Tema 1: Introducción a la Programación

La programación informática es una de las habilidades más valiosas hoy en día. Vivimos rodeados de software: el sistema de control de un coche, las plataformas de mensajería, las tiendas online o los programas que ayudan a diseñar medicamentos.
Todos ellos existen porque, en algún momento, alguien tradujo un problema humano en una **secuencia de instrucciones comprensibles por un ordenador**. Esa traducción es, en esencia, **programar**.

1. El ordenador y el sistema operativo

Antes de aprender a programar conviene responder a una pregunta básica: **¿qué es realmente un ordenador?**
Un ordenador no es una "caja mágica": es una máquina formada por componentes electrónicos capaces de **recibir datos, procesarlos siguiendo instrucciones y generar resultados**.

1.1 Hardware y software

Podemos imaginar el ordenador en dos grandes dimensiones:

- **Hardware**: la parte física y tangible → procesador, memoria, disco duro, pantalla, teclado, etc.
- **Software**: la parte lógica → programas e instrucciones que hacen que el hardware cobre sentido.

El hardware, por sí solo, apenas sabe hacer operaciones eléctricas simples. Es el software el que convierte esa máquina en una herramienta útil, indicándole **qué debe hacer, en qué orden y con qué datos**.

1.2 El papel del sistema operativo

Dentro del software, el componente central es el **sistema operativo (SO)**. Podemos verlo como un **intérprete y coordinador**: traduce lo que pedimos en acciones que el hardware entiende, y organiza todos los recursos del ordenador para que no se estorben entre sí.

Sus funciones principales incluyen:

- **Gestión de procesos** → decide qué programa se ejecuta y cuándo.
- **Gestión de memoria** → asigna espacio en la RAM a cada aplicación.
- **Gestión de dispositivos** → controla periféricos como teclado, impresora, cámara o tarjeta gráfica.
- **Sistema de archivos** → organiza la información en carpetas y documentos.
- **Interfaz de usuario** → nos permite comunicarnos con la máquina, ya sea con una **interfaz gráfica** (ventanas, iconos, ratón) o con una **interfaz de texto** (la terminal).

Sin sistema operativo, cada aplicación tendría que hablar directamente con el hardware. Esto sería caótico y casi imposible de mantener en sistemas complejos.
Gracias al SO, los programadores podemos centrarnos en la lógica de nuestros programas, confiando en que las tareas básicas de gestión ya están resueltas.

1.3 Tipos de sistemas operativos

Existen distintas familias de sistemas operativos, cada una con su historia y filosofía:

- **Windows** → muy usado en entornos personales y empresariales, con gran compatibilidad de programas.
- **macOS** → creado por Apple, destaca en diseño gráfico, edición multimedia y entornos creativos.
- **Linux** → software libre y de código abierto, muy presente en servidores, investigación científica y desarrolladores de software.

Aunque cambien en apariencia, distribución o licencias, **todos comparten la misma misión**: ofrecer una base estable y coherente para ejecutar programas, y facilitar que el usuario pueda trabajar sin preocuparse por el funcionamiento interno del hardware.

1.4 El sistema operativo y la programación

Para un programador, comprender el papel del sistema operativo es esencial. Cuando escribimos un programa, en realidad **le estamos pidiendo al SO que haga cosas por nosotros**: abrir un archivo, mostrar un texto en pantalla, guardar datos o enviar información por internet.

Algunos ejemplos sencillos:

- Al escribir un documento en un procesador de textos, **no es el programa el que controla el disco duro directamente**: es el sistema operativo quien se encarga de guardar el archivo.
- En un videojuego, **no es el código el que pinta cada píxel en la pantalla**: el programa pide al sistema operativo que use la tarjeta gráfica para dibujar la imagen.

En resumen: cada instrucción de un programa **pasa antes por el sistema operativo** para llegar hasta el hardware.

En esta asignatura aprenderemos a usar la **terminal**, una herramienta directa que nos permite comunicarnos con el sistema operativo mediante comandos.

2. Programas y aplicaciones

Después de entender el papel del sistema operativo como coordinador de recursos, toca responder otra pregunta fundamental: **¿qué es un programa y cómo se convierte en una aplicación que usamos a diario?**

En esencia:

- Un **programa** es la realización práctica de un **algoritmo**: una secuencia de instrucciones que transforman datos de entrada en resultados útiles.
- Una **aplicación** es ese programa empaquetado en un contexto concreto, con su interfaz, su forma de distribución y su interacción con el resto del sistema.

Ambos tienen el mismo origen, pero la aplicación está diseñada para que **personas o sistemas** la utilicen de manera sencilla y práctica.

2.1 Concepto y propósito

Un **programa** es un conjunto finito y ordenado de instrucciones, escritas en un lenguaje de programación, para resolver una tarea específica.

Una **aplicación** es ese programa orientado a un caso de uso concreto: está **empaquetada y presentada** para que usuarios o sistemas puedan utilizarla fácilmente.

Independientemente de su tamaño o complejidad, toda aplicación tiene un propósito claro:

- **automatizar** tareas,
- **asistir** en un trabajo,
- **comunicar** personas o dispositivos,
- o **calcular** con fiabilidad y repetición.

En resumen: programar consiste en diseñar instrucciones precisas; crear aplicaciones consiste en darles forma para que tengan **utilidad real**. Sería como seguir una receta de cocina, programar sería crear la receta de cocina y el ordenador la seguirá paso a paso para recrear el plato.

2.2 Componentes esenciales de una aplicación

Aunque existen aplicaciones muy diferentes entre sí, la mayoría comparten cuatro elementos básicos:

1. **Entrada** → los datos que recibe (por teclado, archivos, red, sensores, etc.).
2. **Procesamiento** → las reglas y operaciones que transforman esa entrada (cálculos, validaciones, decisiones).
3. **Salida** → los resultados que genera (texto en pantalla, un archivo guardado, una respuesta por red, una señal enviada a un dispositivo).
4. **Estado** → la información que conserva más allá de una ejecución (preferencias, historiales, bases de datos).

Un aspecto clave del diseño es el **manejo de errores**.
Una aplicación bien hecha no solo funciona cuando todo va perfecto, sino que también sabe reaccionar ante lo imprevisto sin perder seguridad ni fiabilidad.

2.3 Tipos de aplicaciones (visión general)

Las aplicaciones pueden tomar muchas formas, pero se suelen clasificar en varios tipos principales:

- **De línea de comandos (CLI)** → se manejan escribiendo texto en la terminal. Son precisas, rápidas y fáciles de automatizar. Muy usadas en entornos técnicos.
- **Gráficas (GUI)** → usan ventanas, botones y menús. Están pensadas para ser intuitivas y fáciles de usar para el público general.
- **Servicios o demonios** → se ejecutan en segundo plano y ofrecen funcionalidades a otras aplicaciones (por ejemplo, un servidor web o un servicio de impresión).
- **De escritorio y móviles** → diseñadas para sistemas concretos (Windows, macOS, Android, iOS). Se distribuyen en tiendas de apps o gestores de paquetes.
- **Web** → se utilizan desde un navegador. Normalmente tienen una parte en el cliente (interfaz) y otra en el servidor (lógica y datos).
- **Empotradas** → funcionan en dispositivos con recursos limitados, como electrodomésticos, robots o sistemas IoT, donde la fiabilidad y la respuesta en tiempo real son fundamentales.

2.4 Criterios de calidad de una aplicación

No basta con que una aplicación funcione: para considerarse profesional debe cumplir ciertos criterios de calidad. Entre los más importantes están:

- **Corrección** → hace lo que debe en todos los casos previstos.
- **Eficiencia** → aprovecha bien los recursos (tiempo de ejecución, memoria, red).
- **Usabilidad y accesibilidad** → resulta comprensible y fácil de usar para distintos públicos, incluidas personas con necesidades especiales.
- **Mantenibilidad** → su código se entiende, se prueba y se modifica sin excesiva dificultad.
- **Seguridad** → protege los datos y resiste intentos de abuso o fallos inesperados.
- **Observabilidad** → ofrece registros y métricas que permiten analizar su comportamiento en producción.

Estos criterios son los que diferencian un simple programa casero de una aplicación robusta y confiable.

2.5 Lenguajes de programación

Para comunicarnos con un ordenador necesitamos un **lenguaje que entienda tanto la persona como la máquina**.
Estos son los **lenguajes de programación**: conjuntos de reglas y símbolos que nos permiten escribir instrucciones de forma estructurada.

Podemos distinguirlos en varios niveles:

- **Lenguaje máquina** → formado solo por ceros y unos. Es el único que entiende directamente el procesador, pero resulta casi imposible de usar por humanos.
- **Lenguajes de bajo nivel** → más cercanos al hardware, como el *ensamblador*. Requieren conocer los detalles internos del ordenador.
- **Lenguajes de alto nivel** → más cercanos al pensamiento humano, con palabras y estructuras fáciles de leer (por ejemplo, Python, Java, C++).

Un **compilador** o un **intérprete** traduce estos lenguajes de alto nivel a instrucciones que el ordenador puede ejecutar.

2.6 Compiladores e intérpretes

Cuando escribimos un programa en un lenguaje de alto nivel, el ordenador **no lo entiende directamente**. Necesitamos una herramienta que lo traduzca a instrucciones que la máquina pueda ejecutar.

Existen dos enfoques principales:

- **Compiladores** → traducen **todo el programa completo** antes de ejecutarlo.
  - Generan un archivo ejecutable.
  - Suelen ser más rápidos en la ejecución.
  - Ejemplos: C, C++, Java (con compilación intermedia a bytecode).
- **Intérpretes** → traducen el programa **línea a línea mientras se ejecuta**.
  - No generan un ejecutable independiente.
  - Son más flexibles y fáciles de probar en tiempo real.
  - Ejemplos: Python, JavaScript.

Ambos enfoques tienen ventajas y desventajas:

- Compilar da velocidad y optimización.
- Interpretar da rapidez en el desarrollo y facilidad de depuración.

Muchos lenguajes modernos combinan ambas técnicas (por ejemplo, Java usa un compilador a *bytecode* y después una máquina virtual que interpreta y optimiza sobre la marcha).

3. Diagramas de flujo

Cuando empezamos a programar, uno de los mayores retos es **aprender a pensar en pasos ordenados**. Antes de escribir código en un lenguaje concreto, conviene representar las ideas de manera visual.
Ahí es donde entran los **diagramas de flujo**: representaciones gráficas que muestran, con símbolos y flechas, cómo fluye la información y las decisiones dentro de un algoritmo.

¿Por qué son útiles para aprender a programar?

- **Visualizan la lógica** → ayudan a ver el camino que siguen los datos paso a paso.
- **Facilitan la detección de errores** → es más sencillo encontrar pasos mal planteados en un dibujo que en un bloque de código.
- **Son independientes del lenguaje** → un mismo diagrama puede traducirse luego a Python, Java o cualquier otro lenguaje.
- **Refuerzan el pensamiento algorítmico** → acostumbran a dividir los problemas en operaciones pequeñas y ordenadas.

3.1 Símbolos básicos de un diagrama de flujo

Para leer y construir diagramas de flujo necesitamos un **lenguaje visual estándar**. Este lenguaje está formado por símbolos que representan las distintas acciones de un algoritmo.

Símbolo	Nombre	Función
	Inicio / Final	Representa el inicio y el final de un proceso
	Linea de Flujo	Indica el orden de la ejecución de las operaciones. La flecha indica la siguiente instrucción.
	Entrada / Salida	Representa la lectura de datos en la entrada y la impresión de datos en la salida
	Proceso	Representa cualquier tipo de operación
	Decisión	Nos permite analizar una situación, con base en los valores verdadero y falso

3.2 Puertas lógicas y decisiones

Cuando programamos, muchas veces necesitamos que el ordenador tome decisiones.
Estas decisiones se basan en condiciones lógicas que solo pueden dar dos resultados:

- Verdadero (true)
- Falso (false)

A estos valores se les llama booleanos (por George Boole, matemático que desarrolló el álgebra lógica en el siglo XIX). Para combinar y evaluar condiciones, usamos las operaciones lógicas o puertas lógicas:

- AND (Y lógico) → una condición solo es verdadera si todas las partes lo son.
- OR (O lógico) → basta con que una sola de las partes sea verdadera para que el resultado lo sea.
- NOT (NO lógico) → invierte el resultado de una condición (si es verdadero se convierte en falso, y viceversa).

Estas operaciones permiten expresar reglas de decisión en un programa.
Por ejemplo:

- “El alumno aprueba si la nota es mayor o igual que 5 AND ha entregado las prácticas.”
- “El acceso es válido si la contraseña es correcta OR se usa un código de recuperación.”
- “Mostrar ‘Suspendido’ si NOT aprobado”

En programación, cada vez que usamos una instrucción condicional, en el fondo estamos evaluando expresiones booleanas que combinan estas operaciones.

3.4 Operadores y variables

En programación no solo tomamos decisiones: también necesitamos **guardar datos** y **operar con ellos**.
Una **variable** es como una "caja" con un nombre donde podemos guardar información para usarla más tarde.

- Cada variable tiene un **nombre** (ejemplo: `edad`, `nota`, `saldo`).
- Contiene un **valor**, que puede cambiar durante la ejecución del programa.

Los **operadores** son los símbolos que permiten hacer operaciones con los valores de las variables. Podemos agruparlos en tres familias principales:

- **Aritméticos** → trabajan con números
  - Suma (`+`), resta (`-`), multiplicación (`*`), división (`/`), resto (`%`).
- **Relacionales** → comparan valores
  - Igualdad (`==`), distinto (`!=`), mayor (`>`), menor (`<`), mayor o igual (`>=`), menor o igual (`<=`).
  - El resultado siempre es **verdadero o falso**.
- **Lógicos** → combinan condiciones booleanas
  - AND (`&&` o `AND`), OR (`||` o `OR`), NOT (`!` o `NOT`).
  - Sirven para construir decisiones más complejas.

En un diagrama de flujo, las **variables** aparecen cuando guardamos datos de entrada o resultados de operaciones.
Los **operadores** aparecen en los procesos (cálculos, sumas, divisiones) y en las decisiones (comparaciones, condiciones lógicas).

3.4 Problemas para practicar diagramas de flujo

1. Leer un número y muestra si es **positivo** o **negativo**.
2. Leer dos números, calcula y muestra su **suma**.
3. Leer dos números y mostrar cuál es el **mayor**.
4. Leer un número y mostrar si es **par o impar**.
5. Leer la edad de una persona y decidir si es **mayor o menor de edad**.
6. Leer un número y comprobar si está dentro del rango **0 a 100**.
7. Leer la nota de un examen y decidir si está suspenso, aprobado, notable o sobresaliente.
8. Calcular el **área de un rectángulo** a partir de base y altura.
9. Leer tres números y decidir cuál es el **menor**.
10. Calcular el **promedio de tres notas** y decidir si está aprobado (promedio ≥ 5).
11. Haz el ejercicio 5 añadiendo una validación para que la edad no pueda ser negativa, se tendrá que verificar constantemente para que el programa finalice con una edad válida.
12. Añade validación independiente a cada nota en el ejercicio 10.
13. Leer 3 números distintos y ordenarlos de menor a mayor.
14. Leer un número y decidir si es primo o no.
15. Calcular el factorial de un número entero positivo.
16. Calcular la potencia de un número dados la base y el exponente.
17. Leer un número e indicar la tabla de multiplicar de ese número del 1 al 10.
18. Leer una lista de N números, N será una entrada inicial del usuario, luego leerá N números y calculará cuántos son positivos, negativos y ceros.
19. Calcular el promedio de los números pares en un conjunto de N números.