Fundamentos de la Programación de Videojuegos

Lic. Facundo Nicolás Suárez

“Fundamentos de la Programación de Videojuegos”
Lic. Facundo Nicolás Suárez

Este texto fue recopilado de apuntes que han tomado mis alumnas y alumnos de las diferentes cursadas que realizaron conmigo en la materia que doy en la Universidad de José C. Paz llamada “Fundamentos de la Programación I” en la carrera de Producción y Desarrollo de Videojuego.

Por la elaboración de este material, quiero expresar mi agradecimiento a las alumnas y alumnos de las distintas camadas, quienes me facilitaron sus apuntes; a mis colegas docentes, que compartieron sus materiales —y en especial a Rod Gonzalez—; y a mi padre, Eduardo Suárez, quien no solo me apoyó desde pequeño y estimuló mi curiosidad por el arte electrónico, sino que también en esta etapa continúa acompañándome y aportando activamente a los materiales bibliográficos que elaboro.

Nota sobre la versión del documento
Este material corresponde a la segunda revisión, actualizada en **octubre de 2025**. La primera versión, elaborada en septiembre de 2025, circuló internamente en la universidad y contenía algunos errores de tipeo, omisiones menores y detalles pendientes de corrección, los cuales fueron revisados y ajustados en la presente edición. Se recomienda considerar la presente versión (octubre de 2025) como la versión más actual y revisada hasta el momento. Para asegurarse de contar con la última actualización disponible, se sugiere consultar periódicamente mi sitio web www.facundosuarez.com y/o mis redes sociales (@suarezfacuu).

Índice

Prólogo

Está por comenzar un nuevo cuatrimestre en este 2025, y me resulta inevitable hacerme muchas preguntas en este momento del año. En un mundo donde la inteligencia artificial se expande a pasos agigantados, la pregunta sobre el rol del programador en la sociedad vuelve a cobrar vigencia. La programación, que durante décadas fue sinónimo de escribir código línea por línea frente a una pantalla, hoy empieza a mutar hacia otra cosa: un ejercicio más cercano a la abstracción, la lógica y la resolución de problemas que a la mecánica de tipear instrucciones.

En este sentido, cabe preguntarse: ¿qué significa enseñar a programar en 2025? En la materia *Fundamentos de la Programación I* que damos en la carrera de videojuegos en la UNPAZ, por ejemplo, la tentación siempre está en comenzar por enfocarlo finalmente en lo concreto, por el código escrito, como si esa fuera la esencia misma del aprendizaje. Pero, ¿lo será en un futuro cercano?

Hace unos meses leía el libro **"¿Qué es la inteligencia artificial?" de Melisa Cecilia Avolio**, que plantea de forma clara y accesible cómo la IA ya está transformando no solo los oficios técnicos, sino la manera en que entendemos el trabajo humano. Si una máquina puede generar, optimizar e incluso depurar programas en cuestión de segundos, ¿qué lugar le queda al programador? Tal vez no el de "escribir", sino el de **pensar, diseñar, supervisar y decidir**.

La historia de la enseñanza de la programación parece repetirse. Mi padre, en su época de estudiante, aprendía a codificar con **tarjetas perforadas**. Era una práctica engorrosa, lenta y que demandaba una paciencia casi artesanal. En un momento, los docentes de su facultad decidieron dejar de usarlas: comprendieron que insistir en ese método era condenar a los alumnos a un conocimiento que ya no tendría vigencia. ¿No estaremos hoy frente a una disyuntiva parecida? ¿No es acaso escribir código a mano algo que pronto se volverá un ritual innecesario, una formalidad obsoleta frente a lo que ya hacen las IA?

El desafío docente, entonces, tal vez no sea insistir en la sintaxis de un lenguaje que mañana puede cambiar o ser automatizado, sino en formar **pensadores algorítmicos**. Quizás la clave esté en poner más énfasis en los diagramas de flujo, en la lógica de programación y en los fundamentos de cómo se plantea y resuelve un problema.

¿No será acaso más valioso enseñar a **diseñar la idea detrás del código**, en lugar de aferrarnos a la escritura del código mismo? ¿No será el momento de volver al inicio, a lo esencial, y enseñar menos teclas y más pensamiento?

¿Qué es la programación?

La **programación**, como disciplina fundamental en el **desarrollo de software**[1], abarca diversos conceptos esenciales que sientan las bases para la creación de soluciones programáticas efectivas. **A mí en lo personal, me gusta definir a la programación como la posibilidad de crear un universo**. Al programar, no solo se escriben instrucciones para que una máquina ejecute determinadas acciones, sino que se diseñan reglas, lógicas y dinámicas que configuran un espacio nuevo, un sistema con sus propias leyes internas.

De la misma forma en que un universo tiene sus principios físicos y sus condiciones de existencia, un programa establece estructuras, comportamientos y límites que dan vida a un entorno particular. Cada variable, cada función y cada interacción se convierten en piezas de un ecosistema que puede crecer y transformarse, siempre en función de la creatividad y las decisiones del programador. Programar es, en este sentido, un acto de creación que combina rigor lógico con imaginación, y que nos invita a ser arquitectos de realidades posibles. A continuación, exploraremos los elementos clave de los conceptos básicos para adentrarnos en el mundo de la programación.

Lenguajes Naturales	**Lenguajes Formales**
Los lenguajes naturales son aquellos que utilizamos para la comunicación entre seres humanos. Suelen ser ambiguos y estar abiertos a múltiples interpretaciones. Además, evolucionan de manera orgánica con el uso cotidiano. Son flexibles, permiten errores y excepciones sin impedir la comunicación. Por ejemplo, el Español, inglés, francés, chino, etc. Una frase como: *"Podemos vernos mañana a la tarde"* puede interpretarse como a las 14:00, 16:00 o en cualquier momento de la tarde.	Los lenguajes formales, en cambio, son los que describen instrucciones para ser ejecutadas por procesadores. Se caracterizan por estar libres de ambigüedad y ser totalmente precisos. Su evolución se da a través de actualizaciones y versiones bien definidas. A diferencia de los naturales, poseen una sintaxis rígida, donde los errores generan fallos en tiempo de ejecución. Por ejemplo, lenguajes de programación como Python, Javascript, C#, C++, etc. Si escribo "`print("Hola mundo")`" esta instrucción siempre va a mostrar exactamente el texto "Hola mundo", sin lugar a interpretaciones.

¿Qué es un lenguaje de programación?

Un **lenguaje de programación** es un conjunto de **reglas y símbolos** que permiten a los programadores escribir **instrucciones** que una computadora puede entender y ejecutar. Estos lenguajes actúan como intermediarios entre el pensamiento humano y la ejecución de acciones por parte de la máquina. Hay muchos lenguajes de programación diferentes, cada uno con sus propias reglas y sintaxis, diseñados para abordar distintos tipos de problemas y tareas. Los

[1] Aclaro esto porque prefiero no dar nada por sentado y explicarlo por las dudas: el **hardware** es la parte física de la computadora (por ejemplo, el monitor, el teclado o el disco rígido), mientras que el **software** es lo que utilizamos dentro de ella para darle una función, como el navegador Google Chrome para entrar a internet o Word para escribir textos (en otras palabras, los programas)

lenguajes de programación pueden clasificarse en dos categorías principales: **lenguajes de bajo nivel,** más cercanos al lenguaje de la máquina y específicos de la arquitectura de la computadora; y **lenguajes de alto nivel**, que son más abstractos y cercanos al lenguaje humano, facilitando la escritura y comprensión del código. Ejemplos de lenguajes de programación incluyen Python, Java, C++, JavaScript, entre muchos otros. Cada uno tiene sus propias características y se elige según las necesidades del proyecto y las preferencias del programador.

Niveles de abstracción

Lenguaje de Máquina	Lenguaje de Bajo Nivel	Lenguaje de Alto Nivel
Es el nivel más básico, compuesto únicamente por 0s y 1s. Está fuertemente ligado al hardware y presenta un **grado cero de abstracción**, ya que trabaja directamente con instrucciones binarias que la computadora entiende.	Ofrece instrucciones más cercanas al lenguaje humano en comparación con el de máquina, pero sigue estando muy ligado al hardware. Su nivel de abstracción es bajo, ya que aún requiere conocer detalles técnicos de la arquitectura de la computadora.	Se caracteriza por utilizar instrucciones en un lenguaje fácilmente entendible por las personas. Está más ligado al software que al hardware y presenta un **alto nivel de abstracción**, lo que permite centrarse en la lógica del problema sin preocuparse por los detalles técnicos de la máquina.

Sintaxis y Semántica

En los lenguajes de programación, elementos como las palabras reservadas, los operadores y los identificadores se combinan siguiendo un conjunto de reglas llamadas **sintaxis**, y con ellas se forman instrucciones. Las instrucciones, a su vez, tienen un significado asociado: representan una serie de acciones que el procesador debe realizar en tiempo de ejecución. A esta relación entre la instrucción y su acción se la denomina **semántica**.

Palabras reservadas
Los lenguajes de programación, tanto de alto como de bajo nivel, poseen palabras reservadas. Estas palabras ya están en uso por el propio lenguaje y no pueden emplearse para otro fin. Por ejemplo: `int`, `if`, `while`.

Operadores
Los operadores son símbolos que permiten realizar operaciones sobre los datos, ya sea aritméticas (como `+`, `-`, `*`, `/`) o de otro tipo, como la asignación (`=`) o el acceso a datos.

Identificadores
Los identificadores son nombres que el programador elige para reconocer procesos, funciones, variables o datos dentro del código fuente. Por ejemplo: `x`, `resultado`, `miFuncion`.

Ejemplo en C#:

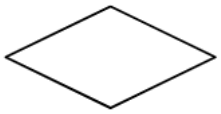
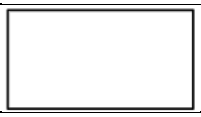
```
int x = 10;
```

Esta instrucción **reserva un espacio de memoria identificado con la letra `x` para almacenar números enteros, y asigna el valor `10` a ese espacio de memoria**.

- `int` → palabra reservada que indica que se va a usar un número entero
- `x` → identificador, es el nombre que damos a un espacio en memoria
- `=` → operador de asignación, que guarda un valor en ese espacio
- `10` → valor literal
- `;` → operador que indica el final de la instrucción (normalmente, en la mayoría de los lenguajes de programación, terminamos la línea de código con punto y coma)

Diagramación Lógica

Un **diagrama de flujo** (o flujogramas) es una forma visual de representar un algoritmo o proceso. Se usan símbolos para mostrar el orden y la lógica de las instrucciones. Todo diagrama de flujo debe tener un principio y un final. Para representar las acciones que realizará cada uno de los pasos de nuestro diagrama de flujo utilizaremos los siguientes símbolos:

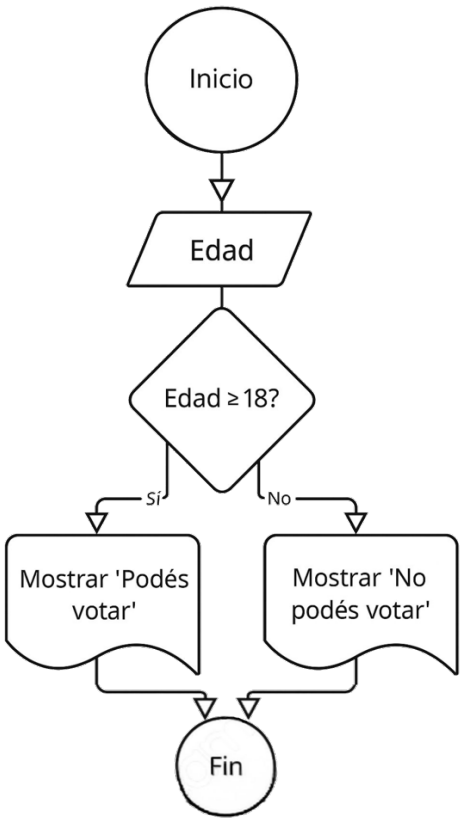
	Inicio y finalización del diagrama de flujo. Marcan el comienzo y la conclusión del proceso.
	Ingreso de datos. Representa la entrada de información al sistema.
	Salida de Información. Indica la entrega de resultados o datos hacia el exterior.
	Decisión o alternativa. Indica un punto dentro del flujo en donde se debe tomar una decisión entre dos o más opciones. Señala un punto en el que se debe elegir entre dos o más opciones posibles.
	Acción. Describe las operaciones quue se realizarán con los objetos del diagrama.

Símbolos básicos

- **Óvalo** → Inicio / Fin
- **Rectángulo** → Proceso o acción
- **Rombo** → Decisión (condición con "sí / no")
- **Flechas** → Flujo del proceso

A continuación, veamos un ejemplo genérico: **"**Si tengo 18 años o más, puedo votar."

```
[Inicio] ↓
[Ingresar Edad] ↓
[Edad ≥ 18?] → Sí → [Mostrar "Podés votar"] ↓ No
[Mostrar "No podés votar"] ↓
[Fin]
```



¿Para qué sirve hacer diagramas de flujo?

Puede servirnos para muchas cosas, pero considero que principalmente nos pueden servir para planificar la lógica antes de programar, evitar errores (o al menos, poder visualizarlos más fácilmente), y comprender mejor el flujo de decisiones.

Hoy en día, con el advenimiento de la inteligencia artificial, considero que el rol de los programadores está en plena transformación. Dado que la IA es capaz de resolver gran parte del trabajo técnico, repetitivo y estructurado, quizás nuestra tarea debería enfocarse más en el **diseño**, en imaginar qué cosas puede y debe hacer un programa, y en pensar en soluciones creativas a los problemas. La IA no debería ser vista como un reemplazo de los programadores, sino como una **herramienta complementaria**, que amplía nuestras capacidades y nos permite dedicar más tiempo a la parte conceptual, lógica y humana del desarrollo. De lo contrario, si nos limitamos a delegar todo a las máquinas sin una reflexión crítica, corremos el riesgo de convertirnos en usuarios pasivos de la tecnología, casi como los personajes de la película *Wall·E*: seres dependientes de sistemas inteligentes que deciden por nosotros. El verdadero desafío, entonces, está en usar la IA de manera consciente, para potenciar nuestra creatividad y no anularla. Es por eso que considero importante aprender los fundamentos de la programación desde cero, y los diagramas de flujo son parte esencial de esa base. Nos permiten entender cómo funcionan los programas más allá del lenguaje en que estén escritos, dándonos una visión clara de la lógica interna de los softwares que creamos.

Para realizar diagramas de flujo, hay varias opciones online, pero particularmente una que recomiendo es "Miro": https://miro.com/

¿Qué es el Pensamiento Computacional?

El pensamiento computacional es una **forma de abordar y resolver problemas** que proviene del campo de la informática, pero que puede aplicarse a cualquier área del conocimiento. Se trata de **pensar como lo haría una computadora**, pero no para convertirnos en máquinas, sino para **organizar nuestras ideas de forma clara, lógica y eficiente**. Esta forma de pensar se basa en **cuatro pilares** fundamentales:

1. Descomposición: Es el proceso de **dividir un gran problema en partes más pequeñas y manejables**. Por ejemplo, si queremos hacer un videojuego, no empezamos de golpe: lo separamos en tareas como "dibujar los personajes", "programar el movimiento", "crear la lógica del puntaje", etc.

2. Detección de patrones: Consiste en **buscar similitudes o repeticiones entre esos problemas más pequeños**. Esto nos permite reutilizar soluciones. Por ejemplo, si notamos que todos los enemigos del juego se mueven de forma parecida, podemos usar una misma función para controlarlos a todos.

3. Abstracción: Es el arte de **ignorar los detalles innecesarios** para enfocarnos en lo importante. Es como decir: "no importa si el personaje es un robot o un monstruo, ambos tienen que saltar cuando apretamos la barra espaciadora". Así, podemos pensar en una **solución general**.

4. Algoritmo: Finalmente, el algoritmo es **la secuencia ordenada de pasos** que seguimos para resolver un problema. Escribir un algoritmo es como armar una receta paso a paso: primero esto, después aquello. Es la base de cualquier programa.

Usar pensamiento computacional **no requiere saber programar**, pero aprender a programar sí mejora esta forma de pensar. Por eso, es una habilidad clave para diseñar videojuegos, resolver bugs, automatizar procesos y crear ideas con lógica y claridad.

Intérpretes y compiladores

Otra forma de clasificar a los lenguajes de programación de alto nivel es según cómo son traducidos a lenguaje máquina. Existen dos enfoques principales: los lenguajes interpretados y los compilados.

Lenguajes interpretados

En este caso, el código fuente se traduce **línea por línea** a lenguaje máquina en el momento de la ejecución. Para poder correr un programa interpretado, es necesario contar tanto con el código fuente como con el intérprete instalado en el sistema. Una característica importante es que, si el intérprete encuentra un error, la ejecución se detiene en ese punto. **Ejemplos:** Python (necesita el intérprete de Python instalado para correr el script), JavaScript (interpretado por los navegadores), PHP (El intérprete de PHP se instala en el servidor -por ejemplo, junto con Apache o Nginx-, y traduce el código cada vez que un usuario pide una página, generando dinámicamente la respuesta en HTML), etc.

Lenguajes compilados

En estos lenguajes, el código fuente es procesado por un **compilador** que lo traduce previamente a un archivo ejecutable. De esta forma, el programa ya está listo para ejecutarse directamente en la computadora sin necesidad del código original. Al estar pre-traducidos, suelen ejecutarse más rápido que los lenguajes interpretados. Además, se puede distribuir y utilizar solo el archivo ejecutable. **Ejemplos**: C (el compilador genera un .exe en Windows o un binario en Linux), C++, Java (aunque técnicamente compila a *bytecode* y luego necesita la **JVM** - Máquina Virtual de Java- para ejecutarse, se lo considera del lado compilado porque primero traduce el código fuente a ese formato intermedio antes de correr)

Aunque tradicionalmente **JavaScript** se considera un lenguaje interpretado porque el navegador ejecuta directamente el código fuente, en la práctica los navegadores modernos utilizan motores que aplican **compilación JIT (Just In Time)**. Esto significa que, en lugar de limitarse a leer e interpretar línea por línea, el motor traduce partes del código a lenguaje máquina "sobre la marcha" mientras el programa se ejecuta, lo que mejora notablemente la velocidad. Por eso, se lo sigue clasificando como interpretado, pero con la salvedad de que combina técnicas de compilación para optimizar el rendimiento. En otras palabras: hoy en día, en realidad, los navegadores modernos no lo ejecutan solo línea por línea. Lo que hacen es usar una técnica **(Just In Time)**, haciendo que JavaScript combine lo mejor de los dos mundos: mantiene la flexibilidad de los lenguajes interpretados, pero con la velocidad extra que le da la compilación.

Paradigmas de programación

Los paradigmas de programación pueden definirse como **enfoques para estructurar los programas y las soluciones** que desarrolla un programador. Cada paradigma ofrece un marco conceptual que brinda ciertas posibilidades y estrategias a la hora de enfrentar un desarrollo. No todos los paradigmas están disponibles en todos los lenguajes de programación. Por eso, es importante que un desarrollador pueda elegir de manera correcta el lenguaje y el paradigma que utilizará al momento de encarar un proyecto.

Programación Imperativa
Este paradigma se enfoca en el estado del programa. En tiempo de ejecución, dicho estado se modifica para obtener el resultado esperado. La programación imperativa se basa en comandos estructurados y secuenciados que se manejan con estructuras de control de flujo, como **condicionales** (if, else) y **bucles** (for, while).

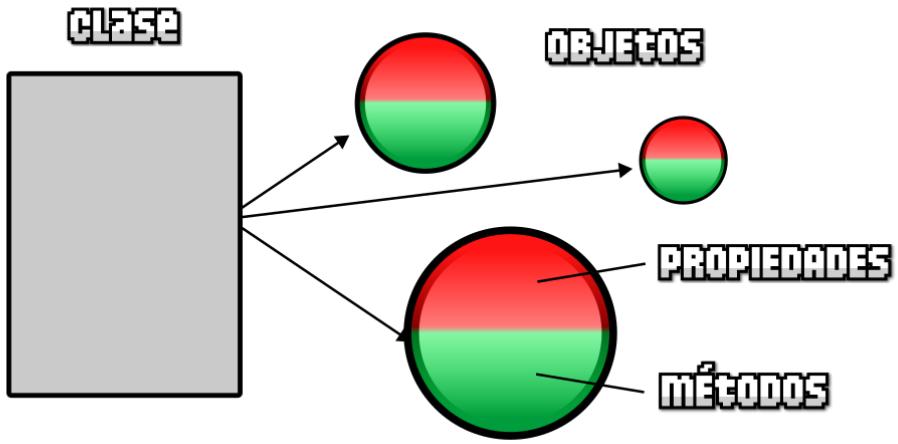
Las principales ventajas de la programación imperativa tienen que evita la repetición de código gracias al uso de funciones o procedimientos, lo que permite organizar mejor las instrucciones y facilitar su lectura y mantenimiento.

Programación Orientada a Objetos (POO)
En el desarrollo de videojuegos, la **programación orientada a objetos** es el paradigma que más utilizamos en la actualidad. La POO se basa en el concepto de **objetos**, que son **instancias de clases**. Una clase funciona como un **molde o plantilla** que define cómo serán los objetos: qué datos tendrán (propiedades o atributos) y qué acciones podrán realizar

(métodos o funciones). Un objeto, en cambio, es la **materialización concreta** de esa clase en el programa.

- **Clase**: plantilla que define las características y comportamientos comunes.
  Ejemplo: Personaje.
- **Objeto**: una instancia[2] de la clase, con datos propios. Ejemplo: heroe = new Personaje().
- **Propiedades**: atributos que describen al objeto. Ejemplo: heroe.vida = 100.
- **Métodos**: funciones que definen acciones que puede realizar el objeto. Ejemplo: heroe.saltar().

La POO se centra en cuatro principios clave: **abstracción** (crear modelos de lo esencial), **encapsulación** (proteger los datos dentro del objeto), **herencia** (permitir que una clase derive de otra) y **polimorfismo** (dar distintos comportamientos a un mismo método según el contexto). Gracias a estas características, la programación orientada a objetos resulta especialmente útil en videojuegos, donde los distintos elementos del juego (jugadores, enemigos, ítems, escenarios) pueden representarse como objetos que interactúan entre sí dentro del mundo virtual.



[2] Una *instancia de una clase* es un objeto concreto creado a partir de la definición de una clase. Mientras que la clase funciona como un molde o plano general (por ejemplo, la receta de una torta), la instancia es el objeto real en memoria que tiene sus propios valores (la torta que realmente horneamos con esa receta).

Algoritmos y Programas

Un **algoritmo** es una serie de pasos precisos que se siguen para resolver un problema. Bien podríamos compararlo con una **receta de cocina**, donde se aclaran los ingredientes, procedimientos, cantidades, tipo y tiempo de cocción. Del mismo modo, nuestro algoritmo debe brindar la información precisa para la resolución del problema planteado. Llamamos **programa** a uno o varios algoritmos escritos en un lenguaje de programación, que pueden ser ejecutados por una computadora para realizar una tarea específica.

Solución de problemas

Dijimos que los programadores resuelven problemas, pero lo hacen de una forma particular: a través de una computadora. Para eso se requiere que el programador escriba un programa, lo que significa que se deben realizar, al menos, estas **cuatro fases**:

1. **Analizar el problema.**
2. **Diseñar un algoritmo que lo resuelva.**
3. **Codificar ese algoritmo en un lenguaje de programación.**
4. **Depurar y validar la solución.**

Fase 1 – Análisis del problema	**Fase 2 – Creación de un algoritmo**	**Fase 3 y 4 – Codificación del algoritmo / Depuración**
En esta etapa debemos comprender el problema y responder algunas preguntas: • ¿Qué entradas se requieren? • ¿Qué salidas se esperan? • ¿Qué procesos pueden generar esa salida? • ¿Existen requisitos adicionales o limitantes en nuestro problema?	En la fase 1 definimos **qué va a hacer** nuestro programa. En la fase 2 determinamos **cómo lo va a hacer**. Para esto solemos usar la estrategia de dividir el problema en problemas más pequeños (subproblemas), que puedan ser resueltos con pocas acciones. Cada subproblema tiene entradas y salidas propias.	Una vez diseñado el algoritmo, debemos **traducirlo** a un **lenguaje de programación**, eligiendo aquel que mejor se adapte a nuestro proyecto y al problema que queremos resolver. En esta fase pasamos de la lógica abstracta del algoritmo a un código concreto que pueda ser interpretado o compilado por la computadora. Posteriormente, en la etapa de depuración, debemos **revisar** y probar nuestro programa **para detectar errores**, tanto de sintaxis como de lógica, asegurándonos de que funcione correctamente y cumpla con lo que se había planteado en el análisis inicial. Esta fase es clave porque nos permite validar que la solución implementada realmente resuelve el problema de manera eficiente.

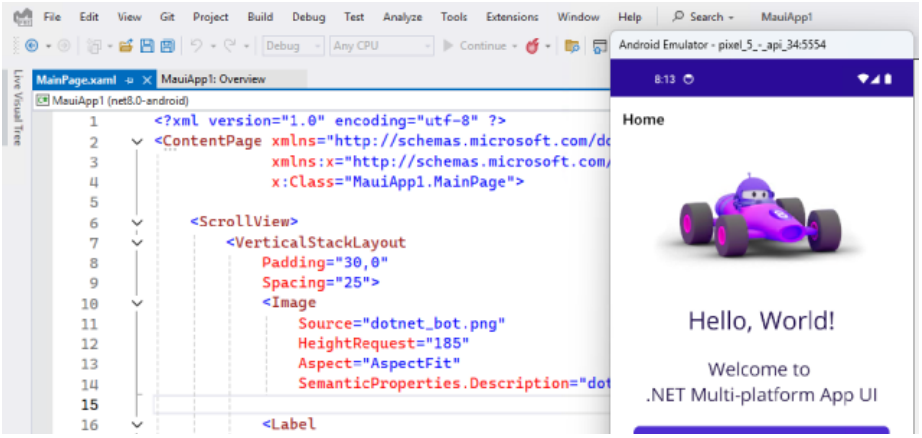
Ejemplo:
Queremos crear un programa que realice la suma de cinco números.
 	
• **Entradas requeridas:** cinco números.
• **Salida esperada:** el resultado de la suma.
• **Proceso:** sumar los cinco números.
• **Limitantes:** deben ser números válidos.
	Ejemplo de algoritmo en pseudocódigo:
	
	Inicio
	Leer número1, número2, número3, número4, número5
	Sumar número1 + número2 + número3 + número4 + número5
	Mostrar resultado suma
	Fin
		Ejemplo en Javascript:
		
		`// Programa para sumar cinco números`
		`let numero1 = 4;`
		`let numero2 = 7;`
		`let numero3 = 10;`
		`let numero4 = 3;`
		`let numero5 = 6;`
		
		`let suma = numero1 + numero2 + numero3 + numero4 + numero5;`
		
		`console.log("La suma de los cinco números es: " + suma);`

Es importante entender que entonces, en definitiva, un algoritmo es un método por el cual vamos a realizar una tarea (o resolver un problema) mediante una serie de pasos **precisos, definidos y finitos**. Para que un algoritmo sea **preciso**, debe indicar el orden de realización de las tareas paso a paso, de manera que puedan seguirse de principio a fin sin ambigüedades. Un algoritmo es **definido** cuando, siempre que se ejecute, produce el mismo resultado: a igual entrada, igual salida. Finalmente, debe ser **finito**, es decir, contar con un número determinado de pasos que garanticen que en algún momento el proceso concluirá.

Ambientes de Programación

Los **ambientes de programación** son entornos en los que los programadores desarrollan, prueban y depuran software. Estos ambientes incluyen todas las herramientas y utilidades que facilitan el proceso de desarrollo, y pueden abarcar desde simples editores de texto hasta completos **entornos integrados de desarrollo (IDE)**.

Un ejemplo de IDE es **Visual Studio**, una herramienta muy utilizada en el ámbito académico y profesional. Durante la cursada de *Fundamentos de la Programación I* solemos trabajar con **Visual Studio Community**, una versión gratuita que se puede descargar desde la página oficial: https://visualstudio.microsoft.com/es/downloads/. Este entorno ofrece numerosas funcionalidades como resaltar la sintaxis, autocompletar código, depuración y ejecución de programas, lo que facilita enormemente el aprendizaje y la práctica.



En mi experiencia personal, también suelo utilizar otros ambientes de programación según el tipo de proyecto. Por ejemplo, uso **Brackets** (https://brackets.io/) cuando trabajo en Unity o hago desarrollo web utilizando HTLM, CCS o Javascriptm dado que es de código abierto, gratuito y bastante cómodo de utilizar; y empleo el **IDE de Arduino** (https://www.arduino.cc/en/software/) cuando desarrollo proyectos con la placa Arduino o con la **Raspberry Pi Pico**. No obstante, es importante remarcar que en realidad el código puede escribirse incluso en un simple **Bloc de notas**. La diferencia está en que los IDE y editores avanzados nos brindan herramientas que agilizan y mejoran nuestro trabajo: resaltan errores, organizan el código con colores, permiten testear y depurar, e incluso ejecutar los programas de manera directa.

Variables y Constantes

Las **variables** son contenedores que almacenan información **modificable** durante la ejecución del programa. Las **constantes**, en cambio, son **valores inmutables**.

Los **tipos de datos** definen la **naturaleza de la información** que puede **almacenarse** en una **variable**. Entre ellos, se encuentran String (cadenas de texto), Boolean (valores lógicos), Number (números), Null (valor nulo), entre otros.

Numéricos

- **Enteros (int):**

Representan **números enteros** sin decimales, como -3, 0, 42.

```
int cantidadDeEstudiantes = 30;
int temperaturaActual = -5;
```

- **Punto Flotante (float):**

Representan **números con decimales**, como 3.14, -0.5.

```
float precioProducto = 19.99f;
float alturaMontaña = 4567.89f;
```

En muchos lenguajes de programación, cuando escribimos un número en un programa, este número se conoce como literal. En el caso de los números decimales, hay dos tipos principales en la mayoría de los lenguajes de programación: *float* y *double*. La "f" al final de un número decimal indica que ese número debe tratarse como un tipo *float*. Es decir, para indicar explícitamente que un número decimal es del tipo *float* en lugar de *double*, se añade la letra "f" al final del número. La distinción es importante en algunos casos para evitar conversiones implícitas y para garantizar que el tipo de dato sea el esperado.

- **Números Decimales (double):**

Los números decimales, representados por el tipo de dato **double**, son valores que pueden contener números con decimales.

```
double precioProducto = 19.99;
double alturaTorre = 345.67;
```

La diferencia principal entre ***double*** y ***float*** radica en la precisión y el rango de valores que pueden representar. El tipo ***double*** ofrece mayor precisión y un rango más amplio que ***float***, por lo tanto, ***double*** es más adecuado para situaciones donde se requiere una alta precisión en cálculos numéricos o se manejan valores extremadamente grandes o pequeños. Sin embargo, este beneficio viene acompañado de un mayor consumo de memoria. En contraste, ***float*** puede ser más eficiente en términos de memoria y es adecuado cuando la precisión extendida de ***double*** no es crucial. La elección entre ***double*** y ***float*** depende de la naturaleza de los datos y los requisitos específicos de cada situación.

Texto

- **Cadenas de Texto (string):**

Secuencias de caracteres, como "Hola, mundo!".

```
string saludo = "Hola, mundo!";
string nombre = "Juan";
```

- **Caracteres (char):**

Representa un único carácter.

```
char primeraLetra = 'A';
```

Booleanas

- **Booleanos (bool):**

Representan valores de verdad o falsedad (true o false).

```
bool esDiaSoleado = true;
bool tareaCompletada = false;
```

Estructuradas / Colecciones[3]

- **Listas:**

Colecciones ordenadas de elementos, cada uno identificado por un índice.

```
List<int> numeros = new List<int>() { 1, 2, 3, 4, 5 };
List<string> palabras = new List<string>() { "manzana", "banana", "cereza" };
```

- **Tuplas:**

Similar a las listas, pero inmutables, es decir, no se pueden modificar después de su creación.

```
Tuple<int, string> persona = new Tuple<int, string>(25, "Juan");
```

- **Arreglos (Array):**

Estructuras que almacenan elementos del mismo tipo en posiciones contiguas de memoria.

```
int[] numerosEnteros = { 1, 2, 3, 4, 5 };
```

Especiales

- `null / None`: representa ausencia de valor
- `undefined`: (en JavaScript) variable declarada pero sin valor asignado
- `dynamic o var`: tipo que se determina en tiempo de ejecución (C#, JS, Python)

---

[3] Además de los tipos de datos vistos, Python ofrece **set** (conjunto) y **dictionary** (diccionario, o map), entre otros. Un **set** almacena elementos únicos y sin orden, por ejemplo `{"rojo", "verde", "azul"}`, mientras que un **dictionary** guarda **pares clave-valor**, como `{"nombre": "Facu", "edad": 28}`. Ambas estructuras facilitan el manejo de datos; pero los tipos más usados siguen siendo los anteriores, y es por ese motivos que no los resalté tanto.

¿Se escriben igual las variables y constantes en todos los lenguajes de programación?

Una de las primeras preguntas que suelen surgir al comenzar a programar es si las **variables y constantes** se escriben igual en todos los lenguajes de programación. La respuesta es que, si bien el concepto es el mismo —espacios de memoria que guardan información—, la forma de **declararlas y utilizarlas** puede variar según el lenguaje. Cada lenguaje tiene su propia sintaxis, palabras reservadas y convenciones, lo que hace necesario adaptarse a la forma particular en que cada uno define variables y constantes.

Variables	Constantes

JavaScript
`let nombre = "Facu";`
`let edad = 25;`
	JavaScript
	`const PI = 3.1416;`

C#
`string nombre = "Facu";`
`int edad = 25;`
	C#
	`const double PI = 3.1416;`

Python `nombre = "Facu" edad = 25`	Python `PI = 3.1416 # Por convención, se escribe en mayúsculas aunque no es realmente constante`

Aclaración de Phyton

En Python, no existe una forma estricta de declarar constantes como en otros lenguajes. Sin embargo, por convención, se utiliza el nombre en mayúsculas (por ejemplo, PI = 3.1416) para indicar que ese valor no debería cambiarse durante la ejecución del programa. Esta convención sirve como señal para otros programadores de que la variable debe tratarse como constante, aunque técnicamente Python sí permite modificar su valor más adelante. Es una regla de estilo y buenas prácticas, no una restricción del lenguaje.

Aclaración de Javascript

¿Por qué se usa `let` y no `var`? Antes de ES6 (una versión moderna de JavaScript lanzada en 2015), se usaba var para declarar variables. Pero `let` (y también `const`) se introdujeron porque son más seguros y predecibles:

- `let` tiene alcance de bloque, lo que significa que solo existe dentro de las llaves {} donde fue declarada.
- `var` tiene alcance de función, lo que puede causar errores difíciles de encontrar.

¿Y cuál es la diferencia con const?

- `let` → se usa cuando el valor puede cambiar.
- `const` → se usa cuando el valor no debe cambiar.

Asignación de Variables en Programación

La **asignación de variables** es un concepto esencial en programación que implica **almacenar un valor** en una ubicación de memoria asociada a un nombre simbólico o identificador. Este proceso es crucial para manipular y gestionar datos en un programa. A continuación, se explica el significado de asignar una variable y se proporcionan ejemplos en varios lenguajes de programación.

¿Qué significa asignar una variable?

Asignar una variable implica almacenar un valor en una ubicación de memoria identificada por un nombre. Las variables son utilizadas para almacenar datos que pueden cambiar durante la ejecución del programa. La sintaxis básica para asignar un valor a una variable varía según el lenguaje de programación, pero generalmente implica el uso del operador de asignación ("=").

Ejemplos de asignación en diferentes lenguajes:

- En C#:
```
int edad; // Declaración de la variable
edad = 25; // Asignación de un valor a la variable
```

- En Python:
```
edad = 25 # Asignación de un valor a la variable
```

- En JavaScript:
```
let edad; // Declaración de la variable
edad = 25; // Asignación de un valor a la variable
```

Cuando asignas un valor a una variable, estás almacenando ese valor en la memoria de la computadora y asociándolo con el nombre de la variable. Esto permite que el programa acceda y manipule ese valor en diversas partes del código. La capacidad de cambiar el valor de una variable a lo largo del programa es lo que distingue a las variables de las constantes, que tienen un valor fijo y no cambian después de su asignación inicial.

Crear Instancias

El concepto de "Crear instancias" se refiere a generar objetos específicos a partir de una clase (plantilla que define datos y comportamientos) en programación orientada a objetos. Una clase es como un plano o plantilla que define qué **datos** (**propiedades**) y qué **comportamientos** (**métodos**) tendrá un objeto. Por ejemplo, utilizando el lenguaje C#, definimos una clase `Persona` con dos propiedades: `Nombre` y `Edad`.

```
// Definición de la clase "Persona"
public class Persona
{
    public string Nombre { get; set; } // Propiedad para el nombre
    public int Edad { get; set; }      // Propiedad para la edad
}
```

Una **instancia** es un objeto concreto creado a partir de esa clase. Cada instancia tiene su propia copia de las propiedades, que podemos modificar sin afectar a otras instancias.

```
// Creamos una instancia de la clase "Persona"

Persona persona1 = new Persona(); // persona1 es un objeto concreto

// Asignamos valores a sus propiedades

persona1.Nombre = "Juan";
persona1.Edad = 30;
```

En este ejemplo, `persona1` es una **instancia** de `Persona`. Podríamos crear otra instancia `persona2` con otros valores, y ambas coexistirían independientemente.

En otras palabras: Crear instancias en programación orientada a objetos significa generar objetos concretos a partir de una clase, que actúa como una plantilla definiendo sus datos (propiedades) y comportamientos (métodos). Por ejemplo, en C# se puede definir la clase `Persona` con las propiedades `Nombre` y `Edad`. Al crear una instancia como `persona1 = new Persona()`, se genera un objeto específico que tiene su propia copia de las propiedades, que podemos modificar sin afectar a otras instancias. Así, podríamos crear otra instancia `persona2` con valores diferentes, y ambas coexistirían de manera independiente.

En la programación de videojuegos, crear instancias nos permite generar múltiples objetos a partir de una misma clase sin tener que definir cada uno desde cero. Por ejemplo, podemos tener una clase `Enemigo` y crear muchas instancias con distintas posiciones o comportamientos, o una clase `Moneda` para colocar coleccionables en distintos lugares del mapa. Cada bala que dispare el jugador puede ser una instancia de la clase `Bala`, con sus propias propiedades como velocidad o dirección. Esto nos ayuda a ahorrar código, mantener el orden y controlar cada objeto de manera independiente, algo fundamental para que los juegos funcionen correctamente y sean escalables.

¿Declarar y asignar una variable es lo mismo?

Una pregunta muy común al empezar a programar es si **declarar y asignar una variable es lo mismo**. La respuesta es que no: **declarar** una variable significa decirle al programa que existe y qué tipo de datos puede contener, por ejemplo `int edad;` (en C#). En cambio, **asignar** es darle un valor concreto a esa variable, como `edad = 30;`.

Igualmente, muchas veces se declara y se asigna al mismo tiempo, si por ejemplo escribimos directamente y por primera vez en el código "`int edad = 30;`", pero es importante entender que declarar crea el espacio en memoria y asignar coloca el valor dentro de ese espacio.

¿Y cuál es la diferencia entre asignar valores a variables y crear instancias de clases?

Son cosas distintas. En resumen, **asignar** es dar un valor a una variable, y **crear instancias** es construir objetos basados en clases para manejar datos y funcionalidades en programas.

¿Qué nos permite hacer un lenguaje de programación?

Un lenguaje de programación (como JavaScript, Python o C#) permite escribir instrucciones que la máquina puede ejecutar. Con un lenguaje de programación podés hacer muchas cosas, pero lo fundamental es que nos permite:

- Tomar decisiones (if/else)
- Repetir acciones (for, while)
- Manipular datos (variables, funciones, arrays)
- Interactuar con el usuario

En otras palabras, los lenguajes de programación son sistemas de reglas y símbolos que nos permiten escribir instrucciones que la computadora puede entender y ejecutar. Aunque las posibilidades son muchas, podemos resumir las más importantes en las páginas siguientes, sin importar el lenguaje que estemos utilizando:

¿Qué es un ciclo for?

Un ciclo for permite repetir un bloque de código varias veces, controlando el número de repeticiones.

Javascript	C#	Python
`for (let i = 0; i < 5; i++) { console.log("Iteración número: " + i); }`	`for (int i = 0; i < 5; i++) { Console.WriteLine("Iteración número: " + i); }`	`for i in range(5): print("Iteración número:", i)`

Usamos un **ciclo for** cuando queremos repetir un bloque de código un número determinado de veces, especialmente si sabemos cuántas iteraciones necesitamos. Por ejemplo, de manera genérica, podemos imprimir los números del 1 al 5 en la consola con `for i in range(1, 6): print(i)`. En videojuegos, un `for` nos sirve para generar múltiples objetos de manera automática, como crear 10 monedas en distintas posiciones del mapa. El siguiente ejemplo está escrito en C#:

```
// Creamos 10 monedas alineadas horizontalmente cada 50 unidades en el eje X
for (int i = 0; i < 10; i++)
{
    CrearMoneda(x: i * 50, y: 100); // Llama a la función que genera la moneda
}
```

Vamos con la explicación paso a paso. Primero vamos a analizar en profundidad "**`for (int i = 0; i < 10; i++)`**"

1. **`int i = 0` → Inicialización**
   - Acá se declara la variable **`i`** y se le asigna el valor inicial **`0`**.
   - **`i`** actúa como **contador** que controla cuántas veces se ejecutará el ciclo.
2. **`i < 10` → Condición**
   - Esta es la **condición de continuación del ciclo**.
   - Mientras esta condición sea verdadera, el ciclo seguirá ejecutándose.
   - En este caso, el ciclo se repetirá mientras i sea menor que 10 (es decir, desde 0 hasta 9).

3. **`i++` → Incremento**
    - Después de cada iteración, se ejecuta esta instrucción.
    - **`i++`** significa aumentar **`i`** en 1. Es una forma resumida de poner: "**`i = i + 1`**"
    - Esto asegura que eventualmente la condición **`i < 10`** dejará de cumplirse y el ciclo terminará.

Resumen del flujo:

- Se **inicializa** el contador **`i = 0`**.
- Se **evalúa la condición `i < 10`**.
    - Si es verdadera → se ejecuta el bloque de código dentro del for.
    - Si es falsa → el ciclo termina.
- Se **ejecuta el incremento `i++`** y se repite la evaluación de la condición.

Por lo tanto: **`for (int i = 0; i < 10; i++)`**
- Crea un ciclo que se repite 10 veces.
- **`i`** es un contador que empieza en **`0`** y termina en **`9`**.
- Cada iteración aumenta **`i`** en 1 (**`i++`**).

2**`. CrearMoneda(x: i * 50, y: 100);`**
- Llama a la función **`CrearMoneda`** en cada iteración.
- El parámetro **`x`** se calcula como **`i * 50`**, por lo que cada moneda queda separada **`50`** unidades en el eje horizontal.
- El parámetro **`y`** se mantiene constante en **`100`**, alineando las monedas en la misma altura.

¿Qué es un condicional?

Los condicionales permiten que el programa tome decisiones. La forma más común es el uso de ***if***, que evalúa si una condición se cumple y ejecuta un bloque de código en respuesta. También existen otras estructuras como **else**, **else if** y **switch**, que permiten manejar múltiples caminos posibles según diferentes valores o situaciones. Para combinar condiciones o negar resultados, se utilizan operadores lógicos, como:

- && (AND): ambas condiciones deben cumplirse.
- || (OR): al menos una condición debe cumplirse.
- ! (NOT): niega una condición, es decir, devuelve verdadero si la condición es falsa.

JavaScript	C#	Python

`let edad = 20;`
`if (edad >= 18) {`
`  console.log("Sos mayor de edad");`
`} else {`
`  console.log("Sos menor de edad");`
`}`
	`int edad = 20;`
	`if (edad >= 18) {`
	`  Console.WriteLine("Sos mayor de edad");`
	`} else {`
	`  Console.WriteLine("Sos menor de edad");`
	`}`
		`edad = 20`
		`if edad >= 18:`
		`  print("Sos mayor de edad")`
		`else:`
		`  print("Sos menor de edad")`

¿Qué es un array?

Un array es una lista de elementos ordenados, que pueden ser accedidos por su posición (índice).

Javascript	C#	Phyton

`let frutas = ["manzana", "banana", "naranja"];`
`console.log(frutas[0]); // imprime "manzana"`
 	
`for (let i = 0; i < frutas.length; i++) {`
`  console.log(frutas[i]);`
`}`
	`string[] frutas = { "manzana", "banana", "naranja" };`
	`Console.WriteLine(frutas[0]); // imprime "manzana"`
	
	`for (int i = 0; i < frutas.Length; i++) {`
	`  Console.WriteLine(frutas[i]);`
	`}`
		`frutas = ["manzana", "banana", "naranja"]`
		`print(frutas[0])  # imprime "manzana"`
		
		`for fruta in frutas:`
		`  print(fruta)`

¿Para qué sirve?

- Para guardar múltiples valores en una sola variable
- Para recorrer y manipular datos

Supongamos que queremos **guardar los nombres de los enemigos** en un nivel y luego mostrar cada uno en la consola:

```
// Definimos un array con los nombres de los enemigos
string[] enemigos = { "Goblin", "Orco", "Troll", "Dragón" };

// Mostramos el primer enemigo
Console.WriteLine(enemigos[0]); // imprime "Goblin"

// Recorremos todos los enemigos y mostramos sus nombres
for (int i = 0; i < enemigos.Length; i++) {
    Console.WriteLine("Enemigo en la posición " + i + ": " + enemigos[i]);
}
```

- **`enemigos`** es un array que almacena los nombres de todos los enemigos.
- Podemos acceder a cada enemigo usando su índice, empezando desde 0.
- El **`ciclo for`** nos permite **recorrer todo el array** para realizar acciones con cada enemigo, como mostrarlos en pantalla o asignarles posiciones en el mapa.

Cuando estamos desarrollando videojuegos, los arrays pueden usarse para **almacenar enemigos de un nivel**, **guardar los nombres de los jugadores en un ranking**, **mantener las posiciones de objetos coleccionables como monedas o power-ups**, **registrar los valores de puntuación o vidas de cada jugador**, y **controlar animaciones mediante la secuencia de sprites de un personaje**, entre otros. Todos estos ejemplos permiten manejar múltiples datos de manera organizada y recorrerlos fácilmente con ciclos.

¿Qué es recorrer un array?

```
for (let i = 0; i < frutas.length; i++) {
console.log(frutas[i]);
}
```

Recorrer un array significa acceder, uno por uno, a cada elemento que contiene ese array, generalmente usando una estructura repetitiva como un bucle `for` o `while`. Es una forma de "pasar por cada casillita" del conjunto de datos que el array almacena. ¿Para qué sirve en un videojuego? En un videojuego, recorrer un array es útil para muchas cosas, por ejemplo, dibujar todos los enemigos en pantalla, revisar si alguno de los proyectiles colisionó con un enemigo, actualizar la posición de varios objetos a la vez (como monedas, enemigos, partículas, etc.), aplicar efectos a varios personajes o detectar eventos (por ejemplo, si todos los objetivos fueron alcanzados), etc. En resumen, recorrer un array permite manejar múltiples elementos de forma organizada y eficiente, algo fundamental en el desarrollo de videojuegos.

Un **array** permite almacenar múltiples valores en una sola variable.

Características principales:

- **Acceso directo:** con un índice (`array[i]`).
- **Tamaño fijo:** definido al crearse.
- **Homogeneidad:** todos los elementos son del mismo tipo.
- **Memoria contigua:** los datos se almacenan juntos, lo que los hace eficientes.

Ejemplo en C#:

```
int[] numeros = { 10, 20, 30, 40 };
Console.WriteLine(numeros[2]); // Accede al tercer elemento: 30
```

Recordá que si, por ejemplo, el array tiene 4 elementos (índices 0–3), y pedimos **`numeros[4]`** nos va a dar error.

Estructuras de Datos

Las **estructuras de datos** son formas organizadas de almacenar y gestionar información dentro de un programa. Nos ayudan a que el código sea más claro y eficiente. Existen muchos tipos y elegir la adecuada es clave para que nuestras soluciones funcionen bien.

Estáticas y Dinámicas

Estáticas	**Dinámicas**
Su tamaño lo define el programador y no cambia en ejecución.	Su tamaño puede crecer o reducirse durante la ejecución.
Ventaja: Uso eficiente de memoria. Desventaja: No se puede modificar su tamaño. Ejemplo: **Arrays**.	Ventaja: Se adaptan a datos variables. Desventaja: Menor velocidad y eficiencia. Ejemplo: **Listas, Pilas, Árboles**.

Tipos Abstractos de Datos (TAD)

Los **Tipos Abstractos de Datos (TAD)** son modelos que definen qué tipo de datos se manejan y qué operaciones pueden realizarse sobre ellos, sin necesidad de conocer cómo están implementados internamente. Es decir, se enfocan en el *qué hacen* y no en el *cómo lo hacen*. Gracias a esta abstracción podemos trabajar de forma más clara y ordenada, ya que utilizamos una interfaz definida sin preocuparnos por los detalles técnicos de su construcción. Por ejemplo, el TAD **Lista** permite agregar, quitar, buscar o limpiar elementos, sin que el programador tenga que saber si internamente la lista está implementada con un array, una estructura enlazada u otro mecanismo.

Strings como estructura de datos

Un **string** es una cadena de caracteres (`char`) ordenados y accesibles por índice:

Ejemplo: `"Hola"`

- H (índice 0)
- o (índice 1)
- l (índice 2)
- a (índice 3)

Podemos recorrer cada carácter con un bucle.

Métodos útiles de String

- `"Pedro".ToLower()` → `"pedro"`
- `"Ramon".Contains("mon")` → `True`
- `"Ramon".Insert(1,"1234")` → `"R1234amon"`
- `"Ramon".Substring(2)` → `"mon"`
- `"Ramon".Remove(2)` → `"Ra"`

Ejemplo en C#

Función que recibe un string y devuelve la cantidad de palabras:

```
int ContarPalabras(string texto)
{
    string[] palabras = texto.Split(' '); // Divide el string en palabras
    return palabras.Length; // Devuelve la cantidad
}
```

Enumeraciones (enum)

Las **enumeraciones** (o *enums*) son un tipo de dato que permite definir un conjunto de valores constantes con nombre. Son muy útiles cuando tenemos un grupo fijo de opciones relacionadas, ya que hacen el código más legible, reducen errores de escritura y facilitan el mantenimiento. Con un **`enum`** aseguramos que una variable solo pueda tomar uno de los valores definidos en esa lista.

Ejemplo básico en C#

```
// Definimos una enumeración de enemigos
enum Enemigo { Gargola, Golem, Salvaje, Canibal }

class Program
{
    static void Main()
    {
        // Creamos una variable del tipo enum y le asignamos un valor
        Enemigo enemigo = Enemigo.Canibal;

        Console.WriteLine("El enemigo seleccionado es: " + enemigo);
    }
}
```

Ejemplo con switch

```
enum Enemigo { Gargola, Golem, Salvaje, Canibal }

class Program
{
    static void Main()
    {
        Enemigo enemigo = Enemigo.Salvaje;

        switch (enemigo)
        {
            case Enemigo.Gargola:
                Console.WriteLine("La gárgola ataca desde el aire.");
                break;

            case Enemigo.Golem:
                Console.WriteLine("El gólem es fuerte pero lento.");
                break;

            case Enemigo.Salvaje:
                Console.WriteLine("El salvaje lucha con agilidad y ferocidad.");
                break;

            case Enemigo.Canibal:
                Console.WriteLine("El caníbal es temido por su brutalidad.");
                break;
        }
    }
}
```

Dentro de un `switch`, el `break` indica el final de cada caso. Sirve para que, una vez que se ejecuta el bloque correspondiente a un valor, el programa salga del `switch` y no continúe ejecutando los demás casos.

En los videojuegos, muchas veces necesitamos trabajar con distintos tipos de enemigos que comparten características generales, como **vida, daño o una descripción**, pero que varían en sus valores concretos. Para organizar mejor esta información podemos usar un **`enum`**, que nos permite definir un conjunto fijo de tipos de enemigos posibles, y luego un **`switch`** para asignarles atributos específicos según corresponda. De esta manera, el código se vuelve más **claro, legible y fácil de mantener**, evitando errores y permitiéndonos escalar el sistema a medida que agregamos más enemigos.

```
using System;

// Definimos la enumeración de enemigos
enum Enemigo { Gargola, Golem, Salvaje, Canibal }

class Program
{
    static void Main()
    {
        // Seleccionamos un enemigo
        Enemigo enemigo = Enemigo.Golem;

        int vida = 0;
        int dano = 0;

        // Según el tipo de enemigo, asignamos valores diferentes
        switch (enemigo)
        {
            case Enemigo.Gargola:
                vida = 50;
                dano = 10;
                Console.WriteLine("Aparece una Gárgola: vuela rápido y es difícil de alcanzar.");
                break;

            case Enemigo.Golem:
                vida = 120;
                dano = 20;
                Console.WriteLine("Un Gólem emerge: resistente pero lento.");
                break;

            case Enemigo.Salvaje:
                vida = 80;
                dano = 15;
                Console.WriteLine("Un Salvaje te embiste con agilidad.");
                break;

            case Enemigo.Canibal:
                vida = 100;
                dano = 25;
                Console.WriteLine("El Caníbal aparece: brutal y feroz.");
                break;
        }

        Console.WriteLine($"Vida del enemigo: {vida}");
        Console.WriteLine($"Daño del enemigo: {dano}");
    }
}
```

Entrada y Salida de datos

En C# podemos interactuar con la **consola** mediante la clase **`Console`**. La consola es una ventana de texto que permite a los programas **mostrar información al usuario** (salida) y **recibir datos del usuario** (entrada). La clase **`Console`** facilita esta interacción directamente desde nuestros programas en plataformas de desarrollo como Visual Studio, permitiendo tanto mostrar mensajes como leer información ingresada por el usuario.

- **Salida de datos:** mostrar mensajes o resultados en pantalla con `Console.WriteLine()`.

```
Console.WriteLine("Hola.");
```

- **Entrada de datos:** leer información ingresada por el usuario con `Console.ReadLine()`.

```
string input;

input = Console.ReadLine();
```

La consola, al ser una interfaz de texto simple que permite al usuario comunicarse con el programa escribiendo datos y viendo respuestas en forma de texto, es muy útil para **probar código**, **mostrar información** y **recibir entradas** de manera directa, antes de crear interfaces gráficas más complejas.

Secuencia de instrucciones

En la **programación imperativa**, las instrucciones se ejecutan **secuencialmente**, de arriba hacia abajo. En **tiempo de ejecución**, el programa procesa cada línea del código realizando todas las tareas escritas, generando la salida correspondiente.

```
1   Console.WriteLine("Hola usuario.");
2   Console.WriteLine("Contaremos hasta 5.");
3   Console.WriteLine("Uno.  (Presione una tecla para continuar)");
4   Console.ReadLine();
5   Console.WriteLine("Dos. (Presione una tecla para continuar)");
6   Console.ReadLine();
7   Console.WriteLine("Tres. (Presione una tecla para continuar)");
8   Console.ReadLine();
9   Console.WriteLine("Cuatro. (Presione una tecla para continuar)");
10  Console.ReadLine();
11  Console.WriteLine("Cinco.");
12  Console.WriteLine("Hemos terminado de contar. Presione una tecla para salir.");
13  Console.ReadLine();
```

```
Hola usuario.
Contaremos hasta 5.
Uno.  (Presione una tecla para continuar)

Dos. (Presione una tecla para continuar)

Tres. (Presione una tecla para continuar)

Cuatro. (Presione una tecla para continuar)

Cinco.
Hemos terminado de contar. Presione una tecla para salir.
```

Los **valores** son datos concretos que el procesador manipula, como **`50`** (numérico) o **`"Hola mundo"`** ("string" / texto). Las **expresiones** son fragmentos de código que representan un valor obtenido a partir de operaciones sobre valores más simples, por ejemplo **`(50/5) - 7`** da **`3`**. Existen expresiones **numéricas, lógicas o con cadenas**, y deben respetar los tipos de datos.

Los **valores literales** son constantes que no cambian durante la ejecución, como **`"Hola mundo"`**. Las **variables** pueden modificarse en tiempo de ejecución y se representan mediante un identificador que apunta a un espacio de memoria; el procesador lee el valor actual en ese espacio cuando lo necesita.

Es importante **recordar** que para poder operar con las variables, estas deben tener un **valor inicial**. El proceso de asignarles este primer valor se conoce como **inicialización**. Sin inicializar una variable, el programa no podrá utilizarla correctamente y puede generar errores durante la ejecución.

Organización del código

Instrucciones

- Son líneas de código que serán **compiladas** por el compilador.
- Cada instrucción representa una o varias **tareas** que el programa debe ejecutar.
- Están compuestas por **palabras reservadas, identificadores, sentencias y expresiones**.

```
int edad = 20; // Esta línea declara una variable y le asigna un valor
Console.WriteLine(edad); // Muestra el valor de la variable en la consola
```

Las **instrucciones** son ejecutadas por el programa.

Comentarios

- Son **anotaciones** en el código que **no se ejecutan**.
- Se utilizan para **documentar**, **explicar el funcionamiento**, o dejar **advertencias** y mensajes para otros desarrolladores.
- En C#:
    - // para comentarios de una sola línea.
    - /* … */ para comentarios de varias líneas.

```
// Este es un comentario de una sola línea

/*
  Este es un comentario
  de varias líneas
  que no se ejecuta
*/
```

Los **comentarios** solo sirven para explicar el código y no afectan su ejecución.

Reglas de nomenclatura

Variables

En C#, por convención, los identificadores de variables usan el estilo **camelCase**. La primera letra es minúscula, cada nueva palabra empieza con mayúscula y se escribe **sin espacios**, y no se permiten **acentos ni caracteres especiales** (!"#$%&). Ejemplo: `nombreJugador`

Clases

- Usan el estilo **PascalCase**.
- Cada palabra comienza con mayúscula y no se usan espacios ni caracteres especiales.
- **Ejemplo:** `Personaje`, `JuegoVideo`.

Métodos y funciones

- Usan **camelCase**, igual que las variables.
- Deben describir claramente la acción que realizan.
- **Ejemplo:** `calcularPuntaje()`, `mostrarMensaje()`.

Constantes

- Se escriben normalmente en **mayúsculas** con guiones bajos para separar palabras.
- Su valor no cambia durante la ejecución.
- **Ejemplo:** `MAX_VIDAS, PI`.

Nombres de espacios de nombres (namespaces)

- Usan **PascalCase**, similar a las clases.
- Se recomienda reflejar la estructura del proyecto o módulo.
- **Ejemplo:** `MiJuego.Logica`, `MiJuego.Entidades`.

Directivas

- Son instrucciones que controlan el **proceso de compilación** del programa.
- Se señalan con el símbolo `#`.
- Ejemplo: `#region … #endregion` se utiliza para **organizar y agrupar bloques de código** dentro del editor.

```
1  #region declaracion de variables
2
3  int numero = 10;
4  string saludo = "Hola mundo.";
5
6  #endregion
```

Lógica Booleana

Como hemos visto antes en este mismo texto, la **lógica booleana** es un sistema matemático basado en **dos valores**:

- **Verdadero (True)** → representado por 1.
- **Falso (False)** → representado por 0.

Estos valores son fundamentales en programación para representar **condiciones y decisiones**.

Operadores Relacionales

Los operadores relacionales permiten **comparar datos** y obtener un **valor de verdad (True o False)**.

Operador	Nombre	Ejemplo	Resultado
<	Menor que	1 < 3	Verdadero
<=	Menor o igual	7 <= 7	Verdadero
>	Mayor que	6 > 9	Falso
>=	Mayor o igual	9 >= 8	Verdadero
==	Igual que	8 == 8	Verdadero
!=	Distinto que	8 != 8	Falso

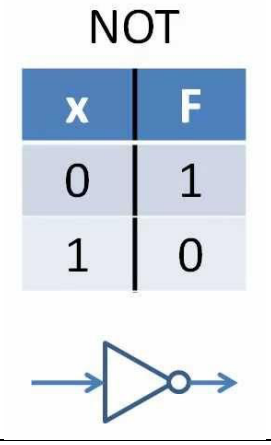
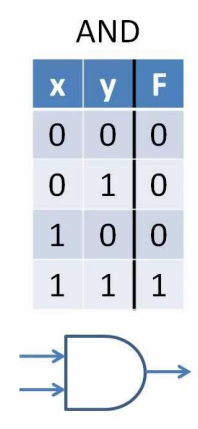
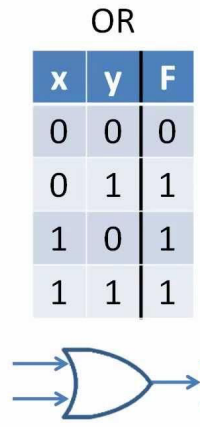
Ejemplo: determinar si es verdadero o falso:

- 14 == "Catorce" → Falso
- 345 > 983 → Falso
- 123 >= 123 → Verdadero
- "receta" == "Receta" → Falso
- "Nacional" != "Nacional" → Falso
- Verdadero == Verdadero → Verdadero

También se pueden usar operadores relacionales para **comparar variables**.

Operadores Lógicos

Permiten combinar resultados de comparaciones usando **AND (Y)**, **OR (O)** y **NOT (Negación)**.

NOT (!)	AND (&)	OR (||)
Invierte el valor lógico de su entrada. Ejemplo en C#: `!(8 == 8)` Resolver la comparación: `8 == 8` → Verdadero Aplicar NOT: !Verdadero → Falso	Requiere **dos valores de entrada**. Solo devuelve **Verdadero** si **ambos** son Verdadero; si al menos uno es Falso, devuelve Falso. Ejemplo en C#: `(7 < 8) & (15 >= 12)` → Verdadero	Requiere **dos valores de entrada**. Devuelve **Verdadero** si al menos una entrada es Verdadero; solo devuelve Falso si todas son Falso. Ejemplo en C#: `("Facu" == "Facundito") || (13 < 1)` → Falso
		

Por otro lado, también existe el operador **XOR (O exclusivo).** Es otro operador lógico fundamental que devuelve **Verdadero solo cuando una de sus entradas es Verdadero, pero no ambas**; si ambas son iguales (Verdadero/Verdadero o Falso/Falso), el resultado será Falso. En C# se representa con el símbolo `^`. Por ejemplo, `true ^ false` devuelve Verdadero, mientras que `true ^ true` devuelve Falso. Este operador se utiliza frecuentemente en programación para alternar estados, controlar condiciones que no pueden cumplirse al mismo tiempo o en operaciones más avanzadas como criptografía y manipulación de bits.

La diferencia entre **OR (||)** y **XOR (^)** radica en cómo se comportan cuando ambas condiciones son verdaderas: el operador OR devuelve verdadero siempre que al menos una de las entradas lo sea, incluso si las dos son verdaderas; en cambio, el operador XOR devuelve verdadero únicamente cuando una de las dos condiciones es verdadera y la otra es falsa, devolviendo falso cuando ambas coinciden (ya sea verdadero/verdadero o falso/falso).

Múltiples operadores

La lógica booleana nos permite hacer operaciones múltiples donde la salida de un operador sea la entrada de otro. Esto permite construir sistemas lógicos complejos.

Ejemplo:

```
(7 < 8 || 15 >= 12) && (!(25 == 25))

⇒ (Verdadero || Verdadero) && (!Verdadero)

⇒ Verdadero && Falso

⇒ Falso
```

Estructuras de control condicional

O simplemente **Condicionales**, son estructuras que nos permiten ejecutar una serie de instrucciones si una condición se cumple. Podemos trabajar con condicionales de distintas maneras

Condicional simple

Ejecuta un bloque de instrucciones solo si una expresión booleana es verdadera.

```
if (condición)
{
    // Código a ejecutar si la condición es verdadera
}
```

Ejemplo en C#: Si la variable `edad` es mayor o igual a 18, mostrar en consola `"Sos mayor de edad".`

```
int edad = 20; // Variable con un valor de ejemplo

// Condicional simple: verifica si la edad es mayor o igual a 18

if (edad >= 18)

{

    Console.WriteLine("Sos mayor de edad");

}
```

La instrucción `Console.WriteLine("Sos mayor de edad");` le indica al programa que muestre ese mensaje en la consola. La consola una ventana de texto muy simple donde el programa "habla" con la persona que lo está usando. Al ejecutarse, la frase aparece escrita en pantalla, como si el programa estuviera diciendo directamente ese mensaje al usuario.

Bicondicional (if/else)

Nos permite alternar entre dos bloques de código según el valor de una expresión booleana:

- Si la condición es verdadera, se ejecuta el primer bloque.
- Si es falsa, se ejecuta el segundo.

```
if (condición)
{
    // Código a ejecutar si la condición es verdadera
}
else
{
    // Código a ejecutar si la condición es falsa
}
```

Ejemplo en C#: Si *edad >= 18* mostrar "Sos mayor de edad", de lo contrario mostrar "Sos menor de edad")

```
using System;

class Program
{
    static void Main()
    {
        // Declaramos una variable llamada "edad" y le damos un valor inicial
        int edad = 17;

        // Condicional: verificamos si la edad es mayor o igual a 18
        if (edad >= 18)
        {
            // Si la condición se cumple, mostramos este mensaje en la consola
            Console.WriteLine("Sos mayor de edad");
        }
        else
        {
            // Si la condición NO se cumple, mostramos este otro mensaje
            Console.WriteLine("Sos menor de edad");
        }
    }
}
```

Anidación

Las estructuras condicionales pueden **anidarse**, es decir, colocar una dentro de otra para ir afinando las decisiones y manejar casos más específicos en cada nivel. A continuación, veamos un ejemplo en C#:

```
using System;

class Program
{
    static void Main()
    {
        int edad = 20;
        bool tieneLicencia = true;

        // Condicional principal: verificamos si la persona es mayor o igual a 18
        if (edad >= 18)
        {
            // Condicional anidado: si además tiene licencia, mostramos un mensaje específico
            if (tieneLicencia)
            {
                Console.WriteLine("Podés conducir legalmente");
            }
            else
            {
                Console.WriteLine("Sos mayor de edad pero no tenés licencia");
            }
        }
        else
        {
            // Si no es mayor de edad
            Console.WriteLine("Sos menor de edad");
        }
    }
}
```

Este ejemplo muestra cómo se puede **poner un condicional dentro de otro** para tomar decisiones más específicas. Primero, el programa verifica si la persona es mayor o igual a 18 años; si es así, entra al condicional interno para ver si además tiene licencia y muestra un mensaje distinto según corresponda. Si la persona no es mayor de edad, se ejecuta directamente el bloque del `else` exterior. La anidación permite manejar varias condiciones de forma ordenada y precisa.

En C#, la línea **`using System;`** le indica al programa que puede **utilizar funcionalidades que están definidas en el espacio de nombres System**, como la consola, operaciones matemáticas, manejo de cadenas y otras herramientas básicas que el lenguaje ya ofrece. En palabras simples, **`using System;`** le dice al programa: **"quiero poder usar las herramientas básicas que ya vienen con C#"**, como mostrar mensajes en pantalla, hacer cálculos, trabajar con texto o números, sin tener que escribir todo desde cero. Es como si le dijéramos al programa: "traé la caja de herramientas estándar para poder usarla".

Por otro lado, la línea **`class Program`** declara una **clase llamada Program**, que es como un contenedor o plantilla donde se agrupa todo el código del programa; dentro de esta clase se definen los métodos y funciones que realizará el programa. En conjunto, estas líneas preparan el entorno para escribir y organizar las instrucciones que se ejecutarán.

Operadores de control de flujo: Bucles

Como vimos al principio de este texto, en programación, un **bucle** es una estructura que permite ejecutar un bloque de código repetidamente mientras se cumpla una condición específica. Los bucles son fundamentales para **automatizar tareas repetitivas** y procesar grandes cantidades de datos de manera eficiente.

Todos los bucles tienen una **condición booleana** que determina si deben continuar ejecutándose o detenerse. Dentro del bucle, es importante realizar operaciones que **modifiquen esta condición**, para evitar caer en un **bucle infinito**, que haría que el programa nunca termine.

En un principio veremos tres tipos principales de bucles condicionales:

1. **While**
2. **Do-While**
3. **For**

Iteración: While (Mientras)

El bucle **`while`** ejecuta de manera repetida las instrucciones que se encuentren dentro de él **mientras la condición sea verdadera**. Cuando la condición se vuelve falsa, la iteración se detiene.

```
int a = 0;
while (a < 10) // mientras a sea menor que 10
{
    Console.WriteLine(a); // se imprime el valor de "a"
    a++; // se incrementa a en 1
}
```

En este caso, `a` comienza en 0 y se imprime en pantalla hasta que llegue a 9. Cuando `a` es 10, la condición deja de cumplirse y el bucle termina.

Iteración: Do-While (Hacer-Mientras)

El bucle **`do-while`** funciona de manera similar al **`while`**, pero con una diferencia importante: **la condición se evalúa después de ejecutar las instrucciones**, por lo que el contenido del bucle **siempre se ejecuta al menos una vez**, incluso si la condición inicial es falsa.

```
int a = 0;
do
{
    Console.WriteLine(a); // se imprime el valor de "a"
    a++; // se incrementa a en 1
} while (a < 10); // se repite mientras "a" sea menor que 10
```

En este caso, el bucle asegura que el bloque de instrucciones se ejecute al menos una vez antes de verificar la condición.

Iteración: For

El bucle **`for`** se utiliza cuando conocemos **de antemano el número de iteraciones**. Tiene un contador que se inicializa, una condición de ejecución y un incremento/decremento que se aplica en cada vuelta.

```
for (int i = 0; i < 10; i++)
{
    Console.WriteLine(i); // se imprime el valor de i
}
```

En este ejemplo:

- **`int i = 0`** → inicializa el contador.
- **`i < 10`** → condición de continuación del bucle.
- **`i++`** → incrementa el contador en cada iteración.

El bucle **`for`** es especialmente útil cuando queremos **recorrer listas, arrays o realizar un número determinado de repeticiones**.

En C#, **`i++`** es un **operador de incremento** que aumenta el valor de la variable **`i`** en **`1`**. Es simplemente una forma abreviada de escribir **`i = i + 1`**. Las letras como **`i`** o **`a`** son simplemente **nombres de variables de control** en un bucle; cumplen la misma función y la diferencia está solo en cómo las llamemos (por eso, también podríamos poner `a++`).

¿Qué son los métodos y funciones?

En programación, los **métodos** y **funciones** son bloques de código que realizan una tarea específica y que podemos **usar varias veces** sin tener que escribir el mismo código cada vez.

- Un **método** es una función que generalmente pertenece a una **clase** y puede operar sobre los datos de esa clase. Por ejemplo, en un juego, un método llamado `MoverJugador()` podría cambiar la posición del jugador en la pantalla.
- Una **función** es un concepto más general: un bloque de código que recibe información, hace algo con ella y puede devolver un resultado. Por ejemplo, una función `CalcularPuntaje()` podría tomar los puntos de un nivel y devolver el puntaje total.

En ambos casos, el objetivo es **organizar el código, reutilizarlo y facilitar su lectura**, evitando repetir instrucciones una y otra vez.

```
using System;

class Program
{
    // Este es un método que suma dos números y devuelve el resultado
    static int Sumar(int a, int b)
    {
        return a + b; // Devuelve la suma de a y b
    }

    static void Main()
    {
        int resultado = Sumar(5, 3); // Llamamos al método Sumar con 5 y 3
        Console.WriteLine("El resultado es: " + resultado); // Mostramos el resultado en pantalla
    }
}
```

En C#, `static int` combina dos conceptos: `int` indica que el método va a **devolver un número entero**, mientras que `static` significa que el método **pertenece a la clase y no a un objeto específico** de esa clase. Esto permite llamar al método directamente usando el nombre de la clase, sin necesidad de crear una instancia u objeto de esa clase. Por ejemplo, un método declarado como `static int Sumar(int a, int b)` puede ser usado inmediatamente desde `Main` para obtener la suma de dos números.

Este ejemplo ilustra cómo los **métodos/funciones permiten organizar, reutilizar y simplificar el código**, separando tareas específicas (como sumar dos números) del flujo principal del programa:

`static int Sumar(int a, int b)`: es un método que recibe dos números enteros (**`a`** y **`b`**) y devuelve su suma. La palabra **`int`** indica que el resultado será un número entero, y en C#, cuando un método es **`static`**, significa que **pertenece a la clase en sí y no a ninguna instancia u objeto** de esa clase, por lo que se puede llamar directamente usando el nombre de la clase, sin tener que crear un objeto. Este método encapsula la lógica de sumar dos números, permitiendo reutilizarla cada vez que queramos sumar valores, sin repetir el código.	**`static void Main()`**: es el **método principal del programa**, es decir, el punto de entrada donde empieza a ejecutarse todo el código. La palabra **`static`** indica que pertenece a la clase y puede ejecutarse sin crear un objeto, y **`void`** significa que este método **no devuelve ningún valor**. Dentro de **`Main`** es donde se pueden llamar otros métodos, como **`Sumar(5, 3)`**, y almacenar su resultado en una variable (resultado). Luego, con **`Console.WriteLine`** se muestra ese valor en la pantalla, permitiendo que el usuario vea el resultado de la operación. En pocas palabras, **`Main`** es **el lugar donde el programa arranca y coordina la ejecución de todas las tareas**.

En muchos lenguajes, **método** y **función** son muy similares: ambos son bloques de código que hacen algo específico, pero la diferencia principal es que **un método pertenece a una clase u objeto**, mientras que una **función puede existir de forma independiente**. En C# todo se define dentro de clases, así que las funciones "independientes" se simulan como métodos estáticos.

```
using System;

class Calculadora
{
    // Método: pertenece a la clase Calculadora, actúa sobre objetos
    public int Sumar(int a, int b)
    {
        return a + b;
    }

    // Función "independiente": static, se puede llamar sin crear un objeto
    public static int Multiplicar(int a, int b)
    {
        return a * b;
    }
}

class Program
{
    static void Main()
    {
        // Crear un objeto de la clase Calculadora para usar el método
        Calculadora calc = new Calculadora();
        int suma = calc.Sumar(5, 3); // Llamada a un método de instancia
        Console.WriteLine("Suma: " + suma);

        // Llamada a la "función" estática sin crear un objeto
        int producto = Calculadora.Multiplicar(5, 3);
        Console.WriteLine("Producto: " + producto);
    }
}
```

`Sumar` es un **método de instancia**: pertenece al objeto **`calc`**, por eso primero hay que crear una instancia de **`Calculadora`** para usarlo. **`Multiplicar`** es una **función estática**: no depende de ningún objeto, se llama directamente desde la clase.

La línea "**Calculadora calc = new Calculadora();**" hace varias cosas:

- **`Calculadora calc`**: declara una variable llamada **`calc`** que podrá **almacenar un objeto** de tipo **`Calculadora`**.
- **`new Calculadora()`**: crea un **nuevo objeto** de la clase **`Calculadora`** en memoria. La palabra **`new`** significa literalmente "crear una nueva instancia" de esa clase.
- **`=`**: asigna la nueva instancia creada a la variable **`calc`,** para que podamos usar ese objeto más adelante.

Una **instancia** es un objeto concreto creado a partir de una clase, que actúa como un molde o plano que define sus propiedades y métodos. Cada instancia ocupa un espacio en la memoria y puede tener valores específicos para sus propiedades, permitiendo que varios objetos coexistentes basados en la misma clase funcionen de manera independiente.

Es decir, en resumen, esta línea **reserva memoria para un nuevo objeto de la clase `Calculadora`, lo crea y lo guarda en la variable `calc`**, permitiéndonos luego llamar a sus métodos (como **`Sumar`**) usando ese objeto.

En otras palabras:

- **método → pertenece a un objeto**
- **función → puede ser independiente**

Rutinas

Una **rutina** es una secuencia de instrucciones que realiza una tarea específica. Son bloques de código organizados y reutilizables que se pueden invocar desde distintos puntos del programa. Podemos pensar en ellas como un "código secundario" que se declara y define fuera del bloque principal de ejecución. Para que una rutina se ejecute, debe ser **invocada** como una instrucción más dentro del flujo del programa.

Tipos de Rutinas

Métodos	Procedimientos	Funciones
Son rutinas que **pertenecen a clases u objetos** y se utilizan para encapsular instrucciones que realizan una operación concreta.	Los procedimientos son rutinas **sin valor de retorno**. Ejecutan acciones, pero no devuelven ningún dato.	Las funciones son rutinas que **devuelven un valor**. Pueden devolver distintos tipos de datos, como enteros, cadenas, objetos, etc. Trabajan igual que los procedimientos, pero en su firma y cuerpo se indica qué tipo de dato retornan, y se usa la palabra reservada `return` para devolverlo.

Declaración, definición e invocación

Declaración: Es la línea de código que indica que la rutina existe. Representa su **firma**, indicando el tipo de retorno, el identificador y los parámetros.

```
void MiPrimerRutina()
```

`void` significa que no devuelve ningún valor, convirtiéndola en un procedimiento.

Definición: Es el **cuerpo de la rutina**, donde se escriben todas las instrucciones que se ejecutarán cuando la rutina sea llamada.

```
void MiPrimerRutina()
{
    Console.WriteLine("Hola mundo");
}
```

Invocación: Es el llamado a la rutina desde distintos contextos. Para ejecutarla, usamos su identificador como una instrucción:

```
MiPrimerRutina();
```

En C#, las rutinas se declaran por fuera del código principal y están **atadas al ámbito** donde se definen.

Parametrización

Las rutinas pueden recibir **datos desde otro contexto** al ser invocadas, usando **parámetros** declarados en su firma:

```
void MostrarMensaje(string mensaje)
{
    Console.WriteLine(mensaje);
}
```

- Al invocar la rutina debemos pasar datos del mismo tipo que los parámetros:
- `MostrarMensaje("Hola a todos");`

Formas de pasar parámetros

- **Por valor:** La rutina recibe una copia del dato. Los cambios no afectan la variable original.
- **Por referencia:** La rutina accede al **mismo espacio de memoria** del parámetro, por lo que cualquier cambio se refleja en la variable original.

Funciones con retorno

Si una rutina devuelve un valor, se conoce como **función**. La firma debe indicar el tipo de dato que devuelve y el bloque debe garantizar que **todos los caminos de ejecución retornen un valor** usando `return`:

```
string PedirNombre()
{
    string nombre = "Facu";
    return nombre; // Devuelve el valor de la función
}
```

Invocación de funciones

Al invocar una función debemos **usar el valor de retorno** de alguna manera: asignándolo a una variable, imprimiéndolo o pasándolo como argumento a otra rutina:

```
string miNombre = PedirNombre();
Console.WriteLine(miNombre);
```

Registros

Los **registros** (`record`) son una estructura especial que permite representar objetos con **valores inmutables**, es decir, que una vez creados no pueden modificarse. Son útiles cuando queremos modelar entidades donde importan más los datos que la identidad del objeto.

```
// Definimos un registro para representar un Jugador
public record Jugador(string Nombre, int Nivel, int Puntos);

class Program
{
    static void Main()
    {
        // Creamos una instancia de Jugador
        Jugador jugador1 = new Jugador("Facu", 5, 1200);

        Console.WriteLine($"Jugador: {jugador1.Nombre}, Nivel:
{jugador1.Nivel}, Puntos: {jugador1.Puntos}");
    }
}
```

En este ejemplo, el registro `Jugador` almacena un conjunto fijo de valores (`Nombre`, `Nivel`, `Puntos`) que representan el estado del jugador.

Secuencias – Uso del ForEach

Una **secuencia** es una estructura lineal que organiza elementos en un **orden definido**, donde cada valor ocupa una **posición (índice)**.

- **Acceso posicional:** se puede acceder o modificar un elemento indicando su índice.
- **Longitud:** puede ser fija (arrays) o variable (listas).
- **Modificabilidad:** pueden ser mutables o inmutables según cómo se definan.
- **Condiciones de borde:** hay que evitar acceder a posiciones inexistentes.

La instrucción **`foreach`** permite recorrer de forma segura todos los elementos de una secuencia. Veamos un ejemplo en Javascript.

```
let frutas = ["Manzana", "Banana", "Cereza"];



frutas.forEach(function(fruta){

    console.log(fruta);

});
```

A continuación, veamos un ejemplo en C#:

```
using System;
using System.Collections.Generic;

class Program
{
    // Asignar valores aleatorios a un array
    static void RellenarArray(int[] arr)
    {
        Random rnd = new Random();
        for (int i = 0; i < arr.Length; i++)
            arr[i] = rnd.Next(1, 100); // valores entre 1 y 99
    }

    // Imprimir array con foreach
    static void ImprimirArray(int[] arr)
    {
        foreach (int num in arr)
            Console.Write(num + " ");
        Console.WriteLine();
    }

    // Agregar un número a una secuencia (Lista)
    static void Agregar(List<int> lista, int num)
    {
        lista.Add(num);
    }

    // Quitar todas las ocurrencias de un número en la secuencia
    static void Quitar(List<int> lista, int num)
    {
        lista.RemoveAll(x => x == num);
    }

    static void Main()
    {
        int[] numeros = new int[20];
        RellenarArray(numeros);
        Console.WriteLine("Array inicial:");
        ImprimirArray(numeros);

        List<int> lista = new List<int>(numeros);
        Agregar(lista, 50);
        Quitar(lista, 10);

        Console.WriteLine("\nLista después de operaciones:");
        foreach (int n in lista) Console.Write(n + " ");
    }
}
```

Pilas – FILO (First In, Last Out)

Una **pila** trabaja con el **último elemento ingresado**:

- **Push:** agrega al final.
- **Pop:** quita el último.

Ejemplo en C# – Pila

```
using System;
using System.Collections.Generic;

class ProgramaPila
{
    static void Main()
    {
        Stack<int> pila = new Stack<int>();

        // Push (agregar)
        pila.Push(10);
        pila.Push(20);
        pila.Push(30);

        // Pop (quitar)
        Console.WriteLine("Elemento quitado: " + pila.Pop());

        Console.WriteLine("Elementos restantes en la pila:");
        foreach (int num in pila) Console.WriteLine(num);
    }
}
```

Colas – FIFO (First In, First Out)

Una **cola** trabaja con el **primer elemento ingresado**:

- **Push (Enqueue):** agrega al final.
- **Pop (Dequeue):** quita desde el principio.

Ejemplo en C# – Cola

```
using System;
using System.Collections.Generic;

class ProgramaCola
{
    static void Main()
    {
        Queue<int> cola = new Queue<int>();

        // Push (agregar)
        cola.Enqueue(10);
        cola.Enqueue(20);
        cola.Enqueue(30);

        // Pop (quitar)
        Console.WriteLine("Elemento quitado: " + cola.Dequeue());

        Console.WriteLine("Elementos restantes en la cola:");
        foreach (int num in cola) Console.WriteLine(num);
    }
}
```

Las **Pilas (F.I.L.O.)** y las **Colas (F.I.F.O.)** son estructuras que nos sirven para manejar datos de manera ordenada según cómo los agregamos y quitamos. Una **Pila** funciona como una pila de cartas: el último que entra es el primero que sale (*First In, Last Out*). Esto es útil en videojuegos, por ejemplo, para manejar acciones que deben deshacerse en orden inverso (un sistema de "deshacer movimientos" o historial de estados). En cambio, una **Cola** funciona como una fila de personas: el primero que entra es el primero que sale (*First In, First Out*). Esto se aplica en videojuegos cuando necesitamos procesar eventos o turnos en el orden en que ocurrieron, como una cola de ataques en un combate por turnos o una lista de enemigos que aparecen en secuencia. En resumen: la Pila sirve para trabajar con lo más reciente, y la Cola para respetar el orden de llegada.

```
using System;
using System.Collections.Generic;

class Program
{
    static void Main()
    {
        // Ejemplo de Pila (FILO): historial de movimientos del jugador
        Stack<string> historialMovimientos = new Stack<string>();
        historialMovimientos.Push("Mover Arriba");
        historialMovimientos.Push("Mover Derecha");
        historialMovimientos.Push("Saltar");

        Console.WriteLine("Deshaciendo movimientos:");
        while (historialMovimientos.Count > 0)
        {
            string movimiento = historialMovimientos.Pop(); // Se quita el último movimiento
            Console.WriteLine(movimiento);
        }

        Console.WriteLine();

        // Ejemplo de Cola (FIFO): turnos de enemigos en un combate
        Queue<string> turnosEnemigos = new Queue<string>();
        turnosEnemigos.Enqueue("Goblin");
        turnosEnemigos.Enqueue("Orco");
        turnosEnemigos.Enqueue("Troll");

        Console.WriteLine("Turnos de enemigos:");
        while (turnosEnemigos.Count > 0)
        {
            string enemigo = turnosEnemigos.Dequeue(); // Se procesa el primer enemigo que entró
            Console.WriteLine(enemigo + " ataca!");
        }
    }
}
```

- La Pila (Stack) guarda los movimientos del jugador y los deshace en orden inverso (último en entrar, primero en salir).

- La Cola (Queue) guarda el orden de ataque de los enemigos, asegurando que el primero en entrar sea el primero en actuar.

Nociones a tener presente al programar

Principio del tercero excluido

En programación y lógica, el **principio del tercero excluido** establece que toda proposición es verdadera o falsa, sin posibilidad de un estado intermedio. Esto es fundamental al diseñar algoritmos y estructuras de control, ya que permite tomar decisiones precisas mediante condicionales.

Noción de azar en programación

La **noción de azar** en programación difiere del azar del mundo real. Los generadores de números aleatorios producen secuencias **determinísticas** que imitan la aleatoriedad, conocidas como números pseudoaleatorios, y pueden controlarse mediante semillas. Por lo tanto, en programación, no existe un azar "verdadero".

Parametrización de procedimientos

La **parametrización** permite diseñar funciones o métodos más flexibles y reutilizables. Esto se logra mediante:

- **Argumentos**: valores que se pasan al llamar a un procedimiento.
- **Parámetros formales**: variables definidas dentro del procedimiento que reciben los argumentos.

El intercambio de datos puede realizarse:

- **Por valor**: el procedimiento recibe una copia de la información.
- **Por referencia**: el procedimiento trabaja directamente sobre la variable original, permitiendo modificar su contenido.

Algoritmos para estructuras de datos

Las **estructuras de datos lineales y bidimensionales** son esenciales para organizar información:

- **Vectores (arrays unidimensionales)**
- **Matrices 2D (arrays bidimensionales)**

Sobre estas estructuras se aplican operaciones clave:

- **Poblado**: llenar los elementos con datos.
- **Mapeo**: aplicar una función o transformación a cada elemento.
- **Búsqueda**: localizar un valor o conjunto de valores.
- **Visualización**: mostrar los contenidos de manera legible.

El uso adecuado de algoritmos sobre estas estructuras permite resolver problemas de manera eficiente y sistemática.

Programa integrador (con todo lo visto)

Este programa combina **credenciales, enumeraciones, arrays, condicionales, bucles y control de acceso**:

```
using System;

enum TipoUsuario
{
    Administrador,
    Invitado
}

class Program
{
    static void Main()
    {
        // Credenciales guardadas
        string adminUser = "admin";
        string adminPass = "1234";

        // Array de 10 números enteros
        int[] numeros = new int[10] { 1,2,3,4,5,6,7,8,9,10 };

        // Pedimos credenciales
        Console.WriteLine("Ingrese su nombre de usuario:");
        string usuario = Console.ReadLine();

        Console.WriteLine("Ingrese su contraseña:");
        string pass = Console.ReadLine();

        // Determinar tipo de usuario
        TipoUsuario tipo;

        if (usuario == adminUser && pass == adminPass)
        {
            tipo = TipoUsuario.Administrador;
            Console.WriteLine("Bienvenido Administrador.");
        }
        else
        {
            tipo = TipoUsuario.Invitado;
            Console.WriteLine("Bienvenido Invitado.");
        }

        // Menú según tipo de usuario
        bool salir = false;
        while (!salir)
        {
            Console.WriteLine("\nOpciones:");
            Console.WriteLine("1. Ver array");
            if (tipo == TipoUsuario.Administrador)
            {
                Console.WriteLine("2. Modificar array");
            }
            Console.WriteLine("0. Salir");

            string opcion = Console.ReadLine();

            switch (opcion)
            {
                case "1":
                    Console.WriteLine("Contenido del array:");
                    for (int i = 0; i < numeros.Length; i++)
                    {
                        Console.Write(numeros[i] + " ");
                    }
                    Console.WriteLine();
```

```
                    break;

                case "2":
                    if (tipo == TipoUsuario.Administrador)
                    {
                        Console.WriteLine("Ingrese la posición (0 a 9) a modificar:");
                        int pos = int.Parse(Console.ReadLine());

                        if (pos >= 0 && pos < numeros.Length)
                        {
                            Console.WriteLine("Ingrese el nuevo valor:");
                            int nuevoValor = int.Parse(Console.ReadLine());
                            numeros[pos] = nuevoValor;
                            Console.WriteLine("Valor modificado correctamente.");
                        }
                        else
                        {
                            Console.WriteLine("Posición inválida.");
                        }
                    }
                    else
                    {
                        Console.WriteLine("No tiene permisos para esta acción.");
                    }
                    break;

                case "0":
                    salir = true;
                    break;

                default:
                    Console.WriteLine("Opción no válida.");
                    break;
            }
        }

        Console.WriteLine("Programa finalizado.");
    }
}
```

En este programa tenemos:

- **Credenciales de usuario** con validación.
- **Enumeración (`enum`)** para diferenciar `Administrador` e `Invitado`.
- **Array** de 10 enteros accesible a ambos usuarios.
- **Permisos distintos**: los invitados solo ven el array, los administradores también lo pueden modificar.
- Uso de **bucles, condicionales, switch y entrada/salida por consola**.

Este código está escrito en **C#** y define un programa de consola que simula un sistema con **usuarios y permisos**. En primer lugar, se declara un **enumerado (`enum`) llamado `TipoUsuario`**, que permite diferenciar entre dos tipos de usuarios: **Administrador** e **Invitado**. Esto sirve para controlar qué acciones puede realizar cada usuario dentro del programa.

A continuación, dentro de la clase `Program` y el método `Main`, se definen las **credenciales del administrador** (`adminUser` y `adminPass`) y un **array de 10 números enteros** llamado `numeros`. Este array se utilizará como recurso que el usuario puede ver o, en el caso del administrador, modificar.

El programa luego solicita al usuario que ingrese su **nombre de usuario** y **contraseña** mediante la consola (`Console.ReadLine()`). Con estos datos, se determina el **tipo de usuario**: si las credenciales coinciden con las del administrador, se le asigna el tipo `Administrador`; en caso contrario, se le asigna `Invitado`. Según esto, se muestra un mensaje de bienvenida correspondiente.

Después, el programa entra en un **bucle `while`** que representa un **menú interactivo**. Mientras la variable `salir` sea `false`, se muestran las opciones disponibles: **ver el array**, **modificar el array** (solo si es administrador) y **salir** del programa. La opción seleccionada se lee desde la consola y se procesa mediante un **`switch`**.

Si el usuario selecciona **ver el array (opción 1)**, se recorre el array `numeros` con un **bucle `for`** y se imprime cada valor en la consola. Esto permite al usuario visualizar el contenido del array en cualquier momento.

Si el usuario es un **Administrador** y selecciona **modificar el array (opción 2)**, el programa le solicita la **posición del elemento a modificar** y el **nuevo valor**. Tras la validación de que la posición esté dentro del rango válido (0 a 9), se reemplaza el valor correspondiente en el array y se confirma la modificación. Si un Invitado intenta usar esta opción, se le muestra un mensaje indicando que **no tiene permisos**.

Finalmente, si el usuario selecciona la opción **0**, la variable `salir` se pone en true, saliendo del bucle y terminando el programa. Cualquier otra entrada que no corresponda a una opción válida genera un mensaje de error. Al finalizar el bucle, se imprime `"Programa finalizado."`, indicando que la ejecución ha terminado.

¿Dónde puedo programar y probar mis códigos en C#?

Para programar y probar códigos en **C#** existen varias opciones según tus necesidades y preferencias. Muchos desarrolladores utilizan **Visual Studio**, un entorno de desarrollo integrado (IDE) muy completo que permite escribir, depurar y compilar programas de manera profesional. También se pueden usar editores más simples como **Brackets**, que aunque están más orientados a desarrollo web, pueden servir para escribir código y luego compilarlo con herramientas externas. Por otra parte, hay opciones **online** que no requieren instalar nada en la computadora, lo que resulta muy práctico para aprender o probar fragmentos de código rápidamente; un ejemplo es OnlineGDB (https://www.onlinegdb.com/online_csharp_compiler), donde podés escribir, ejecutar y depurar tus programas de C# directamente desde el navegador.

¿Se pueden desarrollar videojuegos sin saber programar?

Existen varias herramientas que permiten **crear videojuegos utilizando programación visual**, es decir, arrastrando y conectando bloques de acciones en lugar de escribir código tradicional. Este enfoque es ideal para principiantes o para quienes quieren enfocarse en **diseño, narrativa o mecánicas de juego** sin preocuparse por la sintaxis de un lenguaje de programación. Entre estas herramientas destacan:

- **Scratch**: una plataforma muy popular para aprender lógica de programación y crear juegos sencillos mediante bloques de código visual, que fue desarrollada por el MIT.
- **Construct**: permite diseñar juegos en 2D con un sistema de eventos visual, muy intuitivo y potente para prototipos rápidos.
- **GDevelop**: una alternativa open source que facilita la creación de juegos 2D mediante eventos y lógica visual, sin necesidad de programar.
- **RPG Maker**: ideal para crear **juegos de rol (RPG)** con mecánicas predefinidas, escenarios y personajes, usando principalmente interfaces gráficas.

Algunas de estas herramientas permiten **incorporar fragmentos de código**, pero su objetivo principal sigue siendo la programación visual, por lo que no es necesario ser programador para crear juegos funcionales y completos.

Ahora bien, más allá de que utilicemos **programación visual** o escribamos líneas de código directamente, en todos los casos necesitamos contar con una **base sólida de programación**. Más allá de manipular elementos visuales, es esencial entender qué ocurre realmente en el programa. El simple hecho de arrastrar bloques o pulsar botones no garantiza resultados; debemos comprender cómo funciona la lógica del juego. Esto incluye conceptos como **funciones**, **variables**, **constantes**, **ciclos**, **condicionales**, **arrays** y **listas**, entre otros. Tener claros estos fundamentos nos permite **planificar la lógica del juego**, controlar cómo se comportan los personajes, cómo se registran los puntajes, cómo se activan eventos o cómo se gestionan los recursos, etc. En otras palabras, aunque algunas herramientas simplifiquen la escritura de código, el **pensamiento lógico y la comprensión de la programación** siguen siendo esenciales para poder desarrollar un videojuego que funcione correctamente y cumpla con nuestra visión creativa.

Si se quiere desarrollar videojuegos más complejos, con mecánicas avanzadas, gráficos 3D o física realista, se utilizan **motores de videojuegos profesionales**, que requieren conocimientos más avanzados de programación:

- **Unreal Engine**: potente motor 3D, ideal para juegos AAA y experiencias inmersivas.
- **Unity**: ampliamente usado tanto para juegos 2D como 3D, con soporte para C# y una gran comunidad que desarrolla sus videojuegos utilizando este motor.
- **Godot**: open source, flexible y ligero, permite programar en su lenguaje GDScript, C# o C++.

Las plataformas con programación visual son perfectas para iniciarse, mientras que motores profesionales permiten explotar al máximo la creatividad mediante código escrito. El nivel de complejidad y personalización va a depender de la herramienta utilizada.

Fundamentos de la Programación Web

La **programación web** es el conjunto de técnicas y lenguajes que se utilizan para crear sitios y aplicaciones que funcionan en navegadores de internet. Para construir una página web necesitamos tres pilares fundamentales: **HTML, CSS y JavaScript**.

HTML (HyperText Markup Language)

El **HTML** es el lenguaje de marcado que sirve para estructurar el contenido de una página web. Con HTML podemos definir elementos como títulos, párrafos, listas, imágenes, enlaces, botones y formularios. En otras palabras, HTML es el **"esqueleto"** de nuestra página, y todo lo demás se construye sobre él.

CSS (Cascading Style Sheets)

El **CSS** se utiliza para darle **estilo** a nuestra página web. Mientras que HTML define la estructura, CSS define la apariencia: colores, tamaños, tipografías, márgenes, bordes, animaciones y posiciones de los elementos. Gracias a CSS podemos transformar un contenido simple en una interfaz atractiva y fácil de usar.

JavaScript

El **JavaScript** es un lenguaje de programación que permite agregar **interactividad** y dinamismo a nuestras páginas web. Con JavaScript podemos responder a eventos como clicks de botones, movimientos del mouse, validación de formularios, animaciones, juegos y hasta conectar nuestra web con bases de datos o servicios externos. En pocas palabras, JavaScript convierte nuestra página web de estática a interactiva.

Herramientas de Desarrollo Web

Para programar y publicar una página web necesitamos algunas herramientas que faciliten nuestro trabajo:

Brackets

Brackets (https://brackets.io/)es un **editor de código gratuito y open source** que permite escribir HTML, CSS y JavaScript de manera eficiente. Entre sus ventajas se destacan: resaltado de sintaxis, autocompletado de código y vista previa en vivo de la página web. Personalmente este es el que yo uso, aunque vale aclarar que cualquier otro editor o IDE como Visual Studio Code, Sublime Text o Atom también podría servirnos.

FileZilla

FileZilla (https://filezilla-project.org/)es un **cliente FTP gratuito** que permite subir los archivos de nuestro sitio web a un servidor para que estén disponibles en internet. Con FileZilla podemos transferir archivos desde nuestra computadora al hosting de forma sencilla y segura.

GitHub

GitHub (https://github.com/)es una **plataforma de control de versiones y alojamiento de proyectos** que permite guardar, organizar y compartir nuestro código. Si no se tiene un sitio web propio, GitHub Pages permite publicar páginas web directamente desde un repositorio, lo que resulta ideal para proyectos personales o trabajos académicos.

¿Qué es un Sitio Web?

Es importante diferenciar entre **sitio web** y **página web**. Una **página web** es un solamente documento accesible desde internet, como una hoja individual que contiene contenido específico: texto, imágenes, videos o enlaces. Por ejemplo, la página de contacto de una empresa o una entrada de blog. En cambio, un **sitio web** es el conjunto completo de páginas web relacionadas que conforman una presencia en línea, incluyendo la página de inicio, secciones de productos, blogs, contacto, y cualquier otra página que forme parte del mismo dominio.

¿Cómo tener un sitio web en línea?

Para entender cómo hacer que nuestro sitio web esté disponible en internet, necesitamos comprender algunos conceptos básicos:

- **Servidor**: es la computadora que almacena los archivos de tu sitio web y los entrega a los visitantes cuando acceden desde un navegador.
- **Hosting**: es el servicio que te permite **alojar tu sitio web en un servidor** para que esté disponible las 24 horas del día.
- **Dominio**: es la dirección única de tu sitio web en internet, como www.misitio.com, que los usuarios escriben en el navegador para acceder a tu sitio.

Actualmente, personalmente he utilizado servicios como **Bluehost** y **Hostinger** para alojar mis sitios web, aunque no recomiendo ninguno en particular. Lo importante es elegir el servicio que mejor se adapte a tus necesidades, presupuesto y confiabilidad, ya que existen muchas alternativas en el mercado. También es posible usar **GitHub Pages** para proyectos más simples o personales, lo que permite publicar sitios web de manera gratuita sin necesidad de hosting tradicional.

Como normalmente la forma de subir sitios web a GitHub va cambiando, prefiero dejarles este link a continuación donde tienen un instructivo actualizado:
https://docs.github.com/es/pages/quickstart

Funcionalidades

HTML (HyperText Markup Language)

Como vimos anteriormente, HTML es el **lenguaje de marcado** que sirve para **estructurar el contenido de una página web**. Cada elemento de HTML cumple una función específica y se representan mediante **etiquetas**.

Principales elementos y su funcionalidad

1. **Encabezados** (<h1> a <h6>): Definen títulos y subtítulos, `<h1>` es el más importante y `<h6>` el menos.

```
<h1>Título principal</h1>
<h2>Subtítulo</h2>
```

2. **Párrafos** (<p>): Agrupan texto en bloques separados.

```
<p>Este es un párrafo de ejemplo.</p>
```

3. **Listas**

   Ordenadas (<ol>): muestran elementos numerados.

```
<ol>
  <li>Primer elemento</li>
  <li>Segundo elemento</li>
</ol>
```

   Desordenadas (<ul>**)**: muestran elementos con viñetas.

```
<ul>
  <li>Elemento A</li>
  <li>Elemento B</li>
</ul>
```

4. **Enlaces** (<a>): Permiten navegar a otras páginas o sitios web.

```
<a href="https://www.ejemplo.com">Ir a ejemplo.com</a>
```

5. **Imágenes** (<img>): Muestran imágenes en la página.

```
<img src="imagen.jpg" alt="Descripción de la imagen">
```

6. **Tablas** (<table>): Organizan información en filas y columnas.

```
<table border="1">
  <tr>
    <th>Nombre</th>
    <th>Edad</th>
  </tr>
  <tr>
    <td>Ana</td>
    <td>25</td>
  </tr>
</table>
```

7. **Formularios** (<form>): Permiten que los usuarios envíen información.

```
<form>
  <label>Nombre:</label>
  <input type="text" name="nombre">
  <button type="submit">Enviar</button>
</form>
```

Es importante aclarar que **el simple hecho de crear un formulario no envía automáticamente un correo ni guarda la información en ningún lado**. Para que los datos lleguen a un email o se registren en una base de datos, es necesario **configurar un servidor o usar un servicio externo** que procese y gestione esos envíos. El formulario solo genera la interfaz y permite que el usuario escriba datos; la acción de "enviar" requiere programación adicional en **backend** o herramientas de terceros.

8. **Divisiones y contenedores** ( ): Agrupan bloques de contenido para aplicar estilos o estructura.

```
<div>
  <h2>Sección</h2>
  <p>Contenido dentro de la sección.</p>
</div>
```

9. **Span** (): Permite aplicar estilos a una **parte del texto** dentro de un párrafo u otro contenedor.

```
<p>Hola, <span style="color:red;">mundo</span>!</p>
```

10. **Multimedia** (<audio> y <video>): Reproducir sonido o video directamente en la página.

```
<audio controls>
  <source src="audio.mp3" type="audio/mpeg">
</audio>

<video width="320" height="240" controls>
  <source src="video.mp4" type="video/mp4">
</video>
```

11. **Comentarios**: Sirven para agregar notas en el código que no se muestran en la página.

```
<!-- Esto es un comentario -->
```

12. **Otros elementos útiles**
    - **
**: salto de línea.
    - **<hr>**: línea horizontal separadora.
    - ****: resaltar texto en negrita semántica.
    - ****: enfatizar texto (cursiva semántica).

CSS (Cascading Style Sheets)

Ya vimos que CSS permite **dar estilo y diseño visual a una página web**. Mientras HTML define la estructura y contenido, CSS define **cómo se ve**: colores, tamaños, posiciones, fuentes y animaciones.

Principales funcionalidades y ejemplos

1. **Colores y fondos**: Cambiar el color del texto y del fondo de elementos.

```
body {
  background-color: lightblue; /* Fondo */
  color: darkblue;             /* Color del texto */
}
```

2. **Tipografía**: Cambiar tipo de letra, tamaño y estilo.

```
h1 {
  font-family: Arial, sans-serif;
  font-size: 24px;
  font-weight: bold;
}
```

3. **Márgenes y relleno (padding)**: Ajustar espacios internos y externos de los elementos.

```
p {
  margin: 20px;  /* Espacio fuera del párrafo */
  padding: 10px; /* Espacio dentro del párrafo */
}
```

4. **Bordes y esquinas redondeadas**: Dar forma y estilo a los contenedores.

```
div {
  border: 2px solid black;
  border-radius: 10px; /* Esquinas redondeadas */
}
```

5. **Tamaño y posición**: Controlar ancho, alto y ubicación de elementos.

```
img {
  width: 200px;
  height: 150px;
}

.caja {
  position: absolute;
  top: 50px;
  left: 100px;
}
```

6. **Listas y enlaces**: Cambiar estilo de listas o enlaces.

```
ul {
  list-style-type: square; /* Viñetas cuadradas */
}

a {
  text-decoration: none; /* Quitar subrayado */
  color: red;
}
```

7. **Flexbox y Grid (Diseño avanzado)**: Distribuir elementos en filas y columnas de manera flexible.

```
.contenedor {
  display: flex;
  justify-content: space-between; /* Separación entre elementos */
}

.grid {
  display: grid;
  grid-template-columns: repeat(3, 1fr);
  gap: 10px;
}
```

8. **Animaciones y transiciones**: Crear efectos visuales al cambiar propiedades.

```
button {
  background-color: blue;
  color: white;
  transition: background-color 0.3s;
}

button:hover {
  background-color: darkblue; /* Cambia al pasar el mouse */
}
```

   Aunque CSS permite animaciones simples como esta, en **general las animaciones más complejas o interactivas se manejan con JavaScript**, ya que permite controlar el momento exacto del efecto, combinar múltiples animaciones y responder a distintos eventos más allá del hover.

9. **Visibilidad y opacidad**: Mostrar, ocultar o hacer transparente un elemento.

```
.oculto {
  display: none;   /* No se muestra */
}

.semi-transparente {
  opacity: 0.5;    /* 50% visible */
}
```

10. **Pseudo-clases y pseudo-elementos**: Aplicar estilos según estado o posición dentro del DOM.

```
a:hover {
  color: green; /* Cuando el mouse está sobre el enlace */
}

p::first-letter {
  font-size: 24px; /* Primera letra del párrafo más grande */
}
```

El **DOM (Document Object Model)** es una **representación estructurada de todos los elementos de una página web**. Cada etiqueta HTML se convierte en un "nodo" dentro de este árbol, lo que permite que los lenguajes como CSS y JavaScript **accedan, modifiquen o interactúen con los elementos de la página**. Por ejemplo, cuando usamos `a:hover` estamos diciendo "aplica un estilo al nodo `<a>` cuando esté en un estado de hover", y cuando usamos `p::first-letter` nos referimos específicamente al primer carácter de un nodo `<p>`.

En otras palabras, el **DOM es la estructura que conecta el HTML con CSS y JavaScript**, y nos permite aplicar estilos o acciones de manera dinámica según el contenido o la interacción del usuario.

Perfecto, acá te armo un **resumen didáctico de JavaScript**, explicando lo básico que se suele usar y dejando la puerta abierta a cosas más complejas, como para tu instructivo y el videojuego que desarrollaste.

JavaScript

JavaScript es un lenguaje de programación que se utiliza en la **web para agregar interactividad y comportamiento dinámico** a las páginas. Mientras HTML define la estructura y CSS define el estilo, JavaScript permite que los elementos **respondan a eventos**, cambien su contenido o se comporten de manera dinámica.

Funcionalidades básicas más utilizadas

1. **Mostrar mensajes**: Permite comunicarse con el usuario a través de alertas, la consola o el documento.

   ```
   alert("¡Hola, bienvenido!");
   console.log("Mensaje en la consola");
   ```

2. **Manipular el DOM**: Permite **cambiar contenido, estilo o estructura de la página** en tiempo real.

   ```
   document.getElementById("titulo").innerText = "Nuevo título";
   document.querySelector("p").style.color = "red";
   ```

3. **Eventos**: Responder a acciones del usuario como clicks, teclas presionadas o movimientos del mouse.

   ```
   document.querySelector("button").addEventListener("click", function() {
       alert("¡Hiciste clic en el botón!");
   });
   ```

4. **Variables y constantes**: Guardar y usar información dentro del programa.

```
let edad = 25;
const PI = 3.1416;
```

En JavaScript, **`let`** se utiliza para declarar variables cuyo valor puede cambiar a lo largo del programa, mientras que **`const`** se usa para declarar constantes cuyo valor no puede modificarse después de asignarlo. Por ejemplo, `let edad = 25;` permite que luego podamos cambiar edad a otro número, mientras que `const PI = 3.1416;` fija un valor que permanecerá constante. Se usa **`let`** y no "led" porque `let` es la palabra reservada correcta en JavaScript; proviene del inglés "let" que significa "permitir", haciendo referencia a que la variable puede recibir distintos valores durante la ejecución del programa.

5. **Condicionales**: Tomar decisiones según ciertas condiciones.

```
if (edad >= 18) {
    console.log("Eres mayor de edad");
} else {
    console.log("Eres menor de edad");
}
```

6. **Bucles (loops)**: Repetir acciones varias veces, útil para recorrer arrays o generar elementos dinámicamente.

```
let numeros = [1,2,3,4,5];
for(let i = 0; i < numeros.length; i++){
    console.log(numeros[i]);
}
```

7. **Arrays y objetos**: Guardar colecciones de datos o estructuras más complejas.

```
let frutas = ["Manzana", "Banana", "Cereza"];
let persona = {nombre: "Ana", edad: 25};
```

8. **Funciones**: Agrupar código que realiza una acción específica y se puede reutilizar.

```
function saludar(nombre){
    console.log("Hola " + nombre);
}
saludar("Facundo");
```

Nota importante

JavaScript tiene **muchas más funcionalidades avanzadas**, como animaciones complejas, comunicación con servidores, almacenamiento local, APIs, clases, módulos, y mucho más. Pero para empezar a programar juegos simples, como el mini-juego que desarrollamos, **estas son las herramientas fundamentales**: manipulación del DOM, eventos, variables, condicionales, bucles, arrays y funciones.

¿Se pueden hacer videojuegos web utilizando HTML, CSS y JavaScript?

Sí, es totalmente posible crear videojuegos que se ejecuten directamente en un **navegador web** usando **HTML, CSS y JavaScript**. HTML nos permite estructurar el documento del juego, CSS se encarga de la presentación visual y los estilos (colores, tamaños, animaciones simples), y JavaScript añade **interactividad y lógica**, como mover personajes, detectar colisiones o actualizar puntuaciones en tiempo real, etc.

Antes, muchos juegos web se desarrollaban con **Flash**, pero esta tecnología dejó de usarse por problemas de compatibilidad y seguridad. Actualmente, HTML, CSS y JavaScript son los estándares modernos para videojuegos web, aunque también es posible crear juegos con motores más avanzados, como **Unity**, y luego exportarlos para que se ejecuten en un sitio web. Sin embargo, incluso sin estos motores, se pueden desarrollar videojuegos completos usando únicamente HTML, CSS y JavaScript, combinando además **elementos multimedia** como imágenes, videos o audios que se suman a la página para enriquecer la experiencia.

Para **probar los videojuegos**, no es necesario subirlos a un servidor o tener un dominio; se pueden ejecutar directamente desde el navegador de manera local, simplemente abriendo el archivo HTML. Esto facilita mucho la fase de desarrollo y testeo antes de publicarlos en línea.

Mini-juego: Batalla en Coordenadas

El mini-juego **"Batalla en Coordenadas"** es un ejemplo práctico que desarrollé para mostrarles cómo se puede crear un **juego web utilizando únicamente HTML, CSS y JavaScript**. Se encuentra disponible para jugar y para revisar el código de forma online en este enlace: https://facundosuarez.com/minijuegos/BatallaEnCoordenadas.html. En este videojuego, el jugador se desplaza por un mapa representado con coordenadas (`x, y`), enfrentando enemigos y gestionando recursos como **vida, energía y monedas**.

La **estructura básica** del juego está definida en HTML. Se utilizan botones para los controles (mover arriba, abajo, izquierda y derecha, deshacer movimientos y atacar enemigos) y un área de texto donde se muestra el estado del juego en tiempo real. CSS se encarga de dar un estilo sencillo a los botones y al área de salida, mientras que JavaScript maneja toda la **lógica y la interactividad del juego**.

El jugador comienza en la posición (`0,0`) con **vida en 100**, **energía en 50** y **0 monedas**. Los enemigos se ubican en distintas coordenadas dentro del mapa y cada uno tiene un estado "activo" que indica si sigue vivo. Al mover al jugador, el juego calcula la posición más cercana de un enemigo activo y muestra una indicación de la dirección hacia donde se encuentra, lo que permite al usuario planificar sus movimientos estratégicamente.

Cuando el jugador se encuentra en la misma posición que un enemigo, **pierde automáticamente 10 puntos de vida**, simulando un ataque del enemigo. Si la vida llega a cero, el juego muestra un mensaje de "¡Moriste!" y se reinicia automáticamente, restaurando tanto la posición del jugador como la vida, energía, monedas y el estado de todos los enemigos.

El jugador puede **atacar a un enemigo** solo si está en la misma posición que él y tiene al menos 5 puntos de energía. Cada ataque consume 5 de energía y, si derrota al enemigo, gana una moneda. Esto permite incorporar elementos de gestión de recursos, como la energía y las monedas, para hacer la experiencia más dinámica y estratégica. Además, el juego reconoce cuando **todos los enemigos han sido derrotados** y muestra un mensaje de victoria: "¡Venciste a todos los enemigos!".

Otro elemento interesante del juego es la **pila de movimientos**, que permite al jugador deshacer su último movimiento. Esto introduce un concepto de memoria y retroceso, útil para corregir errores o replantear estrategias, y sirve como ejemplo de cómo se pueden usar **estructuras de datos básicas como pilas y arrays** en la programación de un juego.

A continuación, analicemos el código completo:

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Mini-juego Batalla en Coordenadas</title>
    <style>
        body {
            font-family: Arial, sans-serif;
            padding: 20px;
        }

        button {
            margin: 5px;
            padding: 10px 15px;
            font-size: 16px;
        }

        .output {
            margin-top: 15px;
            white-space: pre-line;
            border: 1px solid #ccc;
            padding: 10px;
            height: 250px;
            overflow-y: auto;
        }
    </style>
</head>
<body>
    <h1>Mini-juego Batalla en Coordenadas</h1>

    <button onclick="mover('Arriba')">Mover Arriba</button>
    <button onclick="mover('Abajo')">Mover Abajo</button>
    <button onclick="mover('Izquierda')">Mover Izquierda</button>
    <button onclick="mover('Derecha')">Mover Derecha</button>
    <button onclick="deshacer()">Deshacer Último Movimiento</button>
    <button onclick="atacarEnemigo()">Atacar Enemigo</button>

    <div class="output" id="output"></div>

    <script>
        const output = document.getElementById('output');

        // --- Pila de movimientos ---
        let pilaMovimientos = [];

        // --- Jugador ---
        let jugador = {
            x: 0,
            y: 0,
            vida: 100,
            energia: 50,
            monedas: 0
        };

        // --- Enemigos con coordenadas (x, y) ---
        let enemigos = [
            { nombre: "Goblin", x: 2, y: 3, activo: true },
            { nombre: "Orco", x: 5, y: 1, activo: true },
            { nombre: "Troll", x: 1, y: 4, activo: true },
            { nombre: "Esqueleto", x: 3, y: 0, activo: true }
        ];

        // --- Función para mostrar estado ---
        function mostrarEstado() {
            output.textContent += `Jugador: Vida ${jugador.vida}, Energía ${jugador.energia}, Monedas ${jugador.monedas}\n`;
            output.textContent += `Posición Jugador: (${jugador.x},${jugador.y})\n`;
        }

        // --- Función para mover ---
        function mover(direccion) {
            pilaMovimientos.push({ x: jugador.x, y: jugador.y }); // Guardamos movimiento
            switch (direccion) {
                case 'Arriba': jugador.y += 1; break;
                case 'Abajo': jugador.y -= 1; break;
                case 'Izquierda': jugador.x -= 1; break;
                case 'Derecha': jugador.x += 1; break;
            }
            output.textContent += `Jugador se mueve ${direccion} a (${jugador.x},${jugador.y})\n`;

            // Verificar enemigo más cercano
            let enemigoCercano = enemigos
                .filter(e => e.activo)
                .map(e => ({ e: e, distancia: Math.abs(e.x - jugador.x) + Math.abs(e.y - jugador.y) }))
```

```
            .sort((a, b) => a.distancia - b.distancia)[0];

        if (enemigoCercano) {
            let dx = enemigoCercano.e.x - jugador.x;
            let dy = enemigoCercano.e.y - jugador.y;
            let direccionX = dx > 0 ? "Derecha" : dx < 0 ? "Izquierda" : "";
            let direccionY = dy > 0 ? "Arriba" : dy < 0 ? "Abajo" : "";
            output.textContent += `El enemigo más cercano (${enemigoCercano.e.nombre}) está hacia: ${direccionY} ${direccionX}\n`;
        } else {
            output.textContent += "No hay enemigos activos.\n";
        }

        // Si nos encontramos con un enemigo
        enemigos.forEach(e => {
            if (e.activo && e.x === jugador.x && e.y === jugador.y) {
                jugador.vida -= 10;
                output.textContent += `¡Encontraste al ${e.nombre}! Perdés 10 de vida.\n`;
                if (jugador.vida <= 0) {
                    output.textContent += "¡Moriste! Reiniciando juego...\n";
                    reiniciarJuego();
                    return;
                }
            }
        });

        mostrarEstado();
    }

    // --- Función deshacer ---
    function deshacer() {
        if (pilaMovimientos.length === 0) {
            output.textContent += "No hay movimientos para deshacer.\n";
        } else {
            let ultimo = pilaMovimientos.pop();
            jugador.x = ultimo.x;
            jugador.y = ultimo.y;
            output.textContent += `Se deshizo el movimiento. Nueva posición: (${jugador.x},${jugador.y})\n`;
            mostrarEstado();
        }
    }

    // --- Función atacar ---
    function atacarEnemigo() {
        let enemigo = enemigos.find(e => e.activo && e.x === jugador.x && e.y === jugador.y);
        if (!enemigo) {
            output.textContent += "No hay ningún enemigo acá para atacar.\n";
            return;
        }
        if (jugador.energia < 5) {
            output.textContent += "No tenés suficiente energía para atacar.\n";
            return;
        }
        jugador.energia -= 5;
        enemigo.activo = false;
        jugador.monedas += 1;
        output.textContent += `¡Atacaste y derrotaste al ${enemigo.nombre}! Energía restante: ${jugador.energia}\n`;
        mostrarEstado();

        if (enemigos.every(e => !e.activo)) {
            output.textContent += "¡Venciste a todos los enemigos!\n";
        }
    }

    // --- Reiniciar juego ---
    function reiniciarJuego() {
        jugador = { x: 0, y: 0, vida: 100, energia: 50, monedas: 0 };
        enemigos.forEach(e => e.activo = true);
        pilaMovimientos = [];
        output.textContent += "Juego reiniciado.\n";
        mostrarEstado();
    }

    // Estado inicial
    mostrarEstado();
  </script>
</body>
</html>
```

1. Pila de movimientos

```
let pilaMovimientos = [];
```

- **Qué hace:** Es un array que guarda un historial de las posiciones anteriores del jugador.
- **Para qué sirve:** Permite deshacer el último movimiento, devolviendo al jugador a su posición anterior.
- **Concepto clave:** Esta es una **estructura de datos tipo pila (stack)**, donde el último elemento agregado es el primero en salir (LIFO – Last In, First Out).

2. Jugador y sus recursos

```
let jugador = {
    x: 0,
    y: 0,
    vida: 100,
    energia: 50,
    monedas: 0
};
```

- **Qué hace:** Define un objeto que contiene la posición del jugador (`x` y `y`) y sus recursos (`vida`, `energía`, `monedas`).
- **Para qué sirve:** Permite acceder y modificar fácilmente los datos del jugador en todo momento del juego.
- **Concepto clave:** Los **objetos** en JavaScript permiten agrupar datos relacionados bajo un mismo nombre.

3. Enemigos con coordenadas

```
let enemigos = [
    {nombre: "Goblin", x: 2, y: 3, activo: true},
    {nombre: "Orco", x: 5, y: 1, activo: true},
    {nombre: "Troll", x: 1, y: 4, activo: true},
    {nombre: "Esqueleto", x: 3, y: 0, activo: true}
];
```

- **Qué hace:** Define un array de objetos, donde cada enemigo tiene nombre, posición y un estado `activo`.
- **Para qué sirve:** Permite saber qué enemigos siguen en el juego y dónde están ubicados, para calcular encuentros y ataques.
- **Concepto clave:** Combina **arrays** (lista de enemigos) con **objetos** (cada enemigo tiene propiedades).

4. Mover al jugador

```
function mover(direccion){
    pilaMovimientos.push({x: jugador.x, y: jugador.y}); // Guardamos
movimiento
    switch(direccion){
        case 'Arriba': jugador.y += 1; break;
        case 'Abajo': jugador.y -= 1; break;
        case 'Izquierda': jugador.x -= 1; break;
        case 'Derecha': jugador.x += 1; break;
    }
    ...
}
```

- **Qué hace:** Cambia la posición del jugador según la dirección indicada y guarda la posición anterior en la pila.
- **Para qué sirve:** Permite moverse por el mapa y soporta la opción de deshacer.
- **Concepto clave:** Uso de **switch** para decidir la acción según la entrada del usuario y **modificación de objetos**.

5. Detectar enemigos cercanos

```
let enemigoCercano = enemigos.filter(e => e.activo)
    .map(e => ({e: e, distancia: Math.abs(e.x-jugador.x) + Math.abs(e.y-
jugador.y)}))
    .sort((a,b) => a.distancia - b.distancia)[0];
```

- **Qué hace:**
    1. Filtra los enemigos que aún están activos.
    2. Calcula la distancia Manhattan de cada enemigo respecto al jugador.
    3. Ordena los enemigos por distancia y toma el más cercano.

- **Para qué sirve:** Indica al jugador **hacia dónde está el enemigo más cercano**.
- **Concepto clave:** Combina **arrays**, **funciones de orden superior** (`filter`, `map`, `sort`) y cálculo de distancias.

6. Encontrarse con un enemigo

```
enemigos.forEach(e=>{
    if(e.activo && e.x === jugador.x && e.y === jugador.y){
        jugador.vida -= 10;
        ...
    }
});
```

- **Qué hace:** Recorre todos los enemigos y verifica si alguno está en la misma posición que el jugador.
- **Para qué sirve:** Aplica daño al jugador si coincide con un enemigo.
- **Concepto clave:** Uso de **forEach** y condicionales para verificar colisiones.

7. Atacar enemigo

```
function atacarEnemigo(){
    let enemigo = enemigos.find(e => e.activo && e.x === jugador.x && e.y
=== jugador.y);
    if(!enemigo) return;
    jugador.energia -= 5;
    enemigo.activo = false;
    jugador.monedas += 1;
}
```

- **Qué hace:**
    1. Encuentra un enemigo activo en la misma posición que el jugador.
    2. Resta energía al jugador.
    3. Marca al enemigo como derrotado y suma una moneda.
- **Concepto clave:** Uso de **find** para localizar un elemento en un array y **modificación de propiedades de objetos**.

Mini-juego: Enfrentamiento por carriles

Luego, basándome un poco en la idea del mini-juego anterior "Batalla en Coordenadas", desarrollé esta variante llamada **"Enfrentamiento por Carril"**, que si lo observan ahora, resulta un poco más interesante visualmente, aunque sigue siendo un ejemplo sencillo y didáctico. Al igual que el anterior, **solo utiliza HTML, CSS y JavaScript**, sin incorporar imágenes ni audios externos, lo que permite que el código sea más fácil de entender y manipular.

Este videojuego está disponible para jugarlo y revisar su funcionamiento en el siguiente enlace: https://facundosuarez.com/minijuegos/EnfrentamientoPorCarril.html

Además, realicé una versión con botones específicos para dispositivos móviles (celulares, tablets, etc.), disponible acá:
https://facundosuarez.com/minijuegos/EnfrentamientoPorCarril_DispMoviles.html

No obstante, en este instructivo **voy a explicar la primera versión**, porque es más sencilla de comprender; la segunda es básicamente igual, solo que incorpora botones táctiles para facilitar el control en pantallas táctiles.

Cómo funciona el juego

El jugador se mueve por distintos **carriles horizontales** representados en el área de juego y debe enfrentarse a los enemigos que aparecen en dichos carriles. Para controlar al jugador, se utilizan las teclas:

- **W / S**: mover hacia arriba o abajo entre los carriles.
- **A / D**: mover a la izquierda o derecha dentro del carril.
- **E**: atacar al enemigo que esté en el mismo carril.
- **Z**: deshacer el último movimiento realizado.

El jugador cuenta con **recursos** que se muestran en pantalla:

- **Vida**: disminuye si un enemigo lo alcanza.
- **Energía**: disminuye en 5 unidades cada vez que se realiza un ataque.
- **Monedas**: se incrementan en 1 por cada enemigo derrotado.

Los enemigos se mueven automáticamente hacia la izquierda del escenario y, si colisionan con el jugador, le causan daño. Cada vez que el jugador es golpeado, se mueve automáticamente al **carril más bajo** en la esquina izquierda, evitando colisiones repetidas inmediatas.

Cuando un enemigo es derrotado, desaparece del área de juego y suma una moneda al jugador. Al eliminar todos los enemigos, se muestra un mensaje indicando que el jugador **ha vencido a todos los enemigos**. Si la vida del jugador llega a cero, aparece un mensaje de **"¡Moriste!"** y el juego se reinicia automáticamente, reiniciando posiciones, enemigos y recursos.

Detalles técnicos

Este minijuego utiliza **HTML** para crear la estructura del área de juego y los elementos visuales (jugador, enemigos), aplica **CSS** para dar estilo y animaciones básicas, como el movimiento suave de los elementos o el diseño del área de juego y emplea **JavaScript** para la lógica del juego: movimientos, ataque, control de recursos, animación de enemigos y reinicio automático.

El código hace uso de **arrays** y **objetos** para representar los enemigos y el jugador, una **pila de movimientos** para permitir deshacer acciones y cálculos de colisiones y posiciones dentro de los carriles. Además, se implementa un bucle con `requestAnimationFrame` para animar los enemigos de forma continua.

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Mini-juego Enfrentamiento por Carril</title>
    <style>
        body { font-family: Arial; margin: 20px; }
        #gameArea {
            width: 600px;
            height: 400px;
            border: 2px solid black;
            position: relative;
            background: linear-gradient(to bottom, #b3e5fc, #81d4fa);
            overflow: hidden;
        }
        .jugador, .enemigo {
            width: 40px;
            height: 40px;
            position: absolute;
            display: flex;
            align-items: center;
            justify-content: center;
            font-weight: bold;
            color: white;
            border-radius: 5px;
            transition: all 0.2s;
        }
        .jugador { background-color: blue; }
        .enemigo { border: 2px solid black; }
        #recursos { margin-top: 10px; font-weight: bold; }
    </style>
</head>
<body>
    <h2>Mini-juego Enfrentamiento por Carril</h2>
    <div id="gameArea"></div>
    <div id="recursos"></div>
    <p>
        Usá las teclas <b>W A S D</b> para moverte,
        <b>E</b> para atacar al enemigo en tu carril,
        <b>Z</b> para deshacer el último movimiento.
    </p>

    <script>
        const gameArea = document.getElementById('gameArea');
        const recursosDiv = document.getElementById('recursos');

        // --- Carriles ---
        const carriles = [0, 50, 100, 150];

        // --- Variables del juego ---
        let jugador, pilaMovimientos, colaEnemigos, recursos;

        // --- Función para inicializar/reiniciar el juego ---
        function iniciarJuego() {
            // Limpiar gameArea
            gameArea.innerHTML = '';

            // --- Jugador ---
            jugador = {x: 0, y: carriles[carriles.length - 1], div: null, vida: 100};
            jugador.div = document.createElement('div');
            jugador.div.classList.add('jugador');
            jugador.div.innerText = "P";
            gameArea.appendChild(jugador.div);

            // --- Recursos ---
            recursos = [jugador.vida, 50, 0]; // Vida, Energía, Monedas

            // --- Pila de movimientos ---
            pilaMovimientos = [];

            // --- Cola de enemigos ---
            colaEnemigos = [
                {nombre: "Gárgola", color: "darkred", div: null, top:0},
```

```
            {nombre: "Gólem", color: "brown", div: null, top:50},
            {nombre: "Salvaje", color: "green", div: null, top:100},
            {nombre: "Caníbal", color: "purple", div: null, top:150}
        ];

        // Crear enemigos visuales
        colaEnemigos.forEach((e) => {
            e.div = document.createElement('div');
            e.div.classList.add('enemigo');
            e.div.style.backgroundColor = e.color;
            e.div.style.top = e.top + "px";
            e.div.style.left = "560px";
            e.div.innerText = e.nombre[0];
            gameArea.appendChild(e.div);
        });

        actualizarRecursos();
        actualizarJugador();
    }

    // --- Funciones ---
    function actualizarRecursos() {
        recursosDiv.innerText = `Vida: ${recursos[0]} | Energía: ${recursos[1]} | Monedas:
${recursos[2]}`;
    }

    function actualizarJugador() {
        jugador.div.style.left = jugador.x + "px";
        jugador.div.style.top = jugador.y + "px";
    }

    function mover(dx, dy) {
        pilaMovimientos.push({x: jugador.x, y: jugador.y});
        jugador.x = Math.max(0, Math.min(560, jugador.x + dx));
        if(dy !== 0){
            let index = carriles.indexOf(jugador.y);
            if(index !== -1){
                let  nuevoIndex  =  Math.max(0,  Math.min(carriles.length  -1,  index  +
(dy/50)));
                jugador.y = carriles[nuevoIndex];
            }
        }
        actualizarJugador();
    }

    function deshacer() {
        if(pilaMovimientos.length > 0) {
            let ultima = pilaMovimientos.pop();
            jugador.x = ultima.x;
            jugador.y = ultima.y;
            actualizarJugador();
        }
    }

    function moverJugadorACarril(carrilIndex){
        jugador.y = carriles[carrilIndex];
        actualizarJugador();
    }

    // --- Ataque ---
    function atacarEnemigo() {
        if(recursos[1] <= 0){
            alert("¡No tenés suficiente energía para atacar!");
            return;
        }

        // Restar 5 de energía por ataque
        recursos[1] -= 5;
        if(recursos[1] < 0) recursos[1] = 0;
        actualizarRecursos();

        let enemigoAtacado = null;
        for(let i=0; i<colaEnemigos.length; i++) {
```

```
                if(colaEnemigos[i].top === jugador.y) {
                    enemigoAtacado = colaEnemigos[i];
                    colaEnemigos.splice(i,1);
                    break;
                }
            }
            if(enemigoAtacado){
                gameArea.removeChild(enemigoAtacado.div);
                alert(`¡Atacaste al ${enemigoAtacado.nombre}!`);

                // Sumar moneda
                recursos[2] += 1;
                actualizarRecursos();

                if(colaEnemigos.length === 0){
                    alert("¡Venciste a todos los enemigos!");
                }
            } else {
                alert("No hay enemigos en tu carril para atacar.");
            }
        }

        // --- Animación enemigos ---
        function animarEnemigos() {
            colaEnemigos.forEach(e => {
                let x = parseInt(e.div.style.left);
                e.div.style.left = (x-1) + "px";

                // Colisión jugador
                if(e.top === jugador.y && x <= jugador.x + 40 && x + 40 >= jugador.x){
                    let danio = Math.floor(Math.random()*10)+5;
                    recursos[0] -= danio;
                    if(recursos[0] <= 0){
                        recursos[0] = 0;
                        actualizarRecursos();
                        alert("¡Moriste!");
                        iniciarJuego();
                        return;
                    }
                    actualizarRecursos();
                    alert(`${e.nombre} te atacó y causó ${danio} de daño!`);

                    // Mover jugador a esquina inferior izquierda (carril más bajo)
                    jugador.x = 0;
                    moverJugadorACarril(carriles.length - 1);

                    // Reiniciar enemigo
                    e.div.style.left = "560px";
                }

                if(x < -40) e.div.style.left = "560px";
            });
            requestAnimationFrame(animarEnemigos);
        }

        // --- Controles ---
        document.addEventListener('keydown', (e) => {
            switch(e.key.toLowerCase()){
                case 'w': mover(0,-50); break;
                case 's': mover(0,50); break;
                case 'a': mover(-50,0); break;
                case 'd': mover(50,0); break;
                case 'e': atacarEnemigo(); break;
                case 'z': deshacer(); break;
            }
        });

        // --- Iniciar juego ---
        iniciarJuego();
        animarEnemigos();
    </script>
</body>
</html>
```

1. Área de juego (gameArea)

```
const gameArea = document.getElementById('gameArea');
```

- Este `div` es el contenedor donde se muestran todos los elementos del juego: el jugador y los enemigos.
- Sus dimensiones y estilo se definen en CSS (`width: 600px; height: 400px;`) y es **relativo**, lo que permite posicionar elementos absolutamente dentro de él con `position: absolute.`

2. Jugador

```
jugador = {x: 0, y: carriles[carriles.length - 1], div: null, vida: 100};
```

- El jugador tiene coordenadas `x` y `y`, un `div` que lo representa visualmente y una cantidad de `vida`.
- `div` se crea dinámicamente y se agrega al `gameArea` para que se vea en pantalla.
- Se actualiza su posición cada vez que se mueve con `actualizarJugador()`.

3. Carriles

```
const carriles = [0, 50, 100, 150];
```

- Representa las posiciones verticales donde puede estar el jugador y los enemigos.
- Cada carril tiene una coordenada y fija, lo que simplifica la lógica de ataque y colisión.

4. Cola de enemigos

```
colaEnemigos = [
    {nombre: "Gárgola", color: "darkred", div: null, top:0},
    ...
];
```

- Cada enemigo tiene un nombre, un `color` para diferenciarlo visualmente, un `div` que representa su posición y una coordenada `top` que indica su carril.
- Esta "cola" permite que los enemigos se muevan y se ataquen por orden.

5. Mover jugador

```
function mover(dx, dy) { ... }
```

- `dx` y dy determinan cuánto se mueve el jugador horizontal y verticalmente.
- Se utiliza `Math.max` y `Math.min` para que el jugador no salga del área del juego.
- La función guarda el movimiento en `pilaMovimientos` para poder deshacerlo.

6. Deshacer movimiento

```
function deshacer() { ... }
```

- Permite regresar a la posición anterior del jugador sacando el último movimiento de la `pilaMovimientos`.
- Funciona como un **"undo"** básico.

7. Atacar enemigo

```
function atacarEnemigo() { ... }
```

- Verifica que haya energía suficiente para atacar (resta 5 por ataque).
- Busca un enemigo en el mismo carril (`top === jugador.y`), lo elimina del `gameArea` y suma 1 moneda.
- Muestra alertas para informar al jugador del resultado.
- Al final, verifica si todos los enemigos fueron derrotados para avisar la victoria.

8. Animación de enemigos

```
function animarEnemigos() { ... }
```

- Mueve los enemigos de derecha a izquierda restando 1 a su left.
- Comprueba colisión: si un enemigo llega al jugador en el mismo carril, le hace daño aleatorio.
- Si la vida llega a 0, muestra "¡Moriste!" y reinicia el juego.
- `requestAnimationFrame(animarEnemigos)` hace que la animación sea fluida y continua.

9. Controles de teclado

```
document.addEventListener('keydown', (e) => { ... });
```

- Asocia teclas W, A, S, D para moverse, E para atacar y Z para deshacer.
- `e.key.toLowerCase()` permite que funcione con mayúsculas y minúsculas.

10. Inicialización

```
iniciarJuego();
animarEnemigos();
```

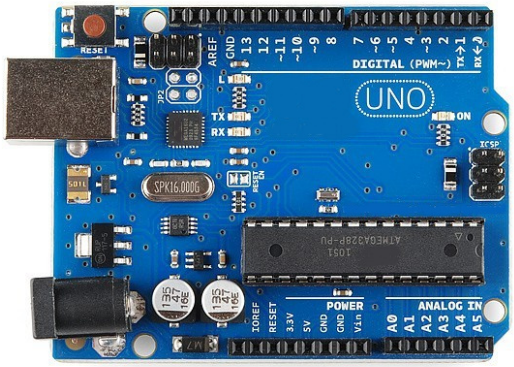
- Llama a las funciones principales para:
    1. Crear al jugador y enemigos.
    2. Configurar sus posiciones y recursos.
    3. Iniciar la animación de los enemigos.

Aclaraciones importantes

- **Recursos**: `[vida, energia, monedas]` se actualizan en pantalla con `actualizarRecursos()`.
- **Alertas**: Informan al jugador de acciones importantes (ataque, daño recibido, victoria o muerte).
- **Coordenadas fijas**: Facilitan la lógica de colisión y ataque sin necesidad de un sistema de física complejo.
- **Uso de `div` dinámicos**: Permite que el juego sea completamente en HTML/CSS/JS sin necesidad de imágenes externas.

Placas de Desarrollo

Las **placas de desarrollo** son herramientas de hardware diseñadas para facilitar la creación y prueba de proyectos electrónicos y de programación. Se utilizan para aprender, experimentar y prototipar sistemas que combinan software y componentes físicos, como sensores, motores, luces y pantallas. Entre las más conocidas se encuentran **Arduino**, **Raspberry Pi Pico**, **ESP32**, entre muchas otras. Todas tienen en común que permiten escribir un programa en la computadora, cargarlo en la placa y, a partir de allí, controlar entradas (como botones o sensores) y salidas (como LEDs, parlantes o motores). Estas placas se convirtieron en un pilar fundamental tanto en la enseñanza de la programación como en el desarrollo de proyectos de robótica, domótica y arte interactivo. A pesar de todo lo anteriormente mencionado, es importante aclarar que igualmente su uso no se limita únicamente a la experimentación o al aprendizaje: también existen **obras de arte electrónico**, instalaciones interactivas e incluso **videojuegos físicos** desarrollados con estas placas, donde la programación y la electrónica se combinan con la creatividad artística y lúdica.



Arduino y Raspberry Pi Pico

Arduino y Raspberry Pi Pico son dos de las placas más populares para introducirse en el mundo de la electrónica y la programación de dispositivos físicos. Ambas permiten interactuar con el mundo real mediante sensores, actuadores, motores, luces y más.

- **Arduino**: se programa principalmente con C/C++ mediante el IDE oficial de Arduino. Está pensado para principiantes y tiene una gran comunidad y cantidad de tutoriales.
- **Raspberry Pi Pico**: se basa en un microcontrolador RP2040 y puede programarse tanto en C/C++ como en MicroPython. Es más flexible, pero también un poco más avanzada para ciertos proyectos. Podemos usar el IDE oficial de Arduino también para programar con él.

En ambos casos, la idea es la misma: enviar instrucciones a la placa para que controle pines de entrada (sensores, botones) y salida (LEDs, motores, pantallas, etc.).

Conceptos básicos de electrónica para trabajar con placas de desarrollo

Cuando usamos placas de desarrollo como **Arduino** o **Raspberry Pi Pico** y queremos avanzar en el desarrollo de proyectos relativamente complejos, lamentablemente no alcanza solo con aprender a programar, y es por esto mismo que opté por agregar este pequeño apartado a este texto, con el objetivo de poder aportarles al menos algunas nociones introductorias al respecto. Este tipo de placas funcionan conectándose a componentes electrónicos (LEDs, resistencias, botones, sensores, motores, etc.), y para utilizarlos de manera correcta y segura necesitamos tener algunas nociones de **electrónica básica**.

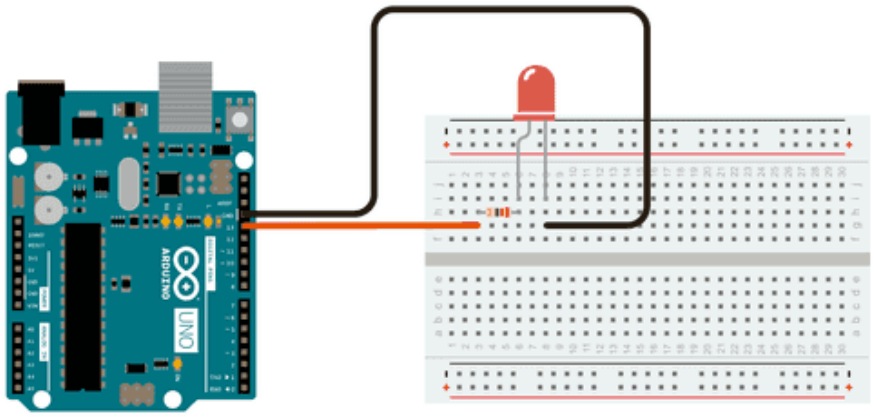
En primer lugar, es fundamental entender qué son **la corriente, el voltaje y la resistencia**.

- **Corriente eléctrica (I)**: es el flujo de electrones que circula por un circuito. Se mide en amperios (A).
- **Voltaje (V)**: es la diferencia de potencial que impulsa esa corriente a través del circuito. Se mide en voltios (V).
- **Resistencia (R)**: es la oposición que ofrece un material al paso de la corriente. Se mide en ohmios (Ω).

La relación entre estos tres conceptos está expresada en la **Ley de Ohm**:

$$V = I \cdot R$$

Esto significa que si conocemos dos de las variables, podemos calcular la tercera. Por ejemplo, si conectamos un LED directamente a un pin de Arduino sin resistencia, la corriente podría ser demasiado alta y dañar el LED o la propia placa. Por eso usamos resistencias que limitan la corriente.



Además, la **Ley de Kirchhoff** nos dice cómo se comporta la corriente y el voltaje en un circuito más complejo. Existen dos leyes:

- **Ley de corrientes de Kirchhoff**: la suma de las corrientes que entran y salen de un nodo es igual a cero (la corriente se reparte entre los caminos disponibles).
- **Ley de tensiones de Kirchhoff**: la suma de todas las diferencias de potencial (voltajes) en un lazo cerrado es igual a cero (el voltaje se distribuye entre los componentes del circuito).

Por otro lado, en Arduino es clave comprender cómo funcionan sus **pines de entrada y salida**:

- Los **pines digitales** (0 o 1, encendido o apagado) permiten leer el estado de un botón o encender un LED, por ejemplo.
- Los **pines analógicos** permiten leer valores más complejos, como la intensidad de luz en un sensor o la posición de un potenciómetro, ya que pueden tomar un rango de valores (0 a 1023 en la mayoría de las placas).

Finalmente, también debemos saber distinguir entre **pines de entrada** (para recibir información desde un sensor o botón) y **pines de salida** (para enviar señales que enciendan un LED, activen un motor, etc.).

Motores

Dentro de los proyectos con placas de desarrollo, los **motores** son uno de los componentes más utilizados para generar movimiento. Existen distintos tipos, y cada uno se adapta a necesidades específicas:

- **Motores de movimiento continuo (DC):** son los más simples y comunes. Al aplicarles corriente, giran en un sentido u otro según la polaridad. No permiten controlar con precisión la posición, solo la dirección y la velocidad aproximada (variando el voltaje o usando técnicas como PWM).
- **Motores paso a paso:** giran en ángulos muy pequeños y definidos (llamados *pasos*). Esto permite controlar con gran precisión la posición del eje, aunque su uso requiere enviar secuencias de pulsos bien programadas. Se utilizan, por ejemplo, en impresoras 3D y plotters.
- **Servomotores:** combinan un motor con un sistema de control interno que permite mover el eje hasta una posición específica y mantenerla. Esto es ideal para proyectos que requieren precisión en el ángulo, como brazos robóticos o mecanismos de dirección.

Un aspecto importante a tener en cuenta es que un **motor también puede comportarse como generador** si lo hacemos girar al revés. Esto significa que puede devolver corriente hacia el circuito, lo cual en algunos casos puede dañar la placa o los componentes conectados. Además, hay que tener en cuenta que siempre, cuando se apaga el motor, la bobina interna genera un pico de tensión en sentido inverso (por la energía almacenada en el campo magnético). Para evitar que esto sea un problema, se suele colocar un **diodo rectificador** en paralelo con el motor, pero en sentido inverso a la corriente normal de funcionamiento, protegiendo la placa que lo controla.

Fundamentos de la Programación en Arduino

Cuando programamos una placa como Arduino, usamos un lenguaje muy parecido a **C/C++**, pero con funciones específicas que permiten interactuar con los pines de entrada y salida de la placa. Estos son algunos de los elementos básicos:

1. `setup()`

Es una función que se ejecuta **una sola vez**, al inicio del programa, cuando la placa se enciende o se reinicia. Sirve para configurar los parámetros iniciales.

```
void setup() {
  pinMode(13, OUTPUT); // Configura el pin 13 como salida
}
```

2. `loop()`

Es la función principal que se repite **en bucle** constantemente. Todo lo que pongamos aquí se ejecutará de forma continua mientras la placa esté encendida.

```
void loop() {
  digitalWrite(13, HIGH); // Enciende el LED
  delay(1000);            // Espera 1 segundo
  digitalWrite(13, LOW);  // Apaga el LED
  delay(1000);            // Espera 1 segundo
}
```

3. `pinMode(pin, mode)`

Sirve para configurar un pin como **entrada** (INPUT) o **salida** (OUTPUT).

```
pinMode(7, INPUT);   // El pin 7 funciona como entrada (ej. botón)
pinMode(8, OUTPUT);  // El pin 8 funciona como salida (ej. LED)
```

4. `digitalWrite(pin, value)`

Permite enviar una señal digital a un pin configurado como salida. Puede ser **HIGH (encendido, 1, 5V/3.3V)** o **LOW (apagado, 0, 0V)**.

```
digitalWrite(8, HIGH); // Enciende el LED conectado al pin 8
digitalWrite(8, LOW);  // Apaga el LED
```

5. `digitalRead(pin)`

Lee el estado de un pin configurado como entrada. Devuelve **HIGH** (si hay voltaje) o **LOW** (si no lo hay).

```
int estadoBoton = digitalRead(7);
```

```
if (estadoBoton == HIGH) {
  digitalWrite(8, HIGH); // Enciende LED si el botón está presionado
}
```

6. `analogWrite(pin, value)`

Genera una señal analógica simulada (PWM: modulación por ancho de pulso). Útil para variar la intensidad de un LED o la velocidad de un motor. El valor va de **0 a 255**.

```
analogWrite(9, 128); // LED en pin 9 con brillo al 50%
```

7. `delay(ms)`

Detiene el programa durante la cantidad de milisegundos indicada.

```
delay(2000); // Pausa de 2 segundos
```

Ejemplos básicos en Arduino

1. **Encender y apagar un LED**

```
int led = 13; // Pin digital 13

void setup() {
  pinMode(led, OUTPUT); // Configurar pin como salida
}

void loop() {
  digitalWrite(led, HIGH); // Encender LED
  delay(1000);             // Esperar 1 segundo
  digitalWrite(led, LOW);  // Apagar LED
  delay(1000);             // Esperar 1 segundo
}
```

Con esto hacemos parpadear un LED cada segundo.

2. **Leer un botón**

```
int boton = 2;  // Pin donde está el botón
int led = 13;   // LED en pin 13

void setup() {
  pinMode(boton, INPUT);
  pinMode(led, OUTPUT);
}

void loop() {
  int estado = digitalRead(boton); // Leer botón
  if (estado == HIGH) {
    digitalWrite(led, HIGH); // Si está presionado, encender LED
  } else {
    digitalWrite(led, LOW);  // Si no, apagar LED
  }
}
```

El LED se enciende solo cuando se presiona el botón.

3. **Leer un sensor analógico (ej. potenciómetro)**

```
int sensor = A0; // Entrada analógica
int valor;

void setup() {
  Serial.begin(9600); // Comunicación con la PC
}

void loop() {
  valor = analogRead(sensor); // Leer valor del potenciómetro
  Serial.println(valor);      // Mostrar en monitor serie
  delay(500);                 // Pausa medio segundo
}
```

El monitor serie mostrará valores entre 0 y 1023 según la posición del potenciómetro.

4. **Mover un servo motor**

```
#include <Servo.h>
Servo servoMotor;

void setup() {
  servoMotor.attach(9); // Conectar servo al pin 9
}

void loop() {
  servoMotor.write(0);    // Girar a 0°
  delay(1000);
  servoMotor.write(90);   // Girar a 90°
  delay(1000);
  servoMotor.write(180);  // Girar a 180°
  delay(1000);
}
```

El servo motor se va a mover entre las posiciones básicas de 0°, 90° y 180°.

Diseño de controles accesibles para videojuegos con Arduino

Una de las aplicaciones más interesantes de Arduino en el campo de los videojuegos es la **creación de mandos personalizados y accesibles**. Esto permite que personas con distintas discapacidades puedan interactuar con los juegos de una manera adaptada a sus necesidades.

Por ejemplo, podríamos desarrollar un **mando alternativo** utilizando Arduino , haciéndolo funcionar como entrada más para un videojuego de PC:

- **Control por voz**: integrando un módulo de reconocimiento de voz (como el EasyVR o el Elechouse Voice Recognition), el jugador podría asignar comandos básicos (“salto”, “disparo”, “izquierda”, “derecha”) y controlarlos hablando.

- **Sensores ultrasónicos**: con un sensor HC-SR04, se podrían medir distancias y mapearlas como acciones en el juego (por ejemplo, acercar la mano para avanzar, alejarla para retroceder).
- **Botones extras o de gran tamaño**: muy útiles para quienes no pueden usar un joystick convencional, ya que Arduino permite crear pulsadores accesibles que simulen las teclas del teclado o botones de un mando.

El Arduino, conectado por USB a la computadora, podría actuar como un **teclado virtual** (usando placas compatibles como Arduino Leonardo o Micro) enviando las señales directamente al videojuego.

Si te interesa aprender en profundidad cómo diseñar interfaces inclusivas que amplíen el acceso a los videojuegos para más personas, o si querés profundizar en el uso de Arduino en general, te recomiendo aprovechar los recursos que la comunidad open source ofrece. Podés visitar la documentación oficial en https://playground.arduino.cc/, donde vas a encontrar guías, ejemplos y referencias detalladas sobre placas, sensores y programación. Además, para resolver dudas, compartir experiencias o aprender de otros usuarios, podés consultar y participar en el foro de Arduino en https://forum.arduino.cc/, que es un espacio actualmente muy activo donde la comunidad colabora constantemente.

Apéndice: Instructivo Desarrollo Videojuego en Unity

Para la materia **"Fundamentos de la Programación I"** de la carrera de Producción y Desarrollo de Videojuegos en la UNPAZ, realicé la siguiente actividad, que corresponde al **Trabajo Práctico 3** que les propongo a mis alumnas y alumnos. La consigna consiste en desarrollar un videojuego en Unity. He decidido incluir el instructivo tal cual, copiando y pegando el material original, a modo de **apéndice**, para quienes puedan estar interesados en realizar la actividad. Si bien podría haberlo adaptado o simplificado para que resultase más genérico, opté por mantenerlo intacto, de manera que se entienda claramente cuál es el objetivo del trabajo; ya que pienso que quizás quienes no estén cursando la materia conmigo, por ahí este material les pueda permitir realizar una especie de **"simulacro de entrega"**, y así poner a prueba los conocimientos adquiridos al leer este material. Además, dejo a disposición los mismos links que les comparto a mis alumnas y alumnos, con los video tutoriales y los materiales que utilizamos como repositorio de códigos. Espero sinceramente que les resulte útil y provechoso.

Consigna:

En este trabajo práctico, las y los estudiantes deberán desarrollar un **videojuego tridimensional (3D) en Unity** que, aunque breve, cuente con un inicio y un cierre definidos, incorporando en su programación al menos un condicional (if), un ciclo for y el uso de un array recorrido en código propio. Además, cada estudiante deberá incorporar al proyecto al menos un **objeto tridimensional propio**, modelado a partir de un elemento de su entorno real (por ejemplo, una taza, una mochila o un cartel), utilizando para ello herramientas como Tripo u otras equivalentes. El propósito de la consigna no es la realización de un proyecto extenso o complejo, sino la construcción de una experiencia acotada que permita ejercitar y consolidar los fundamentos de la programación dentro de un motor profesional. Para ello se recurrirá, en algunos casos, a **"cajas negras"**, entendidas como fragmentos de código previamente preparados y provistos por la cátedra (por ejemplo, el *First Person Controller*), cuyo funcionamiento interno no será objeto de evaluación. Estas herramientas actúan como soporte para que el foco del aprendizaje se ubique en la escritura, análisis y explicación de los scripts que cada estudiante elabore de manera autónoma. Este instructivo se redactó comprendiendo que el paso desde entornos de desarrollo visuales como Scratch, GDevelop, Construct o RPG Maker hacia un motor como Unity requiere de tiempo y práctica; sin embargo, la experiencia indica que mediante ejercicios cortos, concretos y guiados, es posible asimilar progresivamente los conceptos centrales, En este sentido, el objetivo pedagógico de esta actividad no radica en copiar y pegar soluciones ya resueltas, sino en **aprender a programar, comprender qué hace cada script y apropiarse de la lógica que sustenta la creación de un videojuego**.

Materiales/Recursos

Tutoriales introductorios (YouTube)
https://youtu.be/Jv5Xpk6Mz6Q?feature=shared
https://youtu.be/No4xCh-oHOQ?feature=shared
https://youtu.be/mef2p4By7s8?feature=shared

Video sobre Arrays
https://youtu.be/JepSew72nlc?feature=shared

Repositorio de código (cátedra / docente)
https://drive.google.com/drive/folders/1WbwNvS_4YMXC6N9FFgbrzvuhH4qiQ9D6?usp=sharing

Generación de objetos 3D desde fotos (IA sugerida)
https://studio.tripo3d.ai/home

1. Instalación y configuración del entorno

Para comenzar a trabajar en Unity es necesario preparar el entorno de desarrollo de manera ordenada. A continuación, se detallan los pasos recomendados:

1.1 Instalación de Unity Hub y Unity

- Descarguen **Unity Hub**, la aplicación oficial que centraliza la instalación y gestión de proyectos de Unity, desde la página oficial: https://unity.com/download
- Dentro de Unity Hub instalen una versión de Unity **estable.** Es importante que todos los miembros trabajen con la misma versión para evitar incompatibilidades.
- Al instalar, seleccionen también los **módulos de plataformas** que puedan necesitar:
    - **Windows/Mac/Linux Build Support** (para exportar builds de escritorio).
    - **Android Build Support** (si desean compilar en Android).
    - Otros módulos son opcionales y no serán necesarios en este trabajo práctico.

1.2 ¿Es gratis Unity?

Sí, **Unity es gratuito para la mayoría de los creadores individuales y proyectos pequeños**, gracias a la versión **Unity Personal**. Esta edición puede utilizarse sin costo siempre que los ingresos o la financiación del proyecto no superen los **200.000 USD en los últimos 12 meses**.

Detalles de Unity Personal:

- **Gratuita**: acceso completo al motor de Unity sin pagar licencias.
- **Límite de ingresos**: el uso es gratuito siempre que no se superen los 200.000 USD de ingresos o financiación en el período de un año.
- **Orientada a proyectos indie**: resulta ideal para estudiantes, desarrolladores independientes y equipos pequeños que recién comienzan.

¿Cuándo se requiere una licencia de pago?

- **Superar el límite de ingresos**: si el proyecto genera más de 200.000 USD en los últimos 12 meses.
- **Funcionalidades avanzadas**: los planes de pago, como **Unity Pro**, incluyen herramientas adicionales, soporte especializado y opciones diseñadas para estudios profesionales o producciones de gran escala.

En el marco de esta materia, **la versión gratuita Unity Personal es más que suficiente** para todos los trabajos prácticos y proyectos a desarrollar.

1.3 Creación del proyecto

- Abran Unity Hub y seleccionen **New Project**.
- Elijan la plantilla **3D (Core)**, que es la base que utilizaremos en esta materia.
- Asignen un nombre claro al proyecto (por ejemplo: *TP3_Apellido1_ Apellido2* o el nombre del grupo, que puede coincidir con el nombre del juego).
- Elijan una carpeta de destino donde se va a guardar todo el proyecto.

Nota fundamental: todos los archivos de Unity que conforman el proyecto (Assets, ProjectSettings, Library, etc.) tiene que estar dentro de **una misma carpeta**. Esto es lo que va a permitir que el proyecto pueda copiarse, pegarse o trasladarse a otra computadora sin inconvenientes.

1.4 Organización de carpetas

Se recomienda crear dentro de la carpeta **Assets** una estructura clara que facilite el trabajo en grupo:

- Scripts/ → para todos los archivos de código C#.
- Models/ → para los objetos 3D.
- Materials/ → para materiales y texturas.
- Scenes/ → para las escenas del juego.
- Prefabs/ → para objetos reutilizables.
- Audio/ → para efectos de sonido y música.

Mantener esta organización desde el principio les va a evitar confusiones en etapas posteriores del desarrollo.

1.5 Manejo de versiones y trabajo colaborativo

Unity ofrece una **nube oficial (Unity Cloud)**, pero su funcionamiento no siempre es estable ni práctico en todas las circunstancias. Por esa razón, les recomiendo:

- **Realizar backups periódicos** del proyecto (copiando la carpeta completa a un disco externo o a Google Drive).
- Mantener un **proyecto principal o "master"** que concentre el estado más actualizado.
- Adoptar una lógica de trabajo inspirada en la **programación orientada a objetos**: cada integrante del grupo puede avanzar en un script, una escena o un recurso específico en un proyecto individual, y luego esas partes se integran en conjunto dentro del proyecto principal.

Esta dinámica favorece la división de tareas y la posterior integración, a la vez que evita la pérdida de información y facilita la resolución de conflictos.

2. Primer proyecto: escena mínima jugable

En este primer proyecto vamos a construir una **escena mínima jugable** en Unity, es decir, un entorno básico en 3D en el cual el jugador pueda moverse libremente y donde podamos luego incorporar las mecánicas que ustedes programen.

Los pasos serán los siguientes:

1. **Importar o crear un terreno/escena.**
    - Pueden usar un *Plane* o crear un *Terrain* desde el menú: **GameObject > 3D Object**
    - Ajusten la escala para que el área de juego sea suficientemente amplia.
2. **Agregar el *First Person Controller***
    - Como ya hablamos, usaremos algunas **"cajas negras"** para simplificar el desarrollo. Estas son fragmentos de código ya preparados por la cátedra que resuelven tareas básicas y necesarias (como mover al jugador). No forman parte de la evaluación: ustedes no tienen que programarlos ni es necesario comprender en detalle cómo funcionan, aunque sí pueden explorarlos si les interesa.
    - La lógica es: no perder tiempo en reprogramar elementos estructurales del motor, para poder enfocarnos en lo pedagógicamente relevante: sus propios scripts de interacción, arrays, condicionales y bucles.

Un ejemplo de **código sencillo de movimiento** es este:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class SimpleFirstPersonController : MonoBehaviour
{
    public float velocidadMovimiento = 5f;  // Velocidad de movimiento del jugador

    void Update() // Método que se llama una vez por cada fotograma del juego
    {
        // Obtener las entradas de teclado para el movimiento
        float movimientoHorizontal = Input.GetAxis("Horizontal");
        float movimientoVertical = Input.GetAxis("Vertical");

        // Calcular el vector de movimiento
        Vector3 movimiento = transform.forward * movimientoVertical + transform.right * movimientoHorizontal;

        // Mover al jugador
        transform.Translate(movimiento * velocidadMovimiento * Time.deltaTime);
    }
}
```

Este script permite moverse con **WASD**, pero es limitado (no contempla saltar, correr ni caminar lentamente).

En lugar de usar esta versión, vamos a trabajar con un **First Person Controller más completo**, ya disponible en la carpeta de códigos de la cátedra:

En este link tenemos una Carpeta con todos los códigos de First Person Controller:

https://drive.google.com/drive/folders/1p-0dY4BUkirdQ9b7SYztC_JN0OKaH-od

En vez de ese script simple, en este trabajo utilizaremos una versión más completa, que es la que recomiendo para sus proyectos:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class FirstPersonController_DobleSalto_Correr_CaminarLento : MonoBehaviour
{
    public float velocidadMovimiento = 5f;
    public float velocidadCarrera = 10f;
    public float velocidadCaminarLento = 2f;
    public float sensibilidadMouse = 2f;

    private CharacterController characterController;
    private float velocidadVertical = 0f;
    private int saltosRestantes = 2;
    private bool caminarLento = false;

    private void Start()
    {
        characterController = GetComponent<CharacterController>();
        Cursor.lockState = CursorLockMode.Locked;
    }

    void Update()
    {
        float movimientoHorizontal = Input.GetAxis("Horizontal");
        float movimientoVertical = Input.GetAxis("Vertical");
        float mouseX = Input.GetAxis("Mouse X") * sensibilidadMouse;
        transform.Rotate(Vector3.up * mouseX);

        if (characterController.isGrounded)
        {
            velocidadVertical = -0.5f;
            saltosRestantes = 2;
        }
        else
        {
            velocidadVertical += Physics.gravity.y * Time.deltaTime;
        }

        if (Input.GetButtonDown("Jump") && saltosRestantes > 0)
        {
            velocidadVertical = Mathf.Sqrt(2f * 3.0f * Mathf.Abs(Physics.gravity.y));
            saltosRestantes--;
        }

        if (Input.GetKey(KeyCode.LeftControl) && characterController.isGrounded)
        {
            caminarLento = true;
        }
        else
        {
            caminarLento = false;
        }

        float currentSpeed = caminarLento ? velocidadCaminarLento : (Input.GetKey(KeyCode.LeftShift)
? velocidadCarrera : velocidadMovimiento);
        Vector3 movimiento = (transform.forward * movimientoVertical + transform.right *
movimientoHorizontal) * currentSpeed;
        movimiento.y = velocidadVertical;
        characterController.Move(movimiento * Time.deltaTime);
    }
}
```

Este script incluye:

- Movimiento en primera persona con WASD.
- Rotación con el mouse.
- Salto doble.
- Caminar lentamente con Ctrl.
- Correr con Shift.

Para que funcione correctamente:

- Crear un objeto **Cilindro** (GameObject > 3D Object > Cylinder). Se recomienda usarlo como "cuerpo" del jugador.
- Agregarle el componente **Character Controller** (Add Component > Character Controller).
- Agregarle el script del **First Person Controller** (arrastrar desde la carpeta Scripts al cilindro).
- Ajustar la **Main Camera** como hija del cilindro, a la altura de los ojos.
- Agregar un **Capsule Collider** (si no lo tiene ya incorporado el objeto).

Para quienes deseen profundizar en el funcionamiento de estas "cajas negras", recomiendo el siguiente tutorial:

Tutorial sobre los códigos de movimiento

https://www.youtube.com/watch?feature=shared&v=mef2p4By7s8

3. Agregar un objeto 3D propio.

- Importen un archivo **OBJ** creado con Tripo (https://studio.tripo3d.ai/home) u otra herramienta similar.
- También pueden usar un objeto del *Asset Store* siempre y cuando **no incluya código**.
- El objeto debe representar algo de su universo físico real (ejemplo: una taza, mochila, cartel, etc.).

De todas maneras, en este trabajo no se preocupen demasiado por incorporar muchos modelos 3D ni por la calidad visual de los objetos que utilicen. Lo visual, en esta instancia, es secundario: el foco principal está en el pensamiento algorítmico y en cómo traducimos ese pensamiento a un sistema funcional. Lo central es comprender y aplicar correctamente los conceptos de programación que venimos trabajando.

Aun así, me parece importante que cada uno pueda incorporar al menos un objeto 3D elegido por ustedes mismos, aunque sea sencillo. La idea es que experimenten con el proceso completo: importar, colocar y manipular un objeto dentro de la escena. Esta práctica les permite tener un punto de contacto tangible con el mundo virtual que están construyendo, además de afianzar la comprensión de cómo sus líneas de código afectan directamente a los elementos que aparecen en pantalla.

Por eso, aunque no sea necesario que el juego tenga muchos elementos visuales ni que las texturas sean perfectas, considero que hacer esta práctica con un objeto propio es fundamental.

3. Scripting básico en C#

En esta sección vamos a introducir los conceptos centrales de la programación en C# aplicados al desarrollo de videojuegos con Unity. La idea es comenzar con ejemplos simples y concretos, que puedan integrarse fácilmente en los proyectos de cada grupo.

3.1. Script de interacción: uso del `if`

El condicional **`if`** es fundamental para tomar decisiones en un videojuego. Por ejemplo: abrir una puerta al presionar una tecla.

```
using UnityEngine;

public class PuertaInteractiva : MonoBehaviour
{
    public GameObject puerta; // La puerta que se abrirá o cerrará
    private bool abierta = false; // Estado inicial

    void Update()
    {
        if (Input.GetKeyDown(KeyCode.E)) // Al presionar la tecla "E"
        {
            if (abierta)
            {
                puerta.transform.Rotate(0, -90, 0); // Cierra la puerta
                abierta = false;
            }
            else
            {
                puerta.transform.Rotate(0, 90, 0); // Abre la puerta
                abierta = true;
            }
        }
    }
}
```

Este ejemplo muestra cómo una sola tecla puede alternar estados en el juego, algo básico pero clave en cualquier mecánica.

En Unity, cada objeto en la escena tiene un componente llamado **`Transform`**. Este componente es fundamental porque define la posición, rotación y escala del objeto en el espacio 3D (o 2D, si trabajás en 2D). Básicamente, dice dónde está el objeto, cómo está orientado y qué tamaño tiene.

En este código tenemos esta línea:
`puerta.transform.Rotate(0, 90, 0); // Abre la puerta`

- **`puerta`** es el objeto al que queremos afectar.
- **`puerta.transform`** accede al Transform de ese objeto.
- **`Rotate(0, 90, 0)`** indica que queremos rotarlo 90 grados alrededor del eje Y.
    - Eje X → de adelante hacia atrás
    - Eje Y → de izquierda a derecha (vertical)
    - Eje Z → de arriba hacia abajo (profundidad)

En otras palabras, con **`Transform`** podés:

1. Mover un objeto:

   ```
   transform.position = new Vector3(0, 1, 0); // Va a la posición (0,1,0)
   ```

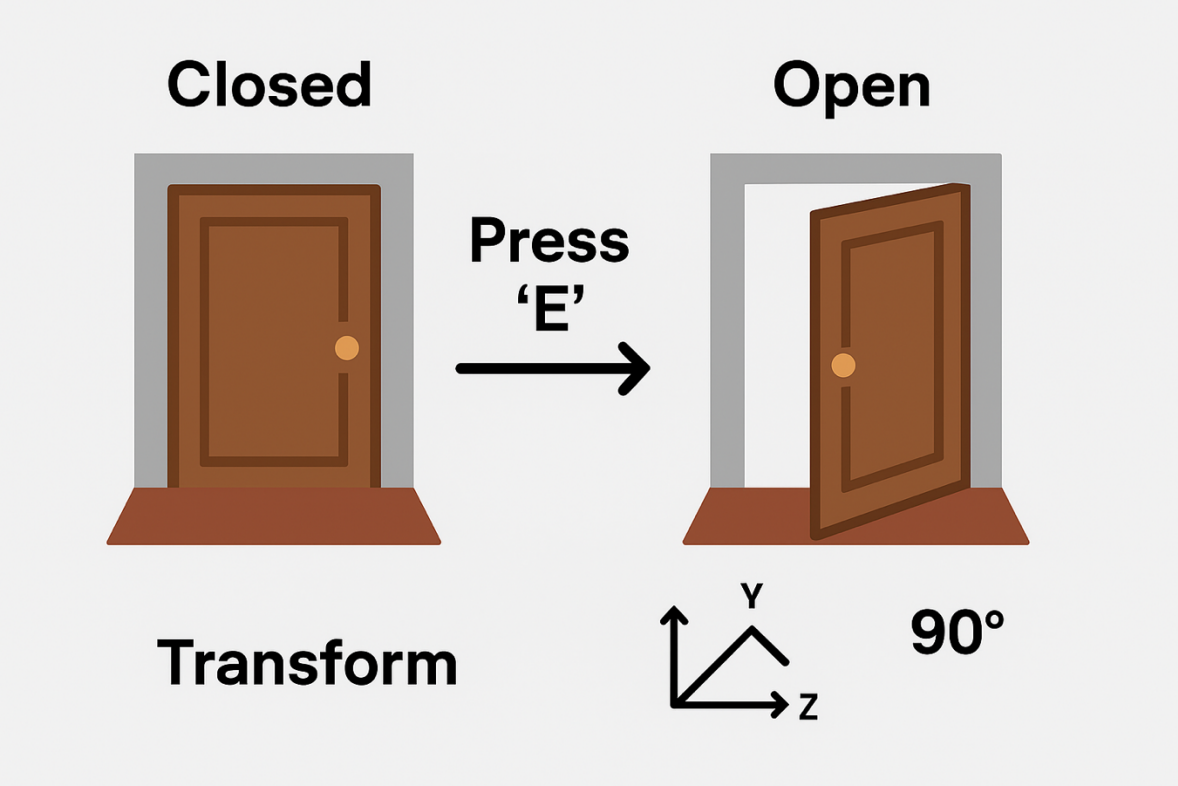
2. Rotar un objeto:

   ```
   transform.Rotate(0, 90, 0); // Rota 90° en Y
   ```

3. Cambiar el tamaño:

   ```
   transform.localScale = new Vector3(2, 2, 2); // Duplica tamaño en X, Y, Z
   ```

En este ejemplo, el `Transform` es usado para abrir y cerrar la puerta girándola. El script alterna entre `abierta = true` y `abierta = false` para decidir si debe rotarla hacia un lado (abrir) o hacia el otro (cerrar).



3.2. Generación de múltiples objetos con `for`

Muchas veces queremos que se repitan acciones de forma automática. Un **`for`** nos permite crear varios objetos iguales o similares de manera sencilla.

```
using UnityEngine;

public class GeneradorDeCubos : MonoBehaviour
{
    public GameObject cuboPrefab; // Prefab del cubo a generar

    void Start()
    {
        for (int i = 0; i < 10; i++) // Genera 10 cubos
        {
            Vector3 posicion = new Vector3(i * 2.0f, 0, 0);
            // Separa los cubos en el eje X
            Instantiate(cuboPrefab, posicion, Quaternion.identity);
        }
    }
}
```

Este script genera automáticamente 10 cubos en fila al iniciar el juego. Es útil para poblar escenarios con obstáculos, enemigos o recolectables.

La variable pública **`cuboPrefab`** permite asignar el cubo desde el Inspector, y dentro del método **`Start()`** se utiliza un bucle **`for`** que se repite 10 veces. En cada iteración, se calcula una posición en el eje X multiplicando el índice **`i`** por 2, dejando los ejes Y y Z en 0, para que los cubos queden alineados horizontalmente y separados uniformemente. Finalmente, **`Instantiate`** crea una instancia del prefab en esa posición sin rotación (**`Quaternion.identity`**), generando así una fila de cubos espaciados uniformemente en la escena.

`Vector3 posicion = new Vector3(i * 2.0f, 0, 0);`	`Instantiate(cuboPrefab, posicion, Quaternion.identity);`
• **`Vector3`** es una estructura que representa una posición en 3 dimensiones: X, Y y Z. • **`i * 2.0f`** en el eje X significa que cada cubo se coloca 2 unidades más lejos que el anterior, según el índice i del bucle (`i = 0, 1, 2`…). Esto crea un espaciado uniforme entre cubos en horizontal. • Los valores 0 en Y y Z colocan todos los cubos a la misma altura y profundidad, alineados sobre el plano XZ.	• `Instantiate` es un método de Unity que crea una copia de un objeto o prefab en la escena en tiempo de ejecución. • Primer parámetro: el prefab que queremos clonar (`cuboPrefab`). • Segundo parámetro: la posición donde se generará el objeto (`posicion`). • Tercer parámetro: la rotación del objeto (En este caso, **`Quaterion.identity`**)
En otras palabras, si **`i = 0`**, el cubo estará en (0,0,0); si **`i = 1`**, estará en (2,0,0); si **`i = 2`**, en (4,0,0), y así sucesivamente.	**`Quaternion`** representa la rotación en 3D en Unity, pero no usa ángulos directos como (0°, 90°, 180°) sino una forma matemática llamada cuaternión, que evita problemas como el "gimbal lock". **`Quaternion.identity`** significa sin rotación, o sea, que el objeto se genera con la orientación original del prefab.

3.3. Uso de Arrays: declarar, rellenar y recorrer

Un array es una lista de elementos del mismo tipo (por ejemplo, una lista de objetos del inventario del jugador). Esto permite guardar información y recorrerla con un ciclo **`for`**.

Un caso práctico es un sistema de inventario básico:

```
using UnityEngine;
using UnityEngine.UI;

public class RecolectorYInventario : MonoBehaviour
{
    public Text textoInventario;
    private string[] inventario = new string[5];
    private int maxItems = 5;
    private bool inventarioLleno = false;
    private int elementosEnInventario = 0;

    private void OnTriggerEnter(Collider other)
    {
        if (other.CompareTag("Item") && elementosEnInventario <
maxItems)
        {
            string nombreObjeto = other.gameObject.name;
            inventario[elementosEnInventario] = nombreObjeto;
            elementosEnInventario++;

            if (elementosEnInventario >= maxItems)
            {
                inventarioLleno = true;
            }

            ActualizarTextoInventario();
            Destroy(other.gameObject);

            if (inventarioLleno)
            {
                textoInventario.text += "\nInventario Lleno";
            }
        }
    }

    private void ActualizarTextoInventario()
    {
        textoInventario.text = "Inventario:\n";

        for (int i = 0; i < elementosEnInventario; i++)
        {
            textoInventario.text += "- " + inventario[i] + "\n";
        }
    }
}
```

Este sistema permite recoger hasta 5 objetos y mostrarlos en pantalla.

En mi canal de Youtube, tengo un tutorial sobre Arrays:
https://www.youtube.com/watch?feature=shared&v=JepSew72nlc

Este script, **`RecolectorYInventario`**, permite que un jugador recolecte objetos en la escena y los almacene en un inventario limitado a cinco elementos, mostrando la lista de objetos en pantalla a través de un componente UI (**`Text`**). El UI, o User Interface, en Unity se refiere a todos los elementos visuales con los que el jugador puede interactuar o recibir información, como textos, botones, barras de vida o menús; en este caso, **`textoInventario`** es un elemento de UI que muestra el inventario actualizado. Cuando el jugador entra en contacto con un objeto con etiqueta "Item", el script guarda el nombre del objeto en un array, lo destruye de la escena para simular que fue recogido, y actualiza el texto en pantalla con todos los objetos presentes en el inventario; si se alcanza el límite de cinco objetos, además se indica que el inventario está lleno.

Por lo tanto, para que este script funcione, tenés que tener en cuenta lo siguiente:

Crear y asignar etiquetas ("Tags")

1. Seleccioná el GameObject que quieras que sea recolectable (por ejemplo, un cubo que representa un item).
2. En el Inspector, busca la sección Tag (está más bien arriba).
3. Hacé clic en el desplegable y seleccioná Add Tag….
4. En la ventana que aparece, hacé clic en el **"+"** y escribe el nombre de la nueva etiqueta, por ejemplo: **`Item`**.
5. Volvé al GameObject y ahora sí selecciona la etiqueta **`Item`** que acabás de crear.

Esto es importante porque el script usa **`other.CompareTag("Item")`** para identificar qué objetos pueden ser recogidos.

Asignar el componente de UI

1. Andá a GameObject → UI → Text (Legacy) o TextMeshPro - Text si usás TextMeshPro
2. Se va a crear un objeto de UI en la escena con un Canvas. En Unity, un Canvas es un contenedor que organiza y dibuja todos los elementos de la interfaz de usuario (UI) en la pantalla, asegurando que textos, botones e imágenes se muestren correctamente frente a la cámara y se adapten a la resolución.
3. Arrastra este objeto de UI al campo público **`textoInventario`** del script **`RecolectorYInventario`** en el Inspector del jugador u objeto que tenga el script.

Esto asegura que el script pueda actualizar el texto en pantalla cada vez que se recolecta un objeto.

Configurar colisiones

- El jugador debe tener un Collider (por ejemplo, **`Capsule Collider`**) y un Rigidbody para que funcione **`OnTriggerEnter`**. Si da problemas, probar sin Rigidbody en el jugador o con Rigidbody Kinematic, y revisar capas y constraints.
- Los objetos "Item" también deben tener un Collider marcado como Is Trigger.
- Esto permite que Unity detecte la entrada en el trigger y ejecute el método **`OnTriggerEnter`**.

Otras cosas a tener en cuenta

- El array **`inventario`** tiene tamaño 5, que debe coincidir con **`maxItems`**. Si querés cambiar la cantidad máxima de ítems, cambia ambos valores.
- Los nombres de los objetos (**`other.gameObject.name`**) tiene que ser únicos si querés identificarlos claramente en el inventario.
- Si el UI no se actualiza correctamente, asegurate de que el Canvas tenga la opción Render Mode configurada en Screen Space - Overlay y que el texto sea legible.
- Si querés recolectar varios tipos de objetos, asegurate de que todos tengan la etiqueta correcta o modifica el script para aceptar varias etiquetas.

Resumen rápido:

1. Crear y asignar la etiqueta Item a los objetos que se puedan recolectar.
2. Crear un objeto UI Text y conectarlo al campo textoInventario.
3. Asegurarte de que el jugador y los objetos tengan colliders adecuados y que los objetos tengan Is Trigger activo.
4. Revisar que maxItems y el tamaño del array coincidan, y que los nombres de los objetos sean claros.

3.4. Recursos adicionales

En el repositorio de códigos de la cátedra van a encontrar ejemplos listos para usar, que pueden servir como base o "cajas negras" para explorar:

- Follow Player + Change Scene (seguir automáticamente al personaje y cambiar de escena):
  Link a los códigos en Google Drive: Código
  Link al Tutorial en YouTube: Tutorial
- Inventario + Recoger Monedas (gestionar inventario y sumar puntuación):
  Link a los códigos en Google Drive: Código
  Link al Tutorial en YouTube: Tutorial

Por otro lado, recuerden que este es el link con todo el repositorio de códigos: https://drive.google.com/drive/folders/1WbwNvS_4YMXC6N9FFgbrzvuhH4qiQ9D6?usp=sharing, y este es el link a mi canal de YouTube de Tutoriales: https://www.youtube.com/c/FacuSu%C3%A1rezTutoriales

4. Interacciones y feedback

Uno de los aspectos más importantes de un videojuego es cómo el jugador **interactúa** con él y cómo el juego responde a esas acciones. Esto incluye **detectar colisiones**, **usar triggers**, mostrar **información en pantalla** y proporcionar **retroalimentación sonora o visual** que haga que la acción sea satisfactoria.

4.1 Detectar colisiones y triggers

En Unity, para que los objetos "reaccionen" al contacto o al paso del jugador, se utilizan **Colliders** y **Rigidbody**:

- **Collider:** define la forma de un objeto para detectar colisiones. Puede ser de diferentes tipos: Box, Sphere, Capsule, etc.
- **Rigidbody:** hace que un objeto tenga física, permitiendo que Unity detecte colisiones correctamente.

Existen dos tipos de interacción:

1. **Colisiones físicas:** cuando dos objetos "chocan" realmente.
2. **Triggers:** cuando un objeto pasa por un área sin necesidad de chocar físicamente.

En Unity, normalmente usamos **triggers**, siendo estos, como decíamos, un tipo especial de **collider** que no produce una colisión física "real", sino que **detecta cuándo un objeto entra, sale o permanece dentro de un área determinada**. Es decir, **un trigger detecta presencia sin bloquear o empujar objetos**, y sirve para interacciones, pickups, checkpoints, zonas de daño, etc.

- Se configura marcando la casilla **"Is Trigger"** en un Collider (Box, Sphere, Capsule, etc.).
- Cuando un objeto con un Rigidbody entra en esa zona, se puede ejecutar código usando los métodos:
    - OnTriggerEnter(Collider other) → se activa al entrar
    - OnTriggerStay(Collider other) → se activa mientras permanezca dentro
    - OnTriggerExit(Collider other) → se activa al salir

Ejemplo de código para un trigger:

```
using UnityEngine;

public class PickUp : MonoBehaviour
{
    void OnTriggerEnter(Collider other)
    {
        if(other.CompareTag("Player"))
        {
            Debug.Log("¡Recogiste el objeto!");
            Destroy(gameObject); // El objeto desaparece al recogerlo
        }
    }
}
```

4.2 UI básica y feedback visual

La interfaz de usuario (UI) permite mostrar información al jugador: puntajes, vidas, tiempo restante, mensajes de victoria o derrota, entre otras. Para esto se crea un **Canvas** y se agregan elementos como **Text**, **Image** o **Button**.

Consejo práctico: cuando el jugador presiona un botón o recoge un objeto, usar animaciones, cambios de color o mensajes emergentes hace que la acción sea más satisfactoria.

4.3 Audio y efectos

El sonido potencia la experiencia: efectos cortos para acciones concretas, música de fondo para ambientación, o señales sonoras que indiquen logros o errores. En Unity se utiliza el componente **AudioSource** para reproducir clips y el **AudioClip** para almacenar los sonidos.

Ejemplo: reproducir un efecto al recoger un objeto:

```
public AudioSource audioSource;
public AudioClip pickupSound;

void PlayPickupSound()
{
    audioSource.PlayOneShot(pickupSound);
}
```

4.4 Principios de desarrollo y aprendizaje

Más allá de la técnica, es fundamental que **el videojuego empiece y termine**. Para los primeros proyectos, proponerse metas cortas evita la frustración y permite completar algo funcional:

- Trabajar en versiones cortas, completas y cerradas antes de agregar funcionalidades extra.
- Aprender a trabajar en equipo: cada integrante aporta sus fortalezas y permite aprender unos de otros.
- Testear el juego con personas externas al desarrollo: sus observaciones son más objetivas y permiten mejorar la experiencia.

Se puede comparar con la industria del cine: los cineastas a menudo comienzan haciendo cortometrajes antes de lanzarse a películas completas. Esto permite **aprender haciendo**, cometer errores en proyectos manejables y acumular experiencia. Lo mismo aplica a los videojuegos: se recomienda empezar haciendo un juego unitario, luego se puede expandir con más funciones y universos. Lo importante es terminar, aprender y luego mejorar.

4.5 Uso de ChatGPT para depuración y aprendizaje

La inteligencia artificial, y en particular herramientas como ChatGPT, llegaron para quedarse y representan un cambio significativo en la manera de aprender y enseñar programación. No se trata de reemplazar al programador, sino de agregar una herramienta más que nos ayude a crecer y avanzar en nuestro aprendizaje. Si te interesa profundizar en este tema, escribí un artículo al respecto: Programar en tiempos de inteligencia artificial: qué deberíamos enseñar.

El artículo plantea una pregunta fundamental: en un mundo donde la IA puede generar, optimizar e incluso depurar programas en segundos, ¿qué rol le queda al programador? La respuesta que proponemos en este curso es que **el rol del programar sigue y seguirá siendo valioso**, pero el enfoque debe estar en la **lógica, el diseño y la resolución de problemas**, más que en tipear líneas de código sin reflexión.

Por lo tanto, en caso de que quieras utilizar ChatGPT en esta etapa, te dejo algunos consejos:

1. **Aprender primero a programar sin IA:**
   Es fundamental que los estudiantes entiendan cómo funciona el código por sí mismos, para no volverse como los personajes de la película *WALL·E*, que dependen completamente de la tecnología sin comprenderla.

2. **Usarlo como asistente pedagógico:**
   La mejor forma de aprovechar ChatGPT es hacerle preguntas sobre el código que estás escribiendo:
   - ¿Qué hace esta línea de código?
   - ¿Por qué usamos este método en este caso?
   - ¿Qué errores podría generar este fragmento y cómo depurarlo?

   De esta manera, ChatGPT **no reemplaza tu aprendizaje**, sino que te ayuda a **entender profundamente cada parte del código**, explicando su funcionamiento paso a paso.

3. **No copiar y pegar sin comprensión:**
   En esta etapa de formación es más importante **entender el código que escribirlo rápido**. Usar ChatGPT como un "copiador" puede dar resultados inmediatos, pero no desarrolla pensamiento algorítmico ni habilidades para resolver problemas.

4. **Aprender a depurar con ayuda:**
   ChatGPT puede sugerir estrategias para identificar errores y optimizar fragmentos de código. Esto te permite **probar soluciones y aprender de los errores**, mientras manteés el control sobre lo que realmente hace tu programa.

ChatGPT y herramientas similares puede ser **aliados potentes** si se usan con criterio: pueden acelerar la comprensión, ayudar a depurar y clarificar la lógica de programación, **pero nunca van a reemplazar el aprendizaje tradicional de programar "a mano"**. Aprender a pensar y comprender el código es la base que va a sotener toda tu práctica profesional futura.

6. Cómo colocar los archivos en Google Drive, checklist de entrega y documentación

Cuando finalizamos un proyecto de videojuego, es fundamental **organizar y entregar los archivos de manera clara**. Esto no solo facilita la evaluación, sino que también permite que otras personas puedan abrir, estudiar y aprender de nuestro trabajo.

Exportar un videojuego en Unity

Cuando finalizas un juego en Unity, hay dos formas principales de "exportarlo": **el proyecto editable** y el **ejecutable (build)**. Cada uno cumple un propósito diferente:

Proyecto editable

El **proyecto editable** es la carpeta completa de Unity donde se encuentran todos los archivos del juego:

- **Assets:** contiene todos los modelos 3D, sprites, sonidos, scripts y demás recursos.
- **Project Settings:** configuraciones del proyecto (gráficos, físicas, controles, etc.).
- **Packages:** dependencias y librerías que utiliza el proyecto.

Características:

- Permite **seguir editando** el juego en Unity.
- Se puede abrir en otra computadora con Unity instalado (ideal para trabajo en equipo o para continuar desarrollando).
- No es un juego que se pueda ejecutar directamente por un usuario que no tenga Unity.

Cómo compartirlo: normalmente se comprime toda la carpeta del proyecto (.zip) y se sube a Google Drive o GitHub.

Ejecutable (Build)

El **ejecutable** es un archivo que se puede abrir y jugar sin necesidad de Unity.

- Unity genera un **build** específico según la plataforma elegida: Windows (.exe), Mac (.app), Linux (.x86_64), Android (.apk), etc.
- Incluye únicamente lo necesario para **jugar**, no para editar el proyecto.
- Suele generar además algunas carpetas de soporte donde Unity coloca los assets y librerías necesarias.

Cómo generar un build en Unity:

1. **Abrir Build Settings:** File → Build Settings.
2. **Agregar escenas:** asegúrate de que todas las escenas que quieras incluir estén en la lista **Scenes in Build**. La primera escena será la que se cargue al iniciar.
3. **Seleccionar la plataforma:** Windows, Mac, Linux, Android, etc., y hacer clic en **Switch Platform** si es necesario.
4. **Configurar Player Settings:** ícono, resolución, orientación de pantalla (para móviles), nombre del producto, etc.
5. **Build o Build and Run:**
    - **Build:** Unity genera los archivos en la carpeta que elijas.
    - **Build and Run:** además de generar los archivos, Unity abre automáticamente el juego para probarlo.

6.1 Subir archivos a Google Drive en modo público

Google Drive permite almacenar archivos y compartirlos mediante un **link público**, lo que facilita la entrega y el acceso:

1. Ingresá a Google Drive y creá una carpeta nueva con el nombre de tu proyecto.
2. Dentro de la carpeta, subí todos los archivos necesarios:
    - Carpeta **Build** (ejecutable y archivos necesarios para jugar).
    - Carpeta **Assets** o proyecto de Unity completo.
    - Documentación y capturas.
3. Hacé clic derecho sobre la carpeta → **Compartir → Cambiar a cualquier persona con el enlace → Lector**.
4. Copia el enlace y envíalo según las instrucciones de entrega.

Consejo práctico: Mantener una estructura clara de carpetas ayuda a que cualquier persona pueda abrir y testear tu proyecto sin problemas. De ser necesario, crea una cuentan nueva desde cero para tener el espacio suficiente para poder subir todos los archivos.

6.2 Checklist de entrega

Antes de enviar el proyecto, verifica que:

- Todas las escenas importantes están incluidas en el build.
- Los assets necesarios están dentro de la carpeta del proyecto.
- Se incluyen capturas de pantalla o videos cortos del juego funcionando.
- La documentación está completa y con secciones claras.
- El build funciona correctamente en la plataforma seleccionada.
- Se probó el proyecto con alguien que no haya trabajado en él para obtener feedback objetivo.

6.3 Plantilla de documentación

Un documento bien organizado permite que otros **aprendan de tu proyecto** y que el trabajo quede para la posteridad. Se recomienda incluir estas secciones:

1. **Descripción del proyecto:** objetivo, mecánicas principales, público al que va dirigido.
2. **Cómo se hizo:** herramientas utilizadas, metodología, decisiones de diseño.
3. **Códigos usados:** fragmentos de código clave con explicación de su función.
4. **Capturas de imágenes:** imágenes que muestren el videojuego en acción.
5. **Habilidades desarrolladas:** qué aprendiste y qué técnicas utilizaste.
6. **Videos (Gameplay):** videos que muestren el videojuego en acción.
7. **Video a modo de bitácora:** videos que expliquen cómo fue desarrollado el proyecto.

6.4 GitHub y la filosofía open source

Otra opción para almacenar y compartir proyectos es **GitHub**, una plataforma de control de versiones que permite:

- Subir proyectos completos de manera organizada.
- Mantener un historial de cambios (commits).
- Colaborar con otras personas de forma simultánea.
- Compartir tu código con la comunidad, fomentando el aprendizaje colectivo.

Importancia del open source en el ámbito académico

En la universidad, especialmente en proyectos de investigación y formación, es muy valioso que los proyectos sean **abiertos y accesibles**. Esto permite que tus colegas aprendan de tu trabajo, repliquen experimentos, prueben mejoras o utilicen tu código como referencia. Más adelante, en la industria privada o en proyectos comerciales, las reglas pueden cambiar, pero durante la formación académica es fundamental **priorizar la colaboración y la documentación clara**.

Recordá que al desarrollar videojuegos en Unity, es fundamental mantener **buenas prácticas de nomenclatura**, como usar nombres claros y consistentes para objetos, scripts y variables, por ejemplo **`PlayerHealth`** en lugar de **`ph`** o **`Objeto1`**, y emplear mayúsculas para distinguir clases (**`PlayerController`**) y minúsculas con camelCase para variables y métodos (**`currentScore`**). Los **comentarios en el código** deben ser útiles y explicativos, indicando qué hace cada bloque o función y, cuando sea relevante, por qué se tomó cierta decisión de diseño, evitando comentarios triviales como "***`incrementa contador`***" sin aclarar más nada.

Consejo práctico: aunque subas el proyecto a GitHub o Google Drive, incluí siempre la documentación completa y comentarios claros en tu código. Esto va a asegurar que tu proyecto pueda ser entendido y utilizado por otros, cumpliendo con la filosofía de aprendizaje compartido.

6.5 Lista de verificación para entrega

Antes de enviar tu proyecto, revisa que **todo esté completo y organizado**. Esta lista sirve como guía rápida:

- Carpeta del proyecto de Unity completa, con todas las escenas y assets necesarios.
- Build ejecutable funcional para la plataforma correspondiente.
- Carpeta con capturas de pantalla o videos demostrativos del juego.
- Documentación completa con las secciones: Descripción, Cómo se hizo, Códigos usados, Capturas, Habilidades desarrolladas.
- Código comentado y limpio, siguiendo buenas prácticas de nomenclatura y estructura.
- Enlace de Google Drive público correctamente configurado, o repositorio de GitHub accesible.
- Confirmación de que alguien externo al proyecto ha probado el juego y ha dado feedback.
- Versiones de Unity y paquetes utilizados documentados, para reproducibilidad.
- Breve nota de resumen explicando el estado del proyecto y posibles mejoras futuras.

Consejo práctico: usar esta lista evita olvidos, facilita la evaluación y asegura que cualquier compañero o docente pueda abrir y comprender tu proyecto sin dificultades. Además, refuerza la práctica de **terminar proyectos completos**, aunque sean cortos, y dejar un registro claro de todo lo trabajado.

Algunas recomendaciones para el trabajo en Unity

Como anticipé anteriormente, personalmente no tengo problema en que revisen foros o tutoriales de YouTube para realizar los proyectos. De hecho, yo mismo mantengo un repositorio de código que dejo a disposición del alumnado, disponible en el siguiente enlace: https://drive.google.com/drive/folders/1ybhj7GRP903PqJRfi-Vkl3R7EPBo-nVO

Sin embargo, mi principal recomendación es que, sea cual sea el camino o la herramienta que elijan, **entiendan profundamente lo que están haciendo**. Incluso si deciden utilizar inteligencia artificial.

Sea que optemos por usar IA o no, es fundamental comprender **qué estamos haciendo cuando la usamos**. Debemos ser capaces de mantener el control sobre la herramienta, y no permitir que sea ella quien decida por nosotros. Esto no solo responde a una cuestión de uso responsable, sino también a la necesidad de **preservar nuestra capacidad expresiva** dentro del proceso creativo. En nuestro caso, esa expresión se materializa en los videojuegos que elegimos hacer. De lo contrario, corremos el riesgo de que nuestras obras terminen moldeadas por los sesgos, las limitaciones y los intereses que subyacen a los sistemas de IA. No debemos olvidar que estas tecnologías se alimentan del pasado de la humanidad, y que detrás de sus algoritmos y modelos siempre hay personas e instituciones con intenciones concretas.

Por ese motivo, si deciden recurrir a herramientas como ChatGPT, les sugiero que lo hagan **para comprender**, no simplemente para resolver. Que pidan explicaciones, que indaguen sobre cómo y por qué un código funciona de determinada manera, y que utilicen la inteligencia artificial como **una aliada para pensar**, no como un sustituto del pensamiento propio.

Del mismo modo, **no recomiendo utilizar paquetes del Asset Store de Unity que incluyan código prearmado o librerías cerradas**. Estos paquetes, aunque prácticos, funcionan muchas veces como una *caja negra*: realizan acciones que desconocemos en detalle, y nos alejan del entendimiento real de lo que ocurre dentro del juego. En etapas de formación, es preferible evitar estos atajos y concentrarse en la construcción de código propio o en ejemplos didácticos.

Para ello, recomiendo consultar mi canal de YouTube, donde explico desde cero cada ejemplo y analizo el funcionamiento de los scripts paso a paso. Allí podrán encontrar recursos pensados para que comprendan la lógica detrás de cada línea de código, en lugar de limitarse a reproducirla.

Finalmente, para complementar cualquier trabajo técnico, siempre es recomendable recurrir a la **documentación oficial de Unity**, que constituye una fuente confiable, actualizada y de libre acceso: https://docs.unity.com/en-us

Por último, un aspecto que considero fundamental al encarar sus proyectos es el **alcance o *scope***. Este término, muy utilizado tanto en el ámbito del desarrollo de videojuegos como en la gestión de proyectos, hace referencia al tamaño y la complejidad de lo que nos proponemos realizar dentro de un tiempo determinado. Si sienten que la idea que están desarrollando es demasiado grande para el tiempo disponible, **ajusten el scope**: no se trata de renunciar a la ambición, sino de planificar con inteligencia. En la práctica, esto implica **dividir el proyecto en partes más pequeñas**, manejables y realizables, que puedan completarse de manera progresiva.

Así trabajamos no solo en esta materia, sino también en la industria, donde los **plazos de entrega (*deadlines*)** son una realidad constante. Lo importante es que cada etapa del desarrollo tenga un **resultado concreto y exportable**, aunque sea parcial, que puedan presentar y utilizar como base para seguir avanzando.

Por ejemplo, si su objetivo final es desarrollar un juego de plataformas con varios niveles, enemigos, inventario y sistema de puntuación, podrían comenzar en fases:

1. **Versión 1:** Implementar el movimiento del personaje y el control básico.
2. **Versión 2:** Agregar colisiones y detección de obstáculos.
3. **Versión 3:** Incorporar enemigos y condiciones de victoria o derrota.
4. **Versión 4:** Sumar interfaz gráfica, menús y efectos de sonido.

De esta manera, cada entrega constituye un **proyecto funcional en sí mismo**, que se puede ejecutar y evaluar, mientras avanzan hacia una versión más completa. Pensar el desarrollo de esta forma modular no solo facilita la organización del trabajo, sino que también enseña una práctica profesional real: **entregar versiones jugables, medibles y mejorables con el tiempo**.

Por último, recuerden que no les recomiendo depender de la nube de Unity para guardar los proyectos. Es mejor ir haciendo respaldos de manera constante en pendrives, discos externos y/o Google Drive, asegurándose de tener al menos dos versiones de backup del proyecto en todo momento. Al trabajar en grupo, aprovechen las ventajas que ofrece la programación orientada a objetos: cada integrante puede concentrarse en un código específico, en un objeto determinado e incluso en distintas escenas, lo que permite avanzar de manera más organizada y eficiente sin interferir en el trabajo de los demás.

Comentarios Finales

Les doy un consejo sincero: están en la universidad, y eso implica un compromiso distinto. No alcanza solo con venir a clase. Tienen que leer, buscar, investigar por su cuenta. Lo digo con total claridad: si esperan que todo se los dé yo o cualquier docente, se están quedando cortos. Este es un campo que cambia todo el tiempo. Muchas de las herramientas que uso hoy ni existían cuando estudiaba, y lo mismo les va a pasar a ustedes. Por eso, aprender a autoformarse no es opcional: es necesario. Los tutoriales de mi canal de YouTube no reemplazan el estudio, sino que lo complementan. Están para que, si quizás no entendieron la explicación en el momento, tengan la chance de "volver al pasado" y volver a escuchar nuevamente esa explicación. Úsenlos como complemento. Por otro lado, les recomiendo que se acostumbren desde ahora a leer libros (Y sobre este punto hago énfasis en que leer libros sigue siendo clave, aunque a veces parezca pasado de moda. Hay conocimientos y perspectivas que solo van a encontrar en un buen texto leído con tiempo y atención. No se conformen con un resumen, con una explicación rápida, o con lo que se dijo en clase), ver tutoriales de distintas fuentes, participar en foros y ayudarse entre ustedes en espacios como el grupo de WhatsApp, porque este es un campo artístico y tecnológico que cambia todo el tiempo, y es esencial saber trabajar en equipo y buscar por cuenta propia. Para aprender de verdad, van a tener que ir más allá de lo que vemos en clase.

La diferencia no la hace quién sabe más, sino quién sabe aprender mejor y no se queda quieto.

Facu

Bibliografía que recomiendo consultar y que fue consultada para la elaboración de este texto

- Suárez, F. N. (2021). *Kudewe: El audiojuego como herramienta educativa* [Trabajo final de grado, Universidad Nacional de Tres de Febrero]. https://arteselectronicas.untref.edu.ar/uploads/pdf/1655762003.pdf
- Avolio, M. C. (2024). *¿Qué es la inteligencia artificial?* Grijalbo.
- Ceballos Sierra, F. (2011). Capítulo 1 - Fases en el desarrollo de un programa. En *Microsoft C#: Curso de programación* (2ª ed., pp. 3-18). Madrid, España: RA-MA Editorial.
- Ceballos Sierra, F. (2011). Capítulo 2 - Introducción a C#. En *Microsoft C#: Curso de programación* (2ª ed., pp. 19-32). Madrid, España: RA-MA Editorial.
- Ceballos Sierra, F. (2011). Capítulo 4 - Elementos del lenguaje. En *Microsoft C#: Curso de programación* (2ª ed., pp. 61-82). Madrid, España: RA-MA Editorial.
- Ceballos Sierra, F. (2011). Capítulo 5 - Estructura de un programa. En *Microsoft C#: Curso de programación* (2ª ed., pp. 83-108). Madrid, España: RA-MA Editorial.
- Ceballos Sierra, F. (2011). Capítulo 7 - Sentencias de control. En *Microsoft C#: Curso de programación* (2ª ed., pp. 3-18). Madrid, España: RA-MA Editorial.
- Ceballos Sierra, F. (2011). Capítulo 8 - Matrices. En *Microsoft C#: Curso de programación* (2ª ed., pp. 177-224). Madrid, España: RA-MA Editorial.
- Joyanes Aguilar, L. (2020). Capítulo 1 - Introducción a las computadoras y a los lenguajes de programación. En *Fundamentos de programación: Algoritmos, estructura de datos y objetos* (5ª ed., pp. 3-40). Ciudad de México, México: McGraw-Hill Interamericana.
- Joyanes Aguilar, L. (2020). Capítulo 2 - Metodología de la programación y desarrollo de software de programación. En *Fundamentos de programación: Algoritmos, estructura de datos y objetos* (5ª ed., pp. 41-80). Ciudad de México, México: McGraw-Hill Interamericana.
- Joyanes Aguilar, L. (2020). Capítulo 3 - Estructura general de un programa. En *Fundamentos de programación: Algoritmos, estructura de datos y objetos* (5ª ed., pp. 83-130). Ciudad de México, México: McGraw-Hill Interamericana.
- Joyanes Aguilar, L. (2020). Capítulo 4 - Flujo de control I: estructuras selectivas. En *Fundamentos de programación: Algoritmos, estructura de datos y objetos* (5ª ed., pp. 133-161). Ciudad de México, México: McGraw-Hill Interamericana.
- Joyanes Aguilar, L. (2020). Capítulo 5 - Flujo de control II: estructuras repetitivas. En *Fundamentos de programación: Algoritmos, estructura de datos y objetos* (5ª ed., pp. 163-208). Ciudad de México, México: McGraw-Hill Interamericana.
- Joyanes Aguilar, L. (2020). Capítulo 6 - Subprogramas (subalgoritmos): funciones. En *Fundamentos de programación: Algoritmos, estructura de datos y objetos* (5ª ed., pp. 209-251). Ciudad de México, México: McGraw-Hill Interamericana.
- Joyanes Aguilar, L. (2020). Capítulo 7 - Estructuras de datos I (arrays - arreglos - y estructuras). En *Fundamentos de programación: Algoritmos, estructura de datos y objetos* (5ª ed., pp. 255-291). Ciudad de México, México: McGraw-Hill Interamericana.
- Joyanes Aguilar, L. (2020). Capítulo 8 - Las cadenas de caracteres. En *Fundamentos de programación: Algoritmos, estructura de datos y objetos* (5ª ed., pp. 293-314). Ciudad de México, México: McGraw-Hill Interamericana.

- Albahari, J. (2022). *C# 10 in a Nutshell: The Definitive Reference*. California, USA: O'Reilly Media, Inc.
- Albahari, J., & Albahari, B. (2022). *C# 10 Pocket Reference: Instant Help for C# 10 Programmers*. California, USA: O'Reilly Media, Inc.