

PROMETEO

Unidad 1.

Introducción al desarrollo de software y lenguajes

Cuando piensas en un ordenador, probablemente lo primero que imaginas son sus piezas físicas: la pantalla, el teclado, la memoria o el procesador. Esa parte tangible recibe el nombre de **hardware**. Sin embargo, lo que realmente da vida a la máquina y hace posible que obedezca nuestras órdenes es el **software**.

Sesión 1: ¿Qué es el software y cómo se crea?

Cuando piensas en un ordenador, probablemente lo primero que imaginas son sus piezas físicas: la pantalla, el teclado, la memoria o el procesador. Esa parte tangible recibe el nombre de **hardware**. Sin embargo, lo que realmente da vida a la máquina y hace posible que obedezca nuestras órdenes es el **software**. Podríamos decir que el hardware es el cuerpo, mientras que el software es la mente.

Hardware

La parte tangible de un ordenador: la pantalla, el teclado, la memoria o el procesador. El "cuerpo" de la máquina.

Software

Lo que da vida a la máquina y hace posible que obedezca nuestras órdenes. La "mente" que gestiona y permite su funcionamiento.

Sin software, un ordenador es tan útil como una calculadora vacía; con software, se convierte en una herramienta capaz de gestionar una hoja de cálculo, ejecutar un videojuego o enviar un correo electrónico.

¿Qué es el software?

El software se define como el conjunto de programas, instrucciones y datos que permiten que un dispositivo electrónico funcione. Su carácter intangible es clave: no se toca, se ejecuta. Aun así, es tan indispensable como la electricidad que recorre los circuitos. El software no solo dicta qué operaciones realizar, sino también **cómo** y **en qué orden** llevarlas a cabo.

Un ejemplo cotidiano: Tu móvil

Un ejemplo cotidiano lo tienes en el móvil que usas a diario. El sistema operativo (Android o iOS) gestiona recursos como memoria y batería. Tus aplicaciones de mensajería, redes sociales o música se apoyan en él para ofrecerte funciones específicas. Y detrás de todo, existen miles de herramientas de desarrollo que los programadores han usado para que esa experiencia funcione de manera estable.



Sistema Operativo

Android o iOS gestionan recursos como memoria y batería, siendo la base de todo.



Aplicaciones

Mensajería, redes sociales o música se apoyan en el sistema operativo para ofrecerte funciones específicas.



Herramientas de Desarrollo

Miles de herramientas usadas por programadores para que tu experiencia móvil funcione de manera estable.

Tipos de software

En el mercado laboral y en la práctica profesional se distinguen tres categorías principales:

Software de sistema

Actúa como la base sobre la que se apoyan los demás programas. Incluye los sistemas operativos (Windows, Linux, macOS), los controladores que hacen que tu impresora o tarjeta gráfica funcionen, y las utilidades de gestión básica. Es como el "director de orquesta" que coordina al hardware para que responda a las órdenes de manera armónica.

Software de aplicación

Es el que utilizas directamente para realizar tareas concretas. Desde procesadores de texto como Word, hasta aplicaciones móviles como Spotify, pasando por programas de facturación, navegadores web o plataformas de videoconferencia. Cada uno responde a una necesidad específica del usuario o de la organización.

Software de desarrollo

Menos visible para el público general, pero esencial en la industria. Aquí se encuentran los entornos de desarrollo integrados (IDEs como Visual Studio Code, Eclipse o IntelliJ), compiladores, frameworks, bibliotecas y kits de desarrollo (SDKs). Este software es, en esencia, lo que permite que los profesionales creen más software.

El proceso de creación de software

Uno de los mitos más habituales es pensar que desarrollar software es únicamente sentarse frente al ordenador y escribir líneas de código. La realidad es que la programación es solo una fase dentro de un **proceso más amplio** conocido como ciclo de vida del software. Este proceso incluye:

01

Análisis de requisitos

Entender qué problema se quiere resolver, qué espera el usuario y cuáles son las limitaciones técnicas o de negocio.

02

Diseño

Definir la arquitectura del sistema, planificar la interfaz de usuario, elegir tecnologías y establecer cómo se organizarán los datos.

03

Programación

Traducir ese diseño en un lenguaje que la máquina pueda entender. Aquí entra en juego la escritura de código.

04

Pruebas

Verificar que lo que se ha programado cumple los requisitos y funciona correctamente. Incluye pruebas unitarias, de integración y de aceptación por parte del usuario.

05

Mantenimiento

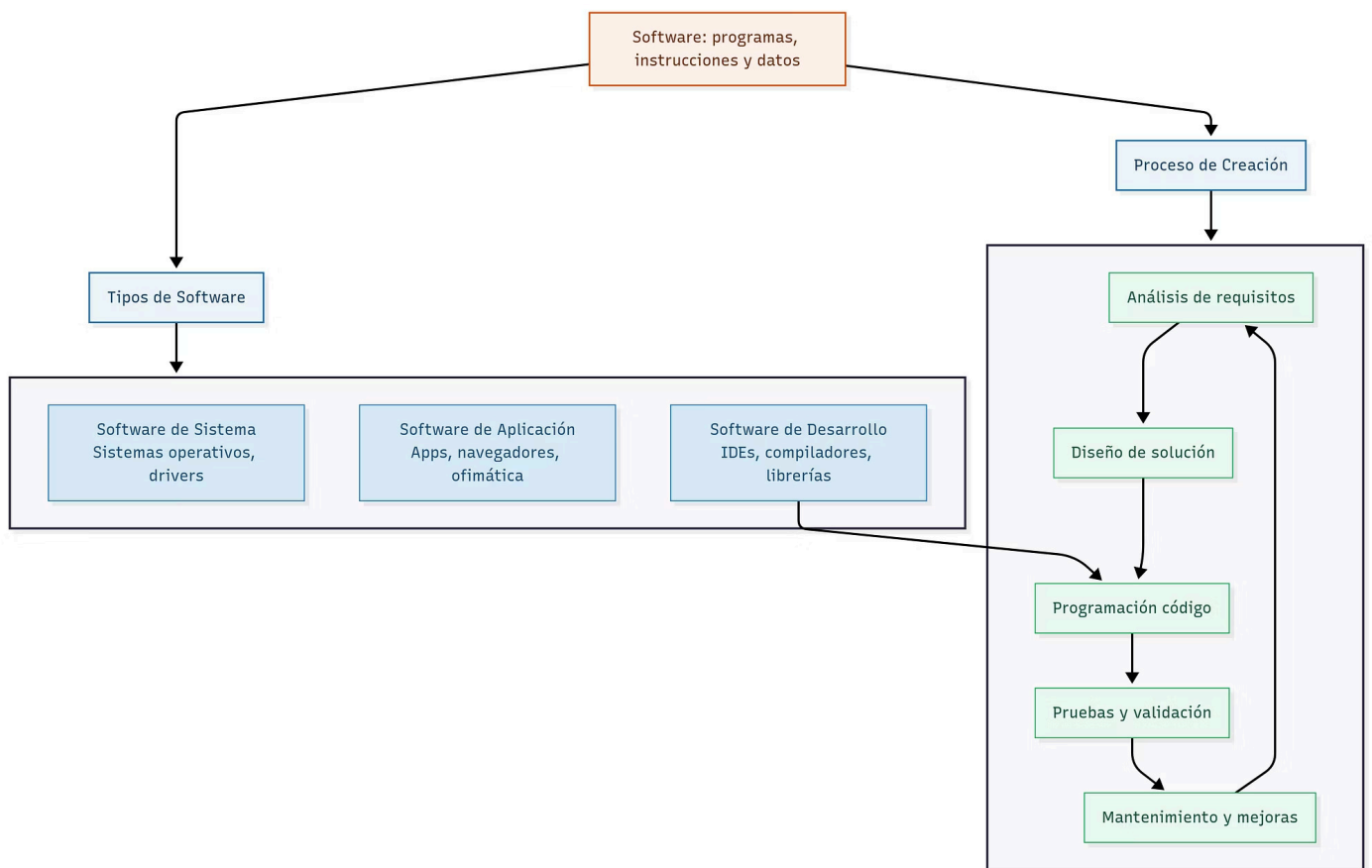
Una vez lanzado, el software requiere correcciones, actualizaciones y mejoras continuas. Ningún programa queda congelado; siempre hay cambios por seguridad, rendimiento o adaptación al mercado.

Hoy en día, este proceso rara vez lo realiza una sola persona. El desarrollo de software es una actividad **colaborativa y multidisciplinar**. En un mismo proyecto participan analistas que definen requisitos, diseñadores UX/UI que piensan en la experiencia del usuario, programadores que construyen la solución, testers que garantizan la calidad, y especialistas en operaciones (DevOps) que aseguran que el software funcione en entornos reales.

En resumen: el software es la "inteligencia" de las máquinas y su creación es un proceso tanto **creativo** como **técnico**, donde convergen la lógica, la organización y la colaboración humana.

Esquema Visual

El siguiente diagrama conceptual resume los tipos de software y el proceso de creación. Está diseñado en formato Mermaid para poder recrearlo con exactitud:



Interpretación:

- El nodo central (A) define el software.
- El bloque "Tipos de Software" muestra las tres categorías principales.
- El bloque "Proceso de Creación" representa el ciclo de vida en bucle continuo.
- La flecha de B3 hacia C3 indica que las herramientas de desarrollo habilitan la programación.








Caso de Estudio – WhatsApp

Contexto

WhatsApp nació en 2009 como una aplicación simple de mensajería. En ese momento, los SMS seguían siendo muy usados, pero tenían coste por mensaje. La propuesta de WhatsApp era ofrecer **mensajes instantáneos gratuitos**, siempre que el usuario tuviera conexión a internet. La app creció rápidamente hasta ser adquirida por Facebook (hoy Meta) en 2014 por más de **19.000 millones de dólares**.

Estrategia

WhatsApp siguió todas las fases clave del proceso de creación de software, adaptándose y evolucionando continuamente:

 Análisis	 Diseño	 Programación
Detectaron una necesidad global de comunicación rápida, gratuita y sencilla.	Interfaz mínima y protocolos eficientes para reducir el consumo de datos.	Uso de desarrollo nativo en iOS y Android con un backend escalable basado en Erlang y sistemas distribuidos.
 Pruebas	 Mantenimiento	
Validación intensiva de rendimiento y estabilidad, clave para soportar miles de millones de mensajes diarios.	Incorporación continua de nuevas funciones: llamadas de voz (2015), videollamadas (2016), cifrado de extremo a extremo, WhatsApp Business y, recientemente, canales de difusión.	

Resultado

Hoy WhatsApp tiene más de **2.000 millones de usuarios activos** en más de 180 países. La clave de su éxito ha sido mantener la simplicidad y evolucionar de manera iterativa.

El caso demuestra que el software no es solo código: es visión estratégica, atención al usuario, diseño de experiencia y evolución constante.

Herramientas y Consejos



Diferencia los tipos de software en la práctica profesional:

- Instala distribuciones de **Linux** para experimentar con software de sistema.
- Usa aplicaciones de productividad como **Google Docs** o **Slack** para identificar software de aplicación.
- Descarga un IDE como **Visual Studio Code** para familiarizarte con software de desarrollo.



Domina entornos de programación modernos:

- IDEs: *Eclipse* para Java, *IntelliJ* para Kotlin, *VS Code* para múltiples lenguajes.
- Control de versiones: *GitHub* o *GitLab* para colaborar en equipo.
- Plataformas de práctica online: *Replit* o *CodeSandbox*.



Empieza con proyectos pequeños pero completos:

- Una calculadora básica en Python.
- Un blog personal con HTML, CSS y JavaScript.
- Un chatbot simple usando librerías de IA en Python.



Adopta metodologías de desarrollo:

- Conoce **modelo en cascada** y **metodologías ágiles** como Scrum.
- Usa herramientas como *Jira* o *Trello* para organizar tareas en sprints.



Integra la calidad desde el inicio:

- Haz pruebas unitarias con *pytest* o *JUnit*.
- Automatiza builds con *GitHub Actions*.
- Documenta con *Markdown* o *Swagger* en proyectos con APIs.

Mitos y Realidades

✗ Mito: "El software se crea solo programando". → FALSO.

La programación es solo una parte. El software surge de un proceso completo que incluye análisis, diseño, pruebas y mantenimiento. Sin estas fases, el código sería ineficiente, inseguro y poco útil.

✗ Mito: "Mientras funcione en la primera versión, ya está terminado". → FALSO.

Todo software necesita mantenimiento. Los sistemas operativos lanzan parches de seguridad, las apps se actualizan con nuevas funciones y la compatibilidad con dispositivos evoluciona constantemente. El desarrollo es un ciclo, no un acto único.



Resumen final

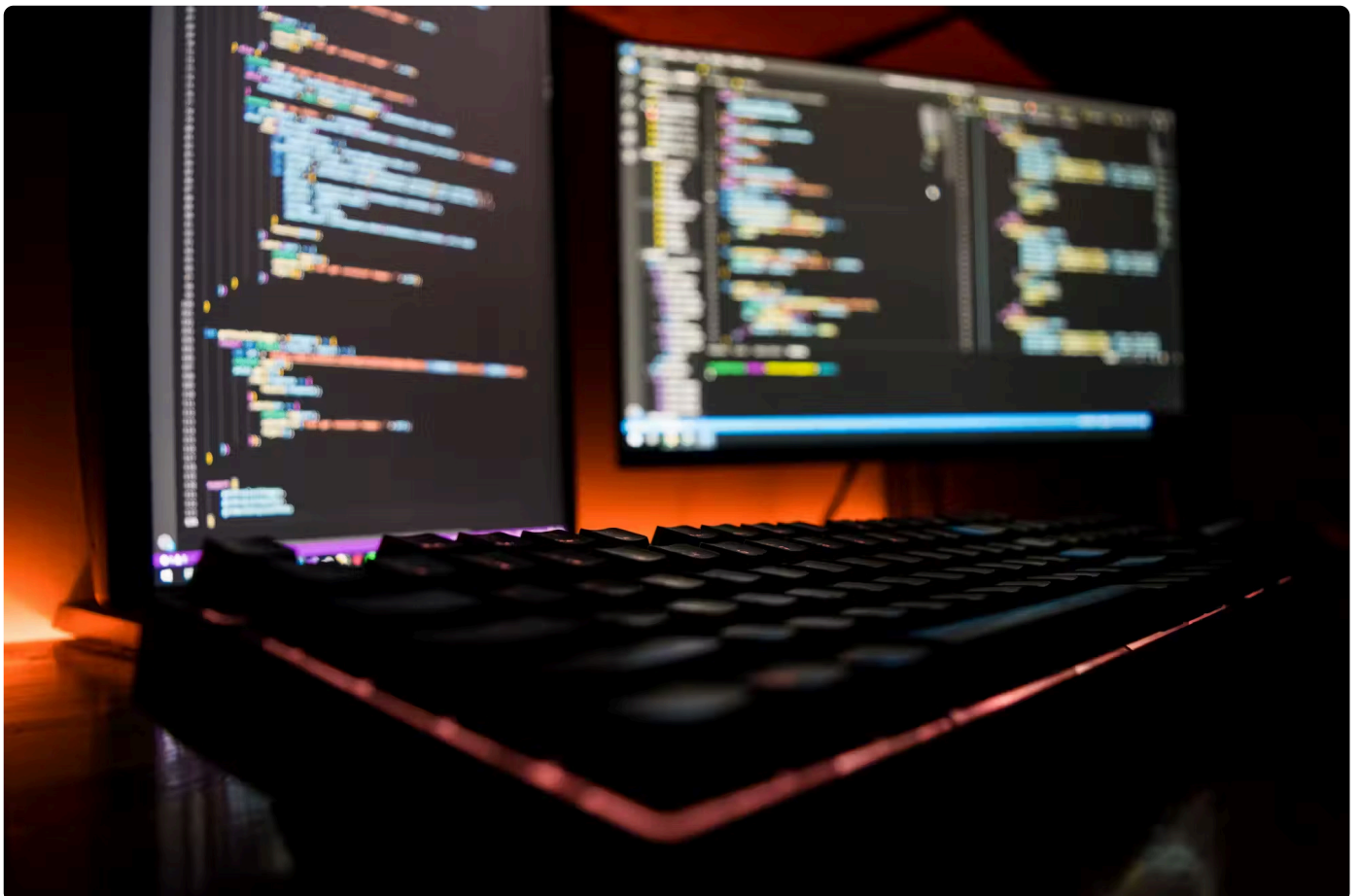
- **Software = programas, instrucciones y datos** que dan vida al hardware.
- **Tipos:** sistema (SO, drivers), aplicación (apps de usuario), desarrollo (IDEs, compiladores).
- **Creación:** análisis → diseño → programación → pruebas → mantenimiento.
- **Clave:** trabajo en equipo, calidad continua, evolución según necesidades reales.

Sesión 2 – Lenguajes de programación y clasificación actual

Un ordenador solo entiende **lenguaje máquina**, es decir, una sucesión de ceros y unos que activan o desactivan transistores. Este sistema binario es imposible de manejar de forma práctica por una persona. Por eso existen los **lenguajes de programación**, que actúan como un puente entre la forma en la que pensamos los humanos y la forma en la que procesa un ordenador.

Un **lenguaje de programación** se define como un conjunto de reglas, sintaxis y estructuras que permiten expresar instrucciones que serán traducidas posteriormente a lenguaje máquina. Al igual que los idiomas naturales (español, inglés, chino), cada lenguaje tiene su gramática, sus palabras clave y sus usos específicos.

Los lenguajes de programación han evolucionado desde los años 50 hasta la actualidad, pasando de simples códigos de ensamblador a lenguajes potentes, expresivos y con grandes comunidades de soporte. Para un profesional del software es fundamental no solo conocer uno o dos lenguajes, sino **entender cómo se clasifican y qué ventajas ofrecen en distintos contextos**.



Clasificación de lenguajes de programación

Las principales clasificaciones son:

Según el nivel de abstracción

- **Lenguajes de bajo nivel:** se aproximan al hardware. Ejemplo: ensamblador, que permite manipular registros del procesador directamente. Su ventaja es el control absoluto sobre el rendimiento; su desventaja, la dificultad y el tiempo de desarrollo.
- **Lenguajes de alto nivel:** más cercanos al lenguaje humano. Ejemplos: Java, Python, C#, JavaScript. Son fáciles de leer, permiten programar más rápido y son hoy el estándar en el mercado.

Según el paradigma de programación

El paradigma define el estilo o enfoque para resolver problemas:

- **Imperativo:** dicta instrucciones paso a paso. Ejemplo: C.
- **Declarativo:** describe el resultado deseado sin detallar cómo lograrlo. Ejemplo: SQL, HTML.
- **Funcional:** inspirado en las matemáticas, donde todo se define como funciones puras. Ejemplo: Haskell, Scala, parte de Python.
- **Orientado a objetos (OOP):** organiza el código en clases y objetos que representan entidades del mundo real. Ejemplo: Java, C++, C#.

Según el ámbito de aplicación

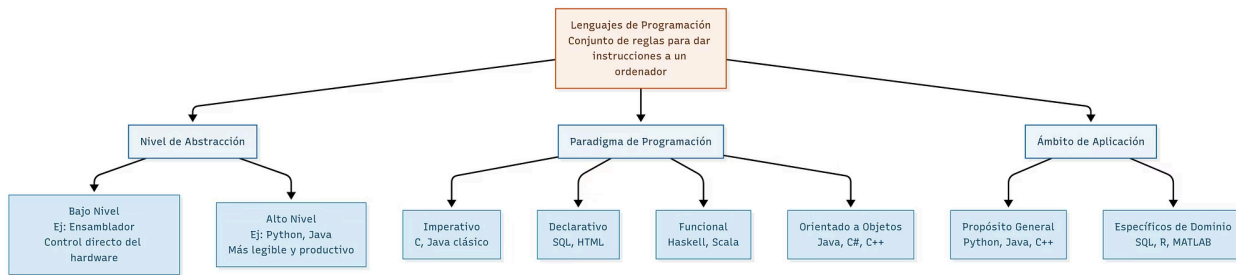
- **Lenguajes de propósito general:** sirven para múltiples dominios. Ejemplos: Python, Java, C++.
- **Lenguajes específicos de dominio (DSLs):** creados para tareas concretas. Ejemplos: SQL para bases de datos, R para estadística, MATLAB para cálculos matemáticos.

Realidad profesional actual

Hoy los proyectos rara vez dependen de un solo lenguaje. Una aplicación web moderna combina HTML, CSS y JavaScript en el frontend, Python, PHP o Java en el backend, y SQL para bases de datos. Las empresas seleccionan la combinación adecuada en función de los objetivos, el equipo disponible y el ecosistema de herramientas.

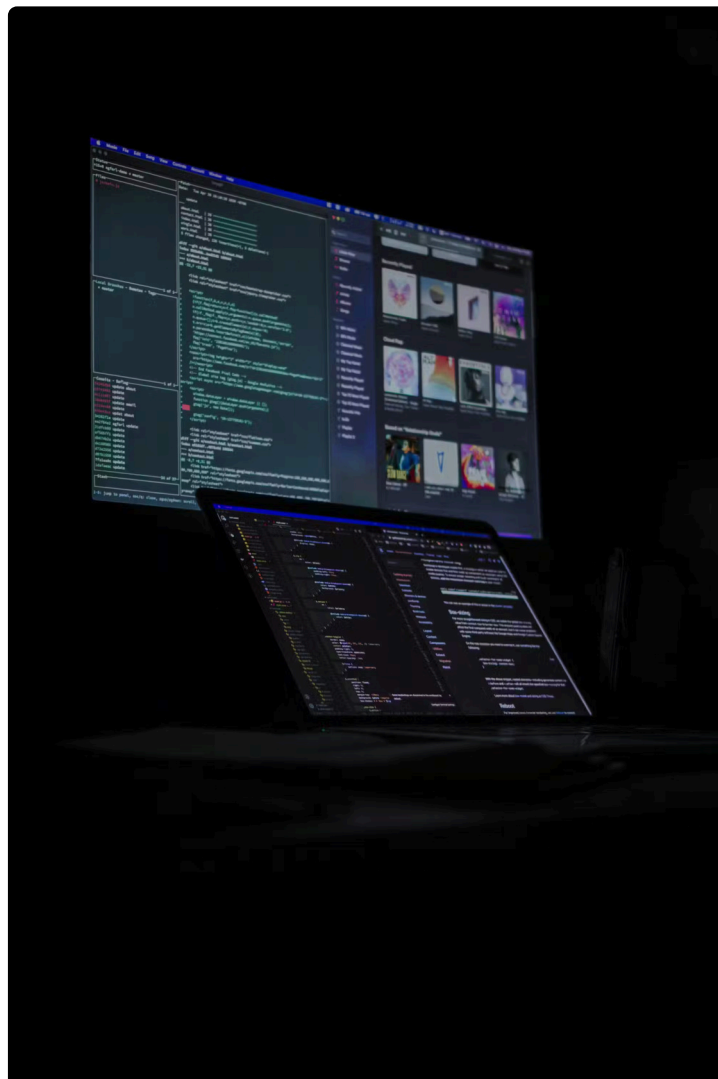
Esquema visual

Aquí tienes el diagrama conceptual de clasificación de los lenguajes, listo para recrearse con Mermaid:



Qué representa:

- Nodo central (A): definición de lenguaje de programación.
- Bloques secundarios: criterios de clasificación.
- Cada subnodo muestra ejemplos y características clave.



Caso de Estudio – Plataforma de cursos online (Coursera)

Contexto

Coursera es una plataforma líder de educación en línea con más de 100 millones de usuarios. Necesita ofrecer contenido a nivel global, en múltiples dispositivos y con un sistema robusto que gestione vídeos, foros, cuestionarios y certificados.

Estrategia

El desarrollo de Coursera integra diferentes lenguajes en función de cada capa del sistema:

- **Frontend (interfaz de usuario):** HTML, CSS y JavaScript (con frameworks como React) para ofrecer una experiencia interactiva y adaptable a móviles.
- **Backend (lógica de negocio):** Python con Django, que facilita la gestión de usuarios, cursos, evaluaciones y pagos.
- **Bases de datos:** SQL (PostgreSQL) para almacenar datos estructurados.
- **Análisis de datos:** R y Python para procesar grandes volúmenes de información y personalizar recomendaciones de cursos.
- **Automatización y despliegue:** scripts en Bash y herramientas de infraestructura en la nube (AWS, Kubernetes).

Resultado

Coursera ha demostrado cómo un ecosistema de lenguajes bien integrado permite escalar un proyecto hasta una audiencia global. Su capacidad de combinar lenguajes de distintos paradigmas y ámbitos de aplicación es una de las razones de su éxito.

Herramientas y Consejos

Construye una base sólida con un lenguaje multipropósito

- *Python*: ideal para aprender y trabajar en web, ciencia de datos, inteligencia artificial.
- *Java*: ampliamente usado en empresas, especialmente en backend y Android.
- *JavaScript*: el estándar del desarrollo web.

👉 Consejo: domina uno de estos lenguajes y complementa después con otros más específicos.

Aprende lenguajes específicos que complementen tu perfil

- *SQL*: obligatorio en casi cualquier proyecto con bases de datos.
- *R*: útil si te interesa la analítica o la investigación.
- *HTML/CSS*: imprescindibles para web.

👉 Consejo: añade al menos un DSL (lenguaje específico de dominio) a tu repertorio.

Utiliza plataformas para practicar sin instalar nada

- *Replit*, *CodeSandbox*, *JDoodle*: ejecuta código online en múltiples lenguajes.
- *HackerRank* o *LeetCode*: resuelve ejercicios y prepárate para entrevistas técnicas.

👉 Consejo: dedica al menos 15 minutos diarios a practicar ejercicios de algoritmia en diferentes lenguajes.

Mantente actualizado con tendencias del sector

- *TIOBE Index*: ranking mensual de popularidad de lenguajes.
- *Stack Overflow Developer Survey*: muestra qué lenguajes son más usados en la industria.

👉 Consejo: revisa estos rankings cada seis meses para saber si tu stack sigue siendo demandado.

Mitos y Realidades

❌ Mito: "Un programador solo necesita dominar un lenguaje".

→ **FALSO**. En proyectos reales se usan varios lenguajes a la vez. Por ejemplo, una web moderna combina HTML/CSS/JavaScript en frontend, Python o Java en backend y SQL en la base de datos. Saber un solo lenguaje te limita en el mercado laboral.

❌ Mito: "Aprender un lenguaje es suficiente para siempre".

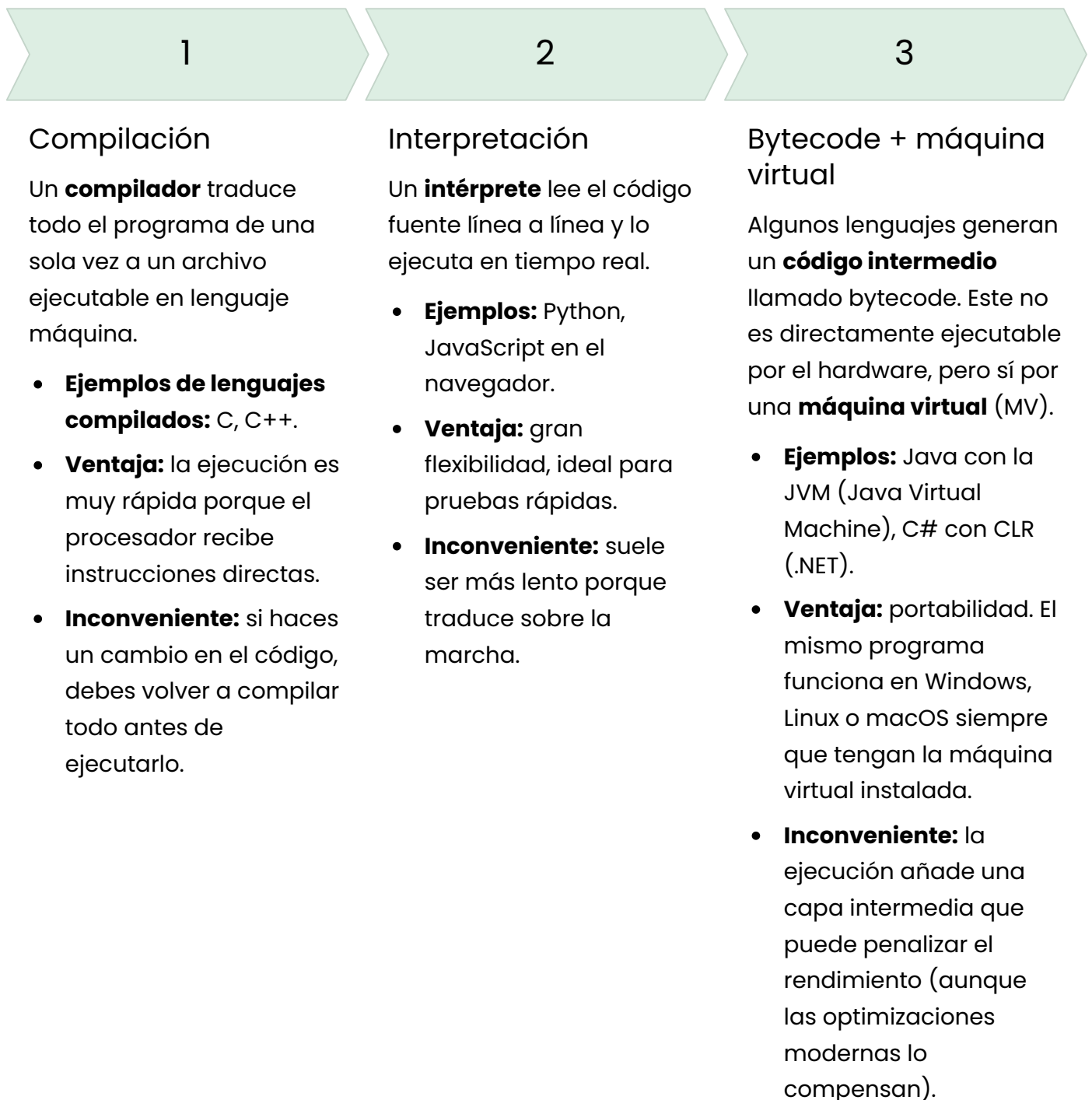
→ **FALSO**. Los lenguajes evolucionan y las tendencias cambian. Hoy Python está en auge, pero hace 15 años lo estaba Java, y mañana puede ser Go o Rust. Un profesional debe estar dispuesto a aprender de forma continua y adaptarse a nuevas herramientas.

📋 Resumen final

- Lenguajes de programación = reglas que permiten dar instrucciones a un ordenador.
- Clasificación:
 - Nivel: bajo (ensamblador) vs. alto (Java, Python).
 - Paradigma: imperativo, declarativo, funcional, orientado a objetos.
 - Aplicación: generales vs. específicos de dominio.
- En proyectos actuales se combinan varios lenguajes.
- Ejemplo Coursera: frontend (JavaScript), backend (Python), bases de datos (SQL), analítica (R).
- Mito: no basta con un solo lenguaje ni se puede dejar de aprender.

Sesión 3: Proceso de ejecución: compilación, interpretación y bytecode

Cuando escribes un programa en Python, Java o C++, lo que produces es **código fuente**: texto con instrucciones comprensibles para un humano. Pero los ordenadores no entienden palabras como print, if o while. Solo entienden **lenguaje máquina**, que es una secuencia de ceros y unos que activan transistores. Aquí entra en juego el **proceso de ejecución**, que traduce el código a algo que el hardware pueda procesar. Existen tres modelos principales de ejecución:



La práctica actual

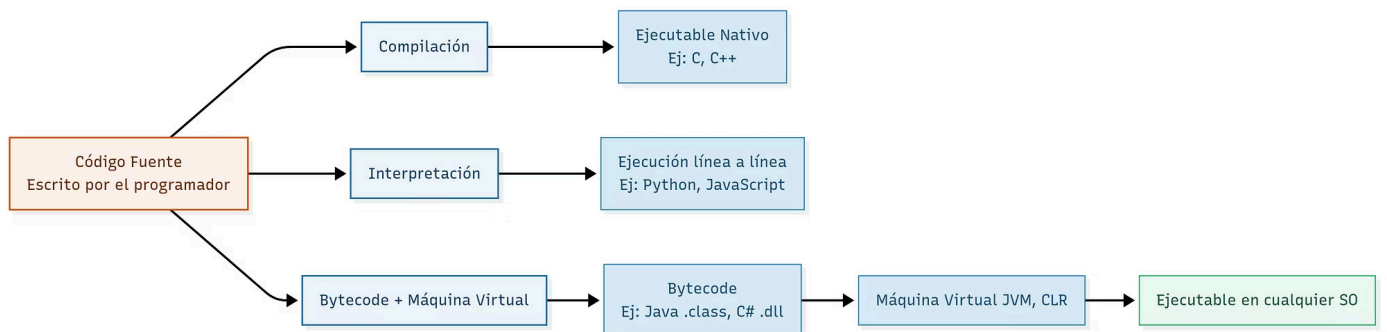
En el mundo real, la frontera entre estos modelos es menos rígida. Python, aunque es interpretado, compila internamente a un bytecode (.pyc) para mejorar el rendimiento. JavaScript en navegadores modernos combina interpretación con **JIT (Just-In-Time Compilation)**: traduce fragmentos a código máquina justo antes de ejecutarlos, logrando velocidad cercana a los compilados.

Comprender estos modelos es esencial porque influye en decisiones de proyecto:

- Si necesitas velocidad extrema (un motor de videojuegos, software aeroespacial), probablemente recurras a compilación nativa.
- Si buscas rapidez de desarrollo y flexibilidad (prototipos, ciencia de datos), la interpretación es muy útil.
- Si el objetivo es portabilidad y escalabilidad, las máquinas virtuales como la JVM o la CLR son ideales.

Esquema Visual

Aquí tienes un diagrama que representa los tres modelos de ejecución y su flujo, listo para recrearse en Mermaid:



Interpretación del diagrama:

- El nodo central representa el código fuente inicial.
- **Compilación:** lleva directamente a un ejecutable nativo optimizado.
- **Interpretación:** ejecuta el código paso a paso en tiempo real.
- **Bytecode + máquina virtual:** genera un intermediario que se ejecuta en cualquier sistema gracias a la máquina virtual.



Caso de Estudio – Aplicación bancaria en Java

Contexto

Un banco internacional busca desarrollar una aplicación de gestión de cuentas y transacciones para clientes de múltiples países. El reto es doble:

1. **Seguridad crítica**, porque se trata de dinero.
2. **Portabilidad**, porque debe funcionar en Windows (oficinas), Linux (servidores) y macOS (algunos clientes).

Estrategia

- Evaluaron **C++ compilado**: excelente rendimiento, pero cada sistema operativo requeriría un ejecutable diferente, aumentando los costes de mantenimiento.
- Evaluaron **Python interpretado**: portabilidad sencilla, pero el rendimiento no era suficiente para procesos de millones de transacciones por minuto.
- Eligieron **Java**, que compila el código fuente a **bytecode** y lo ejecuta en la **JVM**. Esto asegura un equilibrio entre rendimiento y portabilidad.

Resultado

La aplicación se desplegó en múltiples países usando la misma base de código. Gracias a la JVM, la portabilidad se garantizó sin costes adicionales de recompilación por sistema. Además, con las optimizaciones **JIT**, el rendimiento alcanzó niveles cercanos a los compilados. Este caso muestra cómo la elección del modelo de ejecución es estratégica para proyectos críticos.

Además, Java ofrece:

- Librerías robustas para seguridad (cifrado, firmas digitales).
- Un ecosistema maduro para aplicaciones empresariales (Spring Framework, Hibernate).
- Comunidad global que garantiza soporte y actualizaciones.

Herramientas y Consejos

Experimenta con compiladores e intérpretes básicos

Usa gcc o g++ para compilar programas en C/C++. Ejecuta programas en Python con `python archivo.py`. Compila y ejecuta Java con `javac` y `java`. 🙌 Consejo: escribe el clásico "Hola mundo" en C++, Python y Java para comparar el flujo.

Explora máquinas virtuales

Instala la **JVM** para ejecutar bytecode Java. Aprende cómo funciona el **CLR** de .NET para C#. Usa **Docker** para simular entornos y comprobar portabilidad.

Aprovecha optimizaciones modernas

Conoce la **compilación JIT** en lenguajes como JavaScript o Java. Usa herramientas de profiling como *VisualVM* o *PySpy* para analizar rendimiento. Aplica técnicas como *caching* y *multithreading* según el lenguaje.

Aplica la teoría en proyectos reales

Desarrolla un script en Python para automatizar tareas. Implementa un pequeño módulo en C++ para medir rendimiento. Construye una miniaplicación en Java y comprueba cómo funciona en Windows y Linux sin modificar el código.

Mitos y Realidades

❌ Mito: "Los lenguajes interpretados siempre son lentos e ineficientes". → FALSO.

Con las optimizaciones modernas como **JIT**, muchos lenguajes interpretados alcanzan rendimientos muy próximos a los compilados. JavaScript, considerado lento en los años 90, hoy impulsa aplicaciones complejas gracias a motores como **V8 de Google**.

❌ Mito: "Si un programa compila, ya funcionará bien". → FALSO.

Compilar solo significa que la sintaxis es correcta y se ha generado un ejecutable. Pero el programa puede tener errores lógicos, problemas de seguridad o fallos de rendimiento. Por eso, además de compilar, es esencial realizar **pruebas exhaustivas**.

📄 Resumen final

- El ordenador solo entiende **lenguaje máquina (binario)**.
- **Compilación:** traduce todo a ejecutable → rápido (C, C++).
- **Interpretación:** ejecuta línea a línea → flexible pero más lento (Python, JS).
- **Bytecode + máquina virtual:** equilibrio entre rendimiento y portabilidad (Java, C#).
- En la práctica, muchos lenguajes combinan estrategias (compilación + interpretación + JIT).

Sesión 4 – Ciclo de vida y metodologías de desarrollo (cascada vs. iterativas).

El desarrollo de software no es un conjunto improvisado de tareas, sino un proceso estructurado conocido como **ciclo de vida del software**. Este ciclo describe todas las fases necesarias para llevar una idea desde su concepción hasta su funcionamiento real en manos de los usuarios. Las fases comunes son:

01

Análisis de requisitos

Identificar qué se necesita, quién lo pide y con qué restricciones. Aquí se definen funcionalidades, reglas de negocio, limitaciones técnicas y prioridades.

02

Diseño

Transformar los requisitos en un plan técnico. Incluye la arquitectura del sistema, modelos de datos, diagramas de clases, flujos de usuario y prototipos.

03

Implementación (programación)

Escribir el código que materializa el diseño.

04

Pruebas

Validar que el software funciona como se esperaba y que cumple los requisitos. Se hacen pruebas unitarias, de integración, de aceptación y de seguridad.

05

Despliegue

Poner el software en un entorno productivo, accesible a los usuarios finales.

06

Mantenimiento

Resolver errores, actualizar librerías, añadir nuevas funciones y garantizar que el sistema siga siendo útil y seguro.

Modelos de organización del ciclo de vida

Existen varias metodologías para organizar estas fases. Las dos grandes corrientes son:

Modelo en cascada

Propone que cada fase se complete de manera secuencial. Primero análisis, después diseño, luego programación, pruebas y despliegue. Solo cuando una etapa termina se pasa a la siguiente.

- **Ventaja:** claridad, orden y control. Útil en proyectos donde los requisitos son estables y poco cambiantes (ej. sistemas aeroespaciales o médicos).
- **Inconveniente:** rigidez. Si surgen cambios o se descubren errores tarde, rectificar es costoso.

Modelos iterativos o ágiles

Plantean ciclos cortos en los que se repiten todas las fases (análisis, diseño, implementación y pruebas), entregando una versión funcional en cada iteración.

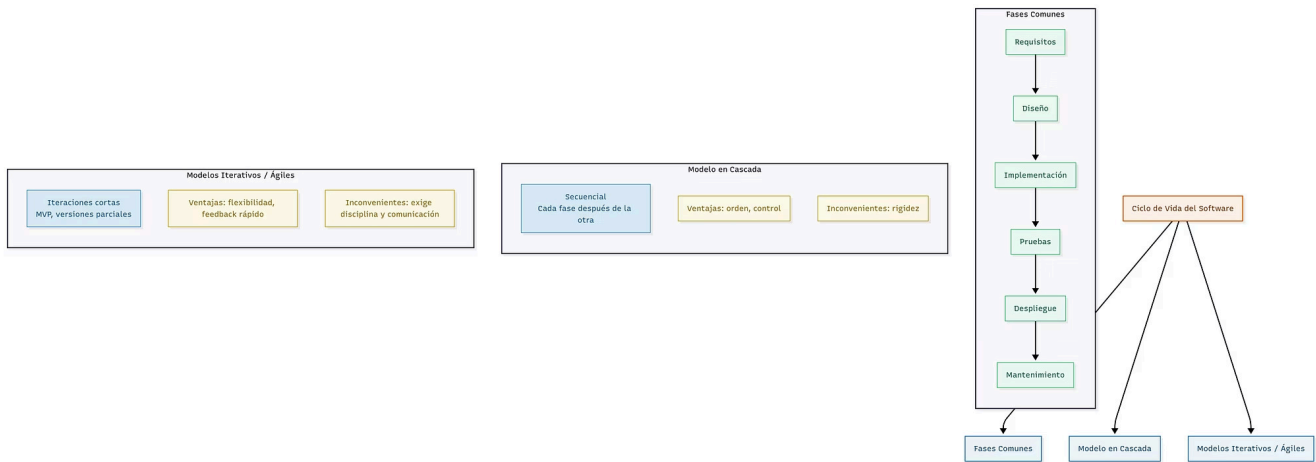
- **Ventaja:** adaptabilidad y feedback continuo.
- **Inconveniente:** requiere disciplina, buena comunicación y más participación del cliente.

La gran diferencia radica en el momento de entrega: En **cascada**, el producto final llega al final del proceso. En **iterativos**, se ofrecen versiones parciales desde el inicio, que evolucionan con las necesidades reales.

Hoy en día, la mayoría de las empresas de software utilizan metodologías **ágiles** como Scrum o Kanban, que se basan en ciclos iterativos y flexibles. Sin embargo, el modelo en cascada sigue vigente en entornos regulados donde la seguridad y el control documental son prioritarios.

Esquema Visual

Diagrama conceptual del ciclo de vida del software y sus metodologías, listo para recrearse en Mermaid:



Interpretación:

- El nodo central es el ciclo de vida.
- Bloque B muestra fases comunes en secuencia.
- Bloque C describe el modelo en cascada (lineal y rígido).
- Bloque D representa los iterativos/ágiles (ciclos cortos y adaptativos).





Caso de Estudio – App de reservas de restaurantes

Contexto

Una startup gastronómica en España quiere lanzar una aplicación para que los usuarios reserven mesas en restaurantes. El mercado es competitivo y ya existen soluciones parecidas.

Estrategia

Enfoque en cascada:

el equipo pasó meses recopilando requisitos, diseñando la aplicación completa y programando todas las funciones antes de lanzar. Resultado: cuando salió, la app era completa pero había perdido relevancia: competidores ya ofrecían funciones más modernas (pagos móviles, reseñas sociales).

Enfoque iterativo (Scrum):

decidieron probar una segunda estrategia. Crearon un **MVP (Producto Mínimo Viable)** con la función básica: búsqueda y reserva. Lo lanzaron en pocas semanas. Cada sprint de dos semanas añadía mejoras:

- Sprint 2: notificaciones de confirmación.
- Sprint 3: integración con pagos.
- Sprint 4: sistema de reseñas de usuarios.
- Sprint 5: recomendaciones personalizadas.

Resultado

Con el modelo iterativo, la empresa entró en el mercado mucho antes, ganó usuarios iniciales y pudo evolucionar basándose en feedback real. Aunque el cascada ofrecía un producto más pulido desde el inicio, el enfoque ágil les permitió ser competitivos y evitar quedar desfasados.

Herramientas y Consejos

Cuando usar cascada

- Proyectos de **infraestructura crítica** (aeroespacial, banca, medicina).
- Necesidad de control documental y auditorías.
- Herramienta clave: *Microsoft Project* para planificación detallada.

👉 Consejo: si trabajas en sectores regulados, acepta que la flexibilidad es limitada, pero puedes reforzar con revisiones tempranas.

Cuando usar iterativos/ágiles

- Startups, productos digitales, entornos cambiantes.
- Equipos que trabajan con incertidumbre y necesitan reaccionar rápido.
- Herramientas: *Jira, Trello, ClickUp, Asana* para gestión de sprints y tableros Kanban.

👉 Consejo: mantén reuniones cortas de sincronización diaria (daily stand-ups) para asegurar la alineación del equipo.

Integrar calidad en ambos enfoques

- Versionado con *Git* y plataformas como *GitHub/GitLab*.
- Pruebas automatizadas: *JUnit, pytest, Selenium*.
- Integración y despliegue continuo: *GitHub Actions, GitLab CI, Jenkins*.

👉 Consejo: incluso en cascada, automatiza pruebas y despliegues para reducir errores humanos.

Documentar sin fricción

- Usa *Confluence* o *Notion* para centralizar decisiones y requisitos.
- Diseña prototipos con *Figma* para alinear expectativas de usuarios y desarrolladores.

👉 Consejo: combina documentación ligera con entregas funcionales frecuentes.

Mitos y realidades

❌ **Mito:** "Las metodologías ágiles eliminan la necesidad de planificar". → **FALSO.**

Ágil no significa improvisación. Los proyectos requieren planificación constante, pero se hace en ciclos cortos, con revisión y adaptación. La diferencia es que la planificación no es rígida, sino flexible y evolutiva.

❌ **Mito:** "Las metodologías ágiles eliminan la necesidad de planificar". → **FALSO.**

Ágil no significa improvisación. Los proyectos requieren planificación constante, pero se hace en ciclos cortos, con revisión y adaptación. La diferencia es que la planificación no es rígida, sino flexible y evolutiva.

📋 Resumen final

- **Ciclo de vida = fases comunes:** requisitos → diseño → implementación → pruebas → despliegue → mantenimiento.
- **Cascada:** secuencial, rígido, útil en proyectos críticos con requisitos estables.
- **Iterativos/ágiles:** entregas parciales, feedback continuo, mayor flexibilidad.
- Elección depende del contexto: cascada en entornos regulados, ágil en mercados dinámicos.