Código fuente, objeto y ejecutable

Entornos de desarrollo

Diapositivas realizadas por Aitor Ventura Edo

Del código fuente al binario: ¿cómo llega un programa a la CPU?

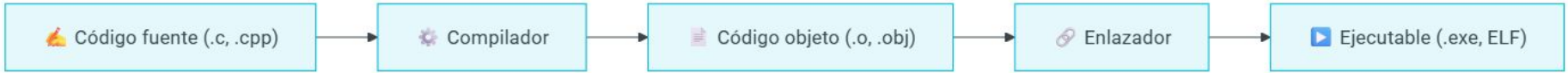
Idea

Idea básica

Un ordenador **solo entiende ceros y unos (binario)**.
Los lenguajes que usamos para programar (C, Java, Python…) deben traducirse hasta llegar a ese formato.
Según el lenguaje, este “viaje” puede ser:

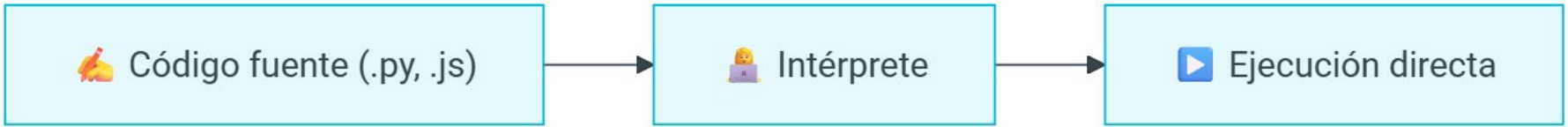
- **Compilado** (todo antes de ejecutar).
- **Interpretado** (línea a línea en el momento).
- **Híbrido con máquina virtual y JIT** (un punto intermedio).

Lenguaje compilado (C, C++) | Lenguaje interpretado (Python, JS) | Híbrido con VM (Java, C#)



- El código se traduce **antes de ejecutarse**.
- Resultado: un **ejecutable independiente** que la CPU entiende directamente.
- Ejemplo: programas de escritorio o videojuegos en C++.

Lenguaje compilado (C, C++) | **Lenguaje interpretado (Python, JS)** | Híbrido con VM (Java, C#)



- El código se lee **línea a línea** en tiempo real.
- No se genera un ejecutable clásico.
- Ejemplo: scripts de Python, páginas web con JavaScript.

Lenguaje compilado (C, C++) | Lenguaje interpretado (Python, JS) | **Híbrido con VM (Java, C#)**

- El compilador genera un **bytecode intermedio**.
- Ese bytecode se ejecuta dentro de una **máquina virtual**.
- La VM usa **JIT** para traducir lo más usado a binario real en tiempo de ejecución.
- Ejemplo: aplicaciones Android (Java/Kotlin).

Compilación, ensamblado y enlazado (en compilados)

Definiciones

- **Compilación** → traduce el código fuente a **código objeto** (binario incompleto).
- **Ensamblado** → organiza esas instrucciones en código máquina real.
- **Enlazado (link)** → une todo (archivos objeto + librerías) para obtener el **ejecutable final**.

Definiciones

Enlazado estático vs dinámico

- **Estático** → el ejecutable incluye todo. Más grande, pero no necesita librerías externas.
- **Dinámico** → el ejecutable depende de librerías externas (`.dll` , `.so`). Más ligero, pero puede fallar si faltan.

Diferencia entre código objeto y ejecutable

📄 **Código objeto** | ▶️ **Código binario ejecutable**

- Archivo **intermedio** tras compilar.
- Tiene instrucciones en binario, pero con **marcas y símbolos sin resolver**.
- Ejemplo: `programa.o` (Linux), `programa.obj` (Windows).
- ⚠️ **No puede ejecutarse directamente**.

📄 Código objeto | ▶️ Código binario ejecutable

- Archivo **final** que puedes abrir (`.exe` , ELF, Mach-O).
- Ya tiene todas las direcciones resueltas y librerías enlazadas.
- Ejemplo: `notepad.exe` en Windows, `/bin/ls` en Linux.
- ✅ **La CPU lo entiende directamente**.

Ejecutable y dependencias

Ejecutable

Un ejecutable puede necesitar:

- Librerías (`libc.so`, `msvcrt.dll`).
- Archivos de configuración.
- Recursos extra (imágenes, sonidos, bases de datos…).

Error común

Creer que un `.exe` lo tiene **todo dentro**.
En realidad, muchos programas fallan si falta una librería externa.

Interpretado vs compilado vs JIT

Tipo	Cómo funciona	Ejemplo	Pros	Contras
Interpretado	El intérprete lee y ejecuta el código **línea a línea**, en tiempo real.	Python, JavaScript	+ **Flexible**: se puede probar y modificar fácilmente sin recompilar. + **Multiplataforma**: el mismo script puede correr en Windows, Linux o macOS siempre que haya intérprete.	– **Más lento**: porque cada instrucción se traduce al vuelo, no está preprocesada. – **Depende del intérprete**: sin él, el programa no puede ejecutarse.

Tipo	Cómo funciona	Ejemplo	Pros	Contras
Compilado	Se traduce **todo el programa** de una vez a código máquina antes de ejecutarlo.	C, C++	+ **Muy rápido**: la CPU ejecuta directamente el binario, sin pasos intermedios. + **Optimizable**: el compilador puede aplicar mejoras (optimizaciones) en el binario.	– **Menos flexible**: cualquier cambio en el código obliga a recompilar. – **Menos portable**: el binario generado suele funcionar solo en un sistema o arquitectura concreta.

Tipo	Cómo funciona	Ejemplo	Pros	Contras
JIT (Just-in-time)	El programa se compila primero a un **código intermedio (bytecode)** y, en tiempo de ejecución, la máquina virtual traduce “al vuelo” las partes más usadas a binario.	Java (JVM), C# (.NET)	+ **Equilibrio**: combina la portabilidad del bytecode con la velocidad del binario. + **Adaptable**: el JIT optimiza según cómo se ejecuta el programa en cada máquina.	– **Más complejo**: necesita tanto un compilador como una máquina virtual. – **Inicio más lento**: al principio puede tardar más porque compila sobre la marcha.

Código intermedio y máquinas virtuales

Idea clave

Algunos lenguajes (Java, C#) no generan binario directo, sino un **código intermedio**.
Ese código se ejecuta dentro de una **máquina virtual (VM)**.

Java | C#

- El compilador genera **bytecode** (`.class`).
- La **JVM** lo ejecuta.
- Para acelerar, usa **JIT** que convierte a binario solo lo más repetido.

Java | C#

- El compilador genera **CIL** (Common Intermediate Language).
- La **CLR** lo ejecuta.
- También usa **JIT** para mejorar rendimiento.

AOT (Ahead-of-Time)

Significa compilar **antes de ejecutar**, evitando JIT.

Ejemplo: GraalVM para Java.

Empaquetado y distribución

Un programa no siempre llega como un único archivo:

- **Instaladores** → `.exe`, `.msi` (Windows), `.deb`, `.rpm` (Linux).
- **Bundling** → programa + dependencias (ej.: apps portables, Electron).
- **Contenedores** → incluyen programa + librerías + mini SO (ej.: Docker).

Ejemplo cotidiano

- 🎮 Videojuego en PC → instalador clásico.
- 📱 App Android → archivo `.apk`.
- 🌐 Servicio web → contenedor Docker.