dentro del constructor de la subclase para llamar al constructor de la clase padre.

También se puede usar super.metodo() para llamar a un método heredado desde la subclase.

#### Encapsulamiento: propiedades públicas y privadas

Desde ECMAScript 2022, JavaScript permite definir **propiedades privadas** usando el símbolo # delante del nombre de la propiedad o método. Esto permite proteger datos internos del objeto.

Texto

El contenido generado por IA puede ser incorrecto.

#### Métodos estáticos

Los **métodos estáticos** pertenecen a la clase y no a sus instancias. Se definen con la palabra clave static y se accede a ellos directamente desde la clase.

Texto

El contenido generado por IA puede ser incorrecto.

#### Getters y Setters

Permiten definir funciones especiales para obtener o modificar propiedades, sin acceder directamente a ellas. Son útiles para validar datos o encapsular lógica.

Texto

El contenido generado por IA puede ser incorrecto.

#### Resumen de palabras clave importantes

|  |  |
| --- | --- |
| Palabra clave | Descripción |
| super | Llama al constructor o métodos de la clase padre |
| static | Define métodos que se llaman desde la clase y no desde instancias |
| get | Define un getter para acceder a una propiedad |
| set | Define un setter para modificar una propiedad |
| # | Define propiedades privadas |
| class | Define una clase |
| constructor | Método especial para crear e inicializar objetos |
| extends | Permite heredar de otra clase |
| this | Referencia al objeto actual (la instancia en curso) |

**Mini proyecto: Registro de Estudiantes**

**Objetivo:**

Construir un sistema orientado a objetos para registrar estudiantes y carreras, con lógica básica de validación y herencia.

**Requisitos:**

**1. Clase Persona**

Propiedades: nombre, dni

Método: presentarse() → muestra: "Hola, soy [nombre] y mi DNI es [dni]."

**2. Clase Estudiante (hereda de Persona)**

Propiedades adicionales: carrera, legajo, #notas (privadas)

Métodos:

agregarNota(nota) → solo permite valores entre 1 y 10

obtenerPromedio()

mostrarDatos() → muestra todos los datos relevantes

**3. Clase Carrera**

Propiedades: nombre, duracion

Método estático: compararDuracion(c1, c2) → indica cuál dura más

**4. Crear varias instancias de Estudiante y Carrera, simular la carga de notas y mostrar los resultados.**

**Desafío extra (opcional):**

Agregar una clase Curso con una lista de estudiantes y métodos para:

* Inscribir estudiantes
* Calcular promedio general del curso
* Mostrar todos los alumnos inscritos

### Reglas para escribir algoritmos eficientes

No basta con que un programa funcione: debe hacerlo **bien**. Esto significa que debe **resolver el problema de forma correcta, rápida y utilizando la menor cantidad de recursos posibles**. En programación profesional —y especialmente en ciencia de datos e inteligencia artificial— la eficiencia es un criterio clave para garantizar el rendimiento, la escalabilidad y la calidad del software.

Un algoritmo eficiente:

* Realiza **el menor número posible de operaciones**.
* Utiliza **estructuras de datos adecuadas**.
* Escala correctamente con el tamaño de entrada.
* Es **claro, modular y mantenible**.

**Ejemplo de código ineficiente vs eficiente**

Texto

El contenido generado por IA puede ser incorrecto.

El código sumaLenta hace un bucle adicional que no aporta nada: eso introduce **cálculos redundantes**. En cambio, sumaRapida realiza una única pasada sobre el array.

## Complejidad algorítmica

La eficiencia también se analiza usando la **notación Big O (O-grande)**, que mide cómo crece el tiempo de ejecución en relación al tamaño de la entrada (n).

|  |  |  |
| --- | --- | --- |
| Tipo de algoritmo | Notación | Ejemplo |
| Constante | O(1) | Acceso directo a un array |
| Lineal | O(n) | Recorrer un array una vez |
| Cuadrático | O(n²) | Doble bucle anidado |
| Logarítmico | O(log n) | Búsqueda binaria |
| Lineal-logarítmico | O(n log n) | Ordenamiento eficiente |

**Regla práctica:** Evitar algoritmos con complejidad O(n²) o peor, especialmente si se espera trabajar con grandes volúmenes de datos.

## Recomendaciones clave para escribir algoritmos eficientes

### Eliminar cálculos repetidos o redundantes

Evitá repetir operaciones innecesarias dentro de bucles.

Pantalla de computadora con letras

El contenido generado por IA puede ser incorrecto.

### 

### Usar bucles simples siempre que sea posible

Optar por **una sola pasada** sobre los datos si no se justifica más.

Una captura de pantalla de un celular

El contenido generado por IA puede ser incorrecto.

### Emplear estructuras de datos adecuadas

Elegí la estructura correcta para cada tarea.

Por ejemplo:

* Usar Set para evitar duplicados y búsquedas rápidas.
* Usar Map para acceso por clave más eficiente que en objetos grandes.
* Usar arrays ordenados si necesitás búsquedas binarias.

Interfaz de usuario gráfica, Texto

El contenido generado por IA puede ser incorrecto.

### Modularizar el código

Dividir el código en funciones pequeñas y específicas mejora la legibilidad, reutilización y facilita pruebas.

Pantalla de computadora con letras

El contenido generado por IA puede ser incorrecto.

### Anticipar el crecimiento de los datos

El código debe escalar. Lo que funciona con 10 elementos puede volverse lento o inútil con 10.000 o más.

Comparación de tiempos (hipotética)

- Algoritmo A: O(n) → 10 ms con 10.000 elementos

- Algoritmo B: O(n²) → 1.000.000 ms con 10.000 elementos ❌

**Consejo:** Siempre que sea posible, **medí el tiempo de ejecución** con console.time() y console.timeEnd() para analizar el rendimiento.

### Herramientas para analizar eficiencia

* console.time() / console.timeEnd()
* Profiler de herramientas como Chrome DevTools
* Medición de uso de memoria (en entornos más avanzados)
* Pruebas con grandes volúmenes de datos simulados

### Conclusión

Escribir código eficiente no es solo una cuestión de velocidad: también es una forma de pensar, diseñar y anticipar el comportamiento de tu software bajo distintas condiciones. Un buen algoritmo:

* Es correcto
* Es claro
* Y es eficiente

## Elaboración de algoritmos tipo

Los algoritmos tipo son patrones de resolución que aparecen una y otra vez en la programación. Familiarizarse con ellos permite abordar problemas nuevos con más facilidad.

Ejemplos comunes:

* Contar elementos que cumplen una condición
* Calcular sumas, promedios, máximos y mínimos
* Aplicar transformaciones a listas

Ejemplo: promedio de edades

Texto

El contenido generado por IA puede ser incorrecto.

Recursividad: factorial

Texto

El contenido generado por IA puede ser incorrecto.

#### Búsquedas y ordenamientos

Encontrar un elemento o poner datos en orden son tareas fundamentales. La forma en que lo hagamos puede impactar mucho en la eficiencia del programa.

Búsqueda lineal:

Texto

El contenido generado por IA puede ser incorrecto.

Búsqueda binaria:

Pantalla negra con letras blancas

El contenido generado por IA puede ser incorrecto.

Ordenamiento burbuja:

Texto

El contenido generado por IA puede ser incorrecto.

### Algoritmos de camino mínimo

Un problema clásico en programación y ciencias de la computación es encontrar el camino más corto entre dos puntos. Este tipo de algoritmos se usa en GPS, planificación de rutas, redes sociales y videojuegos.

Introducción al algoritmo de Dijkstra:

* Se parte del nodo origen y se recorren caminos hacia los nodos vecinos.
* Se actualizan las distancias acumuladas.
* Se elige siempre el camino más corto hasta llegar al destino.

Aunque la implementación completa requiere estructuras avanzadas, es fundamental conocer su existencia y aplicación.

### Elementos de complejidad de algoritmos

El rendimiento de un algoritmo no siempre se nota con pocos datos. Pero cuando se procesan miles o millones de elementos, la diferencia entre un algoritmo eficiente y uno ineficiente puede ser abismal.

Comparación visual:

Texto

El contenido generado por IA puede ser incorrecto.

Un algoritmo O(n²) tarda 100 veces más que uno O(n) si la entrada crece de 10 a 100.

### Pruebas de escritorio

Consisten en simular la ejecución del programa en papel, anotando cada valor de las variables paso a paso. Esto es muy útil para entender y depurar algoritmos nuevos o complejos.

Tabla ejemplo:

|  |  |  |  |
| --- | --- | --- | --- |
| i | suma | valor actual | acción |
| 0 | 0 | 3 | suma += 3 |
| 1 | 3 | 5 | suma += 5 |
| 2 | 8 | 2 | suma += 2 |

#### Verificación unitaria

En el desarrollo profesional, todo código debe ser acompañado por pruebas unitarias, que verifiquen que las funciones hagan lo que se espera.

Ejemplo básico:

Pantalla negra con letras blancas

El contenido generado por IA puede ser incorrecto.

Estas pruebas ayudan a:

* Detectar errores rápidamente
* Garantizar que nuevas modificaciones no rompan el programa
* Documentar el comportamiento esperado

Frameworks como Jest, Mocha o Vitest permiten automatizar pruebas, generar reportes y medir el cubrimiento del código.

## Ejercicios prácticos

1. Área de un rectángulo (Programación modular)

Objetivo: Aplicar el concepto de modularidad dividiendo un problema en funciones específicas.  
Consigna: Escribí un programa que calcule el área de un rectángulo a partir de su base y altura, utilizando dos funciones: una para realizar el cálculo y otra para mostrar el resultado.

2. Clase Persona (Introducción a POO)

Objetivo: Comprender cómo definir una clase con propiedades y métodos.  
Consigna: Creá una clase llamada Persona con propiedades nombre y edad, y un método saludar() que muestre un mensaje de presentación por consola.

3. Herencia de clases (POO)

Objetivo: Utilizar herencia para reutilizar código entre clases.

Consigna: Extendé la clase Persona para crear una clase Estudiante que además tenga una propiedad carrera y un método estudiar(). Mostrá los métodos heredados y propios del nuevo objeto.

4. Código eficiente vs. ineficiente

Objetivo: Identificar y corregir estructuras redundantes.

Consigna: Compará dos funciones que suman los elementos de un array: una con bucles innecesarios (ineficiente) y otra optimizada (eficiente). Mostrá ambas soluciones y explicá las diferencias.

5. Factorial recursivo (Algoritmo tipo)

Objetivo: Implementar una función recursiva.

Consigna: Escribí una función recursiva que calcule el factorial de un número entero positivo. Probala con distintos valores.

6. Búsqueda binaria

Objetivo: Aplicar un algoritmo eficiente de búsqueda.

Consigna: Implementá la búsqueda binaria sobre un array de números ordenados. El programa debe indicar si encontró el valor buscado y en qué posición.

7. Ordenamiento burbuja

Objetivo: Comprender un algoritmo básico de ordenamiento.

Consigna: Escribí una función que ordene un array de números utilizando el método burbuja (bubble sort). Mostrá el arreglo antes y después del ordenamiento.

8. Comparación de complejidad

Objetivo: Visualizar la diferencia de rendimiento entre algoritmos.

Consigna: Implementá dos funciones: una de complejidad O(n) y otra de O(n²). Ejecutalas sobre un mismo array y analizá el impacto del crecimiento de datos.

9. Prueba de escritorio simulada

Objetivo: Simular manualmente la ejecución de un algoritmo.  
Consigna: Creá un programa que recorra un array de números y acumule su suma, mostrando en consola los valores de las variables en cada paso como si fuera una tabla de prueba de escritorio.

10. Verificación unitaria

Objetivo: Comprobar la correcta ejecución de funciones individuales.

Consigna: Escribí dos funciones (esPar() y sumar()) y validá su comportamiento usando console.assert() con distintos casos de prueba.

# Unidad 4: Desarrollo de Programas

## Ambientes de programación

El desarrollo de software moderno se realiza en entornos que integran herramientas de edición, prueba y depuración del código. Estos se conocen como ambientes de programación o IDEs (Entornos de Desarrollo Integrado).

Algunos ejemplos comunes son:

* Visual Studio Code: liviano, extensible, muy usado en programación web.
* Replit: plataforma online colaborativa.
* CodePen: ideal para pruebas rápidas en HTML, CSS y JavaScript.

Estos entornos facilitan la organización de proyectos, la visualización de resultados y la integración con control de versiones como Git y GitHub.

### Ambientes de desarrollo: local, staging y producción

En el ciclo de vida de una aplicación profesional, el código atraviesa distintos ambientes antes de llegar al usuario final. Esta segmentación permite detectar errores, validar nuevas funciones y asegurar calidad.

1. 🖥️ Ambiente local (desarrollo)

Es donde trabaja cada programador/a en su propia máquina. Aquí se escriben, prueban y corrigen las primeras versiones del código.

Se usan herramientas como servidores locales (XAMPP, WAMP, Python, Node.js).

Permite probar sin conexión a internet ni interferir con usuarios reales.

Es el entorno más flexible y controlado.

2. 🔬 Staging (pre-producción)

Es un entorno que simula el entorno productivo real. Sirve para probar el sistema completo antes de publicarlo.

Se aloja en servidores de prueba (internos o externos).

Usa bases de datos separadas o duplicadas (sin datos reales).

Permite validaciones con usuarios beta o equipo QA (aseguramiento de calidad).

3. 🚀 Producción (entorno en vivo)

Es el entorno al que acceden los usuarios reales. Debe ser estable, seguro y confiable.

Toda modificación debe estar probada previamente.

Se monitorean errores, rendimiento y seguridad.

Se realizan backups y actualizaciones planificadas.

### Comparación de ambientes

|  |  |  |  |
| --- | --- | --- | --- |
| Característica | Local | Staging | Producción |
| Uso principal | Desarrollo | Pruebas reales | Uso público |
| Seguridad crítica | Baja | Media | Alta |
| Datos reales | No | Opcional | Sí |
| Acceso | Personal | Equipo interno | Usuarios reales |
| Estabilidad | Variable | Media-Alta | Alta |

Dividir el trabajo en estos entornos mejora la calidad del software, previene errores críticos en producción y permite iterar de forma controlada.

## ¿Cómo funciona una aplicación web en Internet?

Articulo: https://www.hostinger.com/ar/tutoriales/que-es-un-servidor-web

## APIs (Interfaz de Programación de Aplicaciones)

Articulo: https://lab.wallarm.com/what/api-frente-a-webhook-frente-a-websocket-cuando-es-mejor-utilizarlo/?lang=es

Una **API (Interfaz de Programación de Aplicaciones)** es un conjunto de definiciones y protocolos que permiten que distintos programas se comuniquen entre sí. Las APIs actúan como puentes entre aplicaciones, permitiendo que una parte del software acceda a funcionalidades o datos de otra sin necesidad de conocer su implementación interna.

Según la definición de MDN:

**API** es un conjunto de funciones y procedimientos que permiten la creación de aplicaciones que acceden a las características o datos de un sistema operativo, aplicación u otro servicio. [Fuente: MDN – Glosario API](https://developer.mozilla.org/es/docs/Glossary/API)

### Tipos de APIs

* **APIs internas o locales**: usadas dentro de una misma aplicación, entre módulos o componentes.
* **APIs del navegador**: ofrecidas por el entorno de ejecución de JavaScript en el navegador (como DOM, Web Storage, Fetch, etc.).
* **APIs externas o web APIs**: disponibles a través de internet, generalmente mediante el protocolo HTTP y datos en formato JSON.

### ¿Qué es una API?

Una API **define cómo interactuar con un sistema**: qué datos pueden solicitarse, cómo deben enviarse, y qué respuestas se esperan. Las APIs actúan como "contratos" entre quien las ofrece y quien las consume.

Ejemplo práctico:

* Vos desarrollás un sitio que muestra el clima.
* El sitio **consume una API de clima** que le proporciona los datos meteorológicos actualizados.
* El sitio **no necesita saber cómo se generan esos datos**, solo necesita saber **cómo pedirlos y cómo leerlos**.

Diagrama

El contenido generado por IA puede ser incorrecto.

### Cómo se utiliza una API en JavaScript

Una forma común de trabajar con APIs en JavaScript es usando [fetch()](https://developer.mozilla.org/es/docs/Web/API/Fetch_API), una función nativa que permite realizar solicitudes HTTP de manera asincrónica (GET, POST, etc.).

#### Estructura básica con fetch()

Texto

El contenido generado por IA puede ser incorrecto.

Explicación paso a paso:

1. fetch(url) hace la solicitud HTTP (por defecto, de tipo GET).
2. .then(response => response.json()) transforma la respuesta a un objeto JavaScript (formato JSON).
3. .then(data => { ... }) te permite trabajar con los datos recibidos.
4. .catch(error => { ... }) captura y maneja errores de red o de formato.

#### Funciones asincrónicas: async y await

En JavaScript, muchas operaciones no se resuelven instantáneamente (como acceder a una API). Para evitar que el código se detenga esperando una respuesta, usamos funciones asincrónicas.

#### ¿Qué son async y await?

* async: convierte una función en asincrónica y permite usar await dentro de ella.
* await: pausa la ejecución de la función hasta que una **promesa** se resuelva.

#### Ejemplo básico con async/await

Texto

El contenido generado por IA puede ser incorrecto.

###### Ventajas de async/await:

* Código más **claro y legible** que encadenar .then().
* Manejo más **ordenado de errores** con try...catch.
* Facilita la escritura de código asincrónico de manera estructurada.

🔗 Ver más en [MDN - async y await](https://developer.mozilla.org/es/docs/Learn/JavaScript/Asynchronous/Promises#async_y_await)

#### Errores comunes al trabajar con APIs

❌ Acceder a datos antes de que lleguen (resultados undefined)

❌ Usar .then() anidados en exceso (callback hell)

❌ No manejar errores de red o formato

❌ Asumir que la API siempre responderá correctamente

✅ Usar async/await junto con try...catch hace que el código sea **más estable, profesional y mantenible**.

### APIs públicas gratuitas para practicar

Estas APIs no requieren autenticación y son ideales para aprender

|  |  |  |
| --- | --- | --- |
| API | Descripción | URL Base |
| JSONPlaceholder | Datos falsos para testeo | https://jsonplaceholder.typicode.com |
| PokeAPI | Información sobre Pokémon | https://pokeapi.co/api/v2/ |
| Cat Facts | Curiosidades de gatos | https://catfact.ninja/fact |
| Open-Meteo | Datos meteorológicos por coordenadas | https://api.open-meteo.com |
| Dog CEO | Imágenes aleatorias de perros | https://dog.ceo/api/breeds/image/random |

Podés probar estas APIs con fetch() y mostrarlas en consola o en una página web con HTML + JavaScript.

### APIs Web integradas en JavaScript (nativas del navegador)

Además de conectarse con servicios externos, JavaScript puede usar **Web APIs** del navegador. Estas son interfaces que permiten interactuar con funcionalidades del dispositivo o el sistema.

|  |  |  |
| --- | --- | --- |
| **API Web** | **Funcionalidad principal** | **Documentación** |
| DOM | Manipular HTML y el contenido web | DOM |
| Fetch API | Realizar solicitudes HTTP | Fetch |
| Geolocation | Obtener ubicación del usuario | Geolocation |
| Web Storage | Guardar datos en el navegador | Web Storage |
| Canvas API | Dibujar gráficos y animaciones 2D | Canvas |

### Buenas prácticas al usar APIs

* Revisar la documentación oficial de la API que estás utilizando.
* Validar las respuestas recibidas antes de usarlas.
* Manejar errores con try...catch o .catch().
* Evitar peticiones innecesarias (usar almacenamiento local si es posible).
* Respetar los límites y políticas de uso de la API (rate limit, claves, etc.).
* Usar herramientas como Postman o Insomnia para probar APIs antes de implementarlas en código.

## Ejercicios prácticos

1. Primer fetch básico

Objetivo: Realizar una petición simple a una API y procesar la respuesta utilizando .then().  
Consigna: Usando fetch(), conectate a la API de Cat Facts y mostrala en consola. Es tu primer contacto con una API externa.

2. Primer uso de async/await

Objetivo: Reescribir el ejercicio anterior utilizando funciones asincrónicas modernas.  
Consigna: Usá async y await para obtener un dato curioso de gatos, manteniendo el mismo comportamiento pero con mejor legibilidad.

3. Diferencia entre código sincrónico y asincrónico

Objetivo: Demostrar que las operaciones asincrónicas no bloquean el flujo del programa.  
Consigna: Ejecutá un fetch() y mostrale al usuario cómo la consola imprime primero el inicio y fin del programa, antes de mostrar la respuesta de la API.

4. Múltiples llamados secuenciales

Objetivo: Ejecutar varias llamadas a una misma API de forma ordenada, esperando cada una.  
Consigna: Realizá tres solicitudes consecutivas a Cat Facts usando await y mostrá cada curiosidad en consola, con su número correspondiente.

5. Manejo de errores (simulado)

Objetivo: Implementar el manejo de errores de red.

Consigna: Simule un error usando una URL inválida. Usa try/catch para evitar que el programa falle y mostrarle al usuario el mensaje del error.

6. Uso de parámetros dinámicos en URLs

Objetivo: Construir URLs usando valores dinámicos.

Consigna: Hacer una función que reciba un id y obtenga los datos del usuario correspondiente desde JSONPlaceholder. Mostrar nombre y email.

7. Mostrar datos API en HTML

Objetivo: Visualizar la información de una API directamente en el navegador.  
Consigna: Obtener un post desde JSONPlaceholder y mostrarlo dentro de un div HTML como una tarjeta con título y cuerpo del post.

8. Llamadas paralelas con Promise.all

Objetivo: Ejecutar varias peticiones de forma simultánea y procesarlas juntas.  
Consigna: Realizar tres consultas a Cat Facts en paralelo y mostrarlas numeradas. Usa Promise.all() para ahorrar tiempo.

9. Procesamiento de datos anidados (JSON)

Objetivo: Acceder a datos dentro de estructuras JSON complejas.  
Consigna: Usa la API Open-Meteo para obtener el clima actual en Buenos Aires. Extrae la temperatura y velocidad del viento y mostradas en consola.

10. Entrada del usuario + visualización

Objetivo: Tomar datos del usuario para hacer una consulta y mostrar los resultados en pantalla.  
Consigna: Solicita al usuario el nombre de un Pokémon mediante un input. Consulta la PokeAPI y mostrá su altura, peso y nombre en pantalla.