Programación con Java

1. El inicio

1. El camino **Directo al Código** (*Sin instalación*)

   Si quiere usted empezar ya mismo sin estar configurando variables de entorno, use estas plataformas interactivas:

   - **CodeGym**: Es casi un videojuego. Le da tareas prácticas desde el minuto uno y un verificador automático de código. Es ideal si tiene *alergia* a la teoría densa.
   - **Exercism.org** (*Java Track*): Muy recomendado. Le propone retos y, una vez resueltos, mentores reales revisan su código de forma gratuita para decirle cómo hacerlo de forma más *moderna*.
   - **JDoodle / Playgrounds online**: Si solo quiere probar una idea rápida de Java 21 o 25, entra a un compilador *online* y pege su código. No requiere registro.

2. **Bibliografía Imprescindible** (Para cuando quiera entender el *porqué*)

   Aunque quiera programar directo, siempre llega un momento donde un buen libro le ahorra semanas de confusión. Estas son las ediciones actualizadas a 2026:

   - ***Head First Java*** (3ª Edición): Olvide el formato aburrido. Este libro usa dibujos, acertijos y un tono muy divertido. La 3ª edición ya incluye novedades como **Lambdas** y **Records**. Es el mejor para empezar de cero.
   - ***Modern Java in Action***: Este es el *manual de transición*. Si ya sabes algo de programación, este libro le enseña directamente cómo se usa Java hoy (Streams, Programación Funcional, Reactividad).
   - ***Effective Java*** (*Joshua Bloch*): Este es para cuando ya lleve un par de meses. No te enseña la sintaxis, te enseña las buenas prácticas. Es la biblia que separa a un programador de Java de un buen programador de Java.

3. **Mi Recomendación Estratégica**

   - **Instala IntelliJ IDEA** (**Community Edition**): Es el estándar de la industria. Su asistente de IA y sus sugerencias de código le enseñarán más Java en una tarde que tres capítulos de un libro.
   - Busca el **Java Programming MOOC** (*Massive Open Online Course*) de la **Universidad de Helsinki**: Es gratuito, legendario en la comunidad y puramente práctico.

- **Un pequeño Proyecto rápido**: Intente hacer una API simple con **Spring Boot 3**. Es el framework que domina el mercado. En 10 minutos puede tener un "Hola Mundo" que responda a través de su navegador.

2. Un ejemplo para empezar

1. **Un pequeño *script* inicial en Java moderno** para que lo pruebe ahora mismo en un compilador *online*

   Aquí tiene un ejemplo en **Java moderno** (versión 21 o superior) que utiliza tres joyas actuales:

   - *Hilos Virtuales* (para ir rápido)
   - *Records* (para datos limpios)
   - y la *nueva sintaxis de Switch*.

   Copie este código y pruébelo en un compilador *online* como **JDoodle** (asegúrase de seleccionar **JDK 21** o superior):

```
1 import java.util.List;
2
3 public class Java2026 {
4
5     // 1. RECORD: Una clase completa (con constructor, getters, etc.) en una sola línea.
6     // ¡Adiós al boilerplate de 2005!
7     record Tarea(String nombre, int prioridad) {}
8
9     public static void main(String[] args) {
10        var tareas = List.of(
11            new Tarea("Aprender Java Moderno", 1),
12            new Tarea("Beber café", 2),
13            new Tarea("Configurar el IDE", 3)
14        );
15
16        // 2. VIRTUAL THREADS: Lanzamos un hilo ligero para procesar cada tarea.
17        // En 2005 esto habría sido una pesadilla de memoria; hoy es instantáneo.
18        for (var tarea : tareas) {
19            Thread.startVirtualThread(() -> {
20                procesarTarea(tarea);
21            });
22        }
23
24        // Una pequeña pausa para que los hilos terminen antes de que el programa cierre
25        try { Thread.sleep(100); } catch (InterruptedException e) {}
26    }
27
28    static void procesarTarea(Tarea t) {
29        // 3. SWITCH EXPRESSION: Más limpio, seguro y directo.
30        String mensaje = switch (t.prioridad()) {
31            case 1 -> "🚀 ¡URGENTE!: " + t.nombre();
32            case 2 -> "☕ Rutina: " + t.nombre();
33            default -> "📅 Pendiente: " + t.nombre();
34        };
35
36        System.out.println(mensaje);
37    }
38 }
```

```
import java.util.List;

public class Java2026 {
```

```
    // 1. RECORD: Una clase completa (con constructor, getters, etc.) en una sola línea.
    // ¡Adiós al boilerplate de 2005!
    record Tarea(String nombre, int prioridad) {}

    public static void main(String[] args) {
        var tareas = List.of(
            new Tarea("Aprender Java Moderno", 1),
            new Tarea("Beber café", 2),
            new Tarea("Configurar el IDE", 3)
        );

        // 2. VIRTUAL THREADS: Lanzamos un hilo ligero para procesar cada tarea.
        // En 2005 esto habría sido una pesadilla de memoria; hoy es instantáneo.
        for (var tarea : tareas) {
            Thread.startVirtualThread(() -> {
                procesarTarea(tarea);
            });
        }

        // Una pequeña pausa para que los hilos terminen antes de que el programa cierre
        try { Thread.sleep(100); } catch (InterruptedException e) {}
    }

    static void procesarTarea(Tarea t) {
        // 3. SWITCH EXPRESSION: Más limpio, seguro y directo.
        String mensaje = switch (t.prioridad()) {
            case 1 -> "¡URGENTE!: " + t.nombre();
            case 2 -> "Rutina: " + t.nombre();
            default -> "Pendiente: " + t.nombre();
        };

        System.out.println(mensaje);
    }
}
```

2. **¿Qué está pasando aquí?**

- **`record Tarea(...)`**: En 2005, esto habrían sido unas 40 líneas de código (campos privados, constructor, `getNombre()`, `equals()`, `hashCode()`, `toString()`). Hoy es una línea.
- **`var`**: Java deduce que *tareas* es una lista. No hace falta repetir el tipo mil veces.
- **`Thread.startVirtualThread`**: Es la joya de la corona. Puedes lanzar un millón de estos y tu ordenador ni se enterará. Antes, cada hilo consumía mucha RAM del sistema operativo.
- **`switch` con `->`**: Ya no necesitas poner `break;` en cada línea (¿recuerdas el error clásico de que se ejecutaran todos los casos por olvidar un `break`? ¡Ya no pasa!).

Tu primer reto *en directo*:

Si entras en un editor *online*, intenta cambiar el valor de prioridad en el *List.of* o añade una nueva *Tarea*. Verás lo fácil que es leer y modificar el código actual comparado con el de hace 20 años.

3. Utilizando `GitHub`

1. **GitHub**: Tu historial y tu escaparate

   No vea a **GitHub** solo como una *nube para guardar código*. Úsalo como tu **máquina del tiempo** y tu portafolio profesional.

   - **El flujo moderno**:
     No subas archivos sueltos. Usa el `terminal` (o `IntelliJ`) para hacer `git init`, `git add .`, `git commit` y `git push`
   - **GitHub Actions**:
     Es lo más importante hoy. Puede configurar que, cada vez que suba código, **GitHub** pase los tests automáticamente y verifique que todo compila. ¡*Es como tener un robot revisor gratis*!

2. **Docker**: "*En mi máquina funciona*" (Se acabó el problema)

   Antes, pasarle un programa Java a un amigo era un lío de versiones de Java, variables `PATH` y librerías faltantes. Con **Docker**, empaquetas tu programa con todo lo que necesita para correr.

   El *Dockerfile* para tu código de Java 21

   Para el código que hicimos antes, solo necesitarías crear un archivo llamado *Dockerfile* en la misma carpeta con este contenido:

   ```
   # 1. Usamos una imagen ligera de Java 21 (Eclipse Temurin es de las mejores)
   FROM eclipse-temurin:21-jdk-alpine

   # 2. Copiamos nuestro código al contenedor
   COPY Java2026.java /app/Java2026.java
   WORKDIR /app

   # 3. Compilamos y ejecutamos
   RUN javac Java2026.java
   CMD ["java", "Java2026"]
   ```

   ¿Por qué esto es genial? Porque cualquiera que tenga **Docker** puede ejecutar tu código con un solo comando, sin instalar Java, ni configurar nada.

3. **El Flujo de Trabajo *Pro*** (Paso a Paso)

   Si quieres aprender *en directo*, este es el orden que siguen los desarrolladores hoy:

   - **Escribes el código** en tu ordenador (con **IntelliJ IDEA**).
   - **Creas un Repositorio en GitHub** y subes tu código. Esto te da seguridad de que no perderás nada.
   - **Creas el Dockerfile** para que tu aplicación sea *portátil*.
   - **Publicas la Imagen**: Puedes subir tu *contenedor* a **Docker Hub** o **GitHub Packages**. Así, tu App puede correr en un servidor en la nube (AWS, Google Cloud) en segundos.

3.1. El Trío Dinámico

1. **Instalación**

   Para que tú y tus amigos hablen el mismo idioma, todos deben tener estas tres herramientas:

   - **Git**: Para el control de versiones (tu *máquina del tiempo*).
   - **Docker Desktop**: Para que el código corra igual en Windows, Mac o Linux.
   - **IntelliJ IDEA** (**Community**): El editor que te hará la vida fácil.

2. **Configura tu *Contenedor*** (El ambiente portátil)

   Imagina que quieres que un amigo pruebe tu código sin que él tenga que instalar Java 21. Solo crea un archivo llamado Dockerfile en la carpeta de tu proyecto:

```
# Usamos la imagen oficial de Java 21
FROM eclipse-temurin:21-jdk

# Creamos una carpeta dentro del "contenedor"
WORKDIR /app

# Copiamos todo nuestro proyecto adentro
COPY . .

# Compilamos (asumiendo que tu archivo se llama Main.java)
RUN javac Main.java

# El comando que se ejecuta al encender el contenedor
CMD ["java", "Main"]
```

   Para compartirlo: Solo diles a tus amigos: *Descarga Docker, abre la terminal en mi carpeta y escribe docker build -t mi-java-app . y luego docker run mi-java-app*. ¡Magia! Tu Java corre en su PC sin que ellos instalen Java.

3. Conecta **IntelliJ** con **GitHub** (Sin comandos raros)

   Ya no hace falta ser un experto en la consola de comandos. **IntelliJ** lo hace por ti:

   - Abre tu proyecto en **IntelliJ**.
   - Ve al menú superior: **VCS >Share Project** on **GitHub**.
   - Loguéate con tu cuenta y ¡listo! Tu código ya es público (o privado).
   - Para tus amigos: Ellos solo tienen que ir a **GitHub**, copiar la URL de tu proyecto y en su **IntelliJ** dar clic en **File >New >Project from Version Control**.

4. El toque Maestro: **.gitignore**

   Cuando compartas tu proyecto, no quieres compartir basura (archivos temporales del sistema, configuraciones locales, etc.). Crea un archivo llamado **.gitignore** y pega esto:

   ```
   *.class
   .idea/
   *.jar
   out/
   ```

   Esto le dice a Git: *Solo guarda mi código fuente, no la basura que genera el compilador*. Así tu repositorio se mantiene limpio y profesional.

5. El archivo **pom.xml**

   Para que tu proyecto sea reproducible por cualquier amigo o colega, necesitas este archivo en la raíz de tu carpeta. Es la *receta* de tu proyecto.

   - Crea un archivo llamado **pom.xml** (Project Object Model).
   - Pega esto (versión simplificada para Java 21/25):

     ```
     <project xmlns="http://maven.apache.org/POM/4.0.0">
         <modelVersion>4.0.0</modelVersion>
         <groupId>com.tuusuario</groupId>
         <artifactId>mi-primer-java-2026</artifactId>
         <version>1.0-SNAPSHOT</version>

         <properties>
             <maven.compiler.source>21</maven.compiler.source>
             <maven.compiler.target>21</maven.compiler.target>
         </properties>

         <dependencies>
             </dependencies>
     </project>
     ```

   Tu Check-list de *Salto al Vacío*

   Si quieres empezar hoy mismo, este es tu orden de batalla:

   a) Instala IntelliJ IDEA Community. (Es el *avión* donde vas a pilotar).

   b) Crea un *New Project* y selecciona Maven como sistema de construcción. Asegúrate de elegir JDK 21 (o superior).

   c) Copia el código que te pasé antes en la carpeta src/main/java.

   d) Haz tu primer *Commit*: En IntelliJ, pulsa Ctrl+K (o Cmd+K), escribe *Mi primer commit* y dale a Commit and Push hacia GitHub.

   e) Prueba Docker: Abre la terminal en esa carpeta y escribe:

```
docker build -t mi-app .
docker run mi-app
```

4. Tres Consejos de Oro para tu primera sesión

1. **No instales cualquier Java**

   Busca específicamente el JDK 21 (LTS) o el JDK 25. Son las versiones con soporte a largo plazo y las que tienen todas las funciones modernas que hablamos. (Recomiendo la distribución Eclipse Temurin o Amazon Corretto).

2. **IntelliJ es tu mejor profesor**

   Cuando escribas código, si ves una línea subrayada en amarillo, pon el ratón encima y presiona Alt + Enter. El IDE te dirá: *Oye, esto se puede escribir de forma más moderna con un Record o un Switch*. **Hazle caso**, es la forma más rápida de aprender la sintaxis actual.

3. **Rompe el código**

   Una vez que logres que el ejemplo del punto 2 funcione en **Docker**, intenta cambiar algo. Añade una nueva variable al `record`, o intenta que el `switch` devuelva un Emoji. Es ahí donde realmente ocurre el aprendizaje.

5. Requerimiento de hardware

5.1. El Puesto de Trabajo (Hardware)

Para que un estudiante de ingeniería no pierda el tiempo esperando a que el código compile o que el contenedor de **Docker** arranque, estas son las especificaciones *base*:

- **Procesador (CPU)**: **Mínimo 8 núcleos** (Ej: Intel i7 o AMD Ryzen 7 de última generación, o chips Apple M3/M4). Los hilos virtuales de Java brillan cuando el procesador puede gestionar múltiples contextos.
- **Memoria RAM** (El componente más crítico): *** Mínimo: 16 GB**.
  - **Recomendado** (Ideal 2026): **32 GB**. Docker y los microservicios modernos consumen mucha memoria al levantar bases de datos y servicios en paralelo.
- **Almacenamiento**: **SSD NVMe de al menos 512 GB**. La velocidad de lectura/escritura es vital para las dependencias de Maven y las capas de imágenes de Docker.
- **Monitor**: **Dual-monitor** o un monitor **UltraWide**. Programar en Java requiere ver mucho código, documentación y la terminal al mismo tiempo.

5.2. Infraestructura de Servidores (Para la Universidad)

En lugar de que cada alumno dependa solo de su laptop, la facultad debería ofrecer:

- **Registro de Contenedores Privado**: Un servidor (puede ser **Harbor** o **GitHub Packages**) donde los alumnos suban sus imágenes de **Docker**.
- **Nube Educativa (Self-hosted)**: Un clúster pequeño de **Kubernetes** (o **Proxmox**) para que los alumnos desplieguen sus proyectos finales. No hay nada más educativo que ver tu código Java corriendo en un clúster real.

6. El *Stack* de Software (El equipamiento lógico)

Todo el software que menciono es estándar de industria y, en su mayoría, gratuito para educación:

Herramienta	Función	Por qué en 2026
IntelliJ IDEA Ultimate	IDE principal	Las universidades suelen tener licencias gratuitas. Es el mejor para Java.
SDKMAN!	Gestor de versiones	Permite cambiar entre Java 21, 23 o 25 con un solo comando.
Docker Desktop / OrbStack	Contenedores	Para levantar bases de datos (PostgreSQL, Redis) sin instalarlas en el SO.
Postman / Insomnia	Pruebas de API	Para testear los microservicios que creen los alumnos.
GitHub Enterprise/Education	Colaboración	Para manejar el flujo de trabajo (Pull Requests, Code Reviews).

6.1. El *Laboratorio Portátil* (Presupuesto Cero)

Si la universidad no puede renovar el hardware, la tendencia en 2026 es el uso de **Cloud IDEs**:

- **GitHub Codespaces**: Permite que el alumno programe en un entorno potente en la nube desde cualquier navegador (incluso desde una tablet).
- **Dev Containers**: Configuras un archivo en el proyecto y, al abrirlo, el entorno instala automáticamente Java, Maven y todo lo necesario dentro de un contenedor. **Cero configuración manual**.

6.2. El Lenguaje (JDK): Software Libre

Aún cuando **Oracle** ofrece su **JDK**, la comunidad de ingeniería **prefiere** las distribuciones **OpenJDK**.

- **Recomendación**:

  Eclipse Temurin (de la Fundación Eclipse) o *Amazon Corretto*.

- **Por qué**:

  Son gratuitas, tienen licencias permisivas para uso comercial y son las que los alumnos encontrarán en el 90% de las empresas que usan **Docker** y **Kubernetes**.

6.3. El IDE (Editor): El dilema Licenciado vs. Libre

Aquí es donde hay más debate:

- **La opción "Pro" (Licenciada)**: **IntelliJ IDEA Ultimate**. Es el estándar de oro. Para universidades, **JetBrains** ofrece licencias gratuitas para estudiantes y profesores. **Un consejo**: usa esta vía. Es software licenciado, pero al ser gratis para educación, le das al alumno la mejor herramienta del mercado.
- **La opción Libre**: **VS Code** (con el **Extension Pack for Java**) o **Eclipse IDE**. Aunque son potentes, no alcanzan el nivel de análisis de código de **IntelliJ** para Java moderno.

6.4. Infraestructura y Despliegue: Software Libre (Obligatorio)

En Ingeniería Informática, enseñar herramientas propietarias de despliegue es *encadenar* al alumno a una marca.

- **Contenedores**: Usa **Docker Engine** o **Podman** (100% libre y excelente alternativa a **Docker Desktop**).
- **Orquestación**: **Kubernetes** (**K3s** para laboratorios ligeros).
- **Bases de Datos**: **PostgreSQL** o **MariaDB**. Enseñar con bases de datos licenciadas (como **Oracle DB** o **SQL Server**) es cada vez menos común en currículos modernos de **startups** y **Big Tech**.

Comparativa: ¿Por qué elegir Software Libre en la Universidad?

Criterio	**Software Libre (OSS)**	**Software Licenciado (Propietario)**
Costo	$0 (Inversión en conocimiento)	Costo recurrente por asiento.
Soberanía	El alumno puede llevarse su entorno a casa.	Depende de la conexión al servidor de licencias.
Transparencia	Puedes leer el código fuente de la librería.	Es una *caja negra*.
Mercado Laboral	Es el estándar en la cultura **Cloud Native**.	Limitado a entornos corporativos legacy.

6.5. Mi recomendación para el Stack 2026

Para un laboratorio de alto nivel:

- **S.O.**: **Linux** (**Ubuntu** o **Fedora**). Es libre y es donde **Java** y **Docker** corren de forma nativa. Aprender **Java** en **Windows** es como aprender a conducir un Ferrari en un estacionamiento.
- **JDK**: **Eclipse Temurin 21/25** (Libre).
- **Gestor de Dependencias**: **Maven** (Libre).
- **IDE**: **IntelliJ IDEA Ultimate** (Licencia educativa gratuita) como primera opción, y **VS Code** como alternativa libre.
- **Control de Versiones**: **Git** con **GitHub** (Modelo *freemium* muy potente para educación).

6.6. El valor educativo del Software Libre

Al usar Software Libre, el estudiante aprende que puede contribuir. Si encuentra un error en una librería de Java, puede ir a GitHub, ver el código y proponer una solución. Eso es lo que define a un Ingeniero de élite.

7. Manifiesto: Ingeniería de Java + IA1

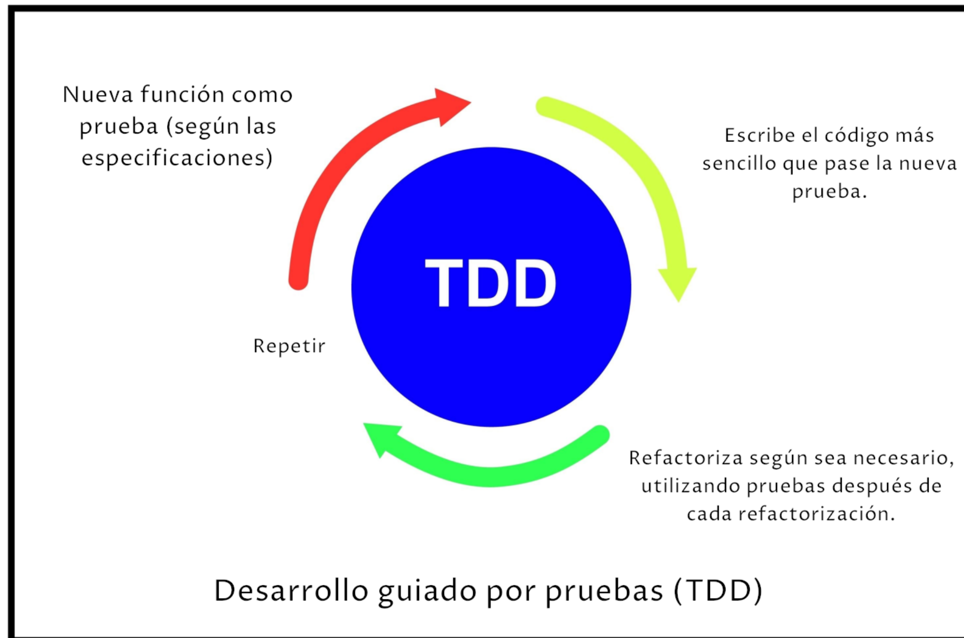
7.1. La IA es el copiloto, Tú eres el Capitán

- **La Regla**: Nunca aceptes una sugerencia de la IA (**GitHub Copilot**, **ChatGPT**, etc.) **que no seas capaz de explicar línea por línea**.
- **Práctica en 2026**: Usa la IA para generar el "esqueleto" (*boilerplate*), pero tú debes definir la lógica de negocio y las restricciones de seguridad.

7.2. Validar con Tests Unitarios (TDD con IA)

Recuerde: **La IA es propensa a inventar métodos o ignorar casos borde**

- **La Regla**: Pide a la IA que genere los tests antes o al mismo tiempo que el código.
- **Práctica**: Si la IA escribe un método, dile: "*Ahora genera 5 casos de prueba usando **JUnit 5**, incluyendo valores nulos y listas vacías*". Si el código pasa los tests, es confiable.



7.3. *Prompt Engineering* enfocado en Arquitectura

No pidas solo "*haz un programa que...*". Sé específico sobre el estándar de 2026.

- **Prompt Incorrecto**: *Hazme una clase Libro en Java.*
- **Prompt Pro**: "*Genera un **Record** inmutable en Java 21 para la entidad Libro, usa **Validation API** para que el título no sea nulo y asegúrate de que sea compatible con la serialización JSON de Jackson*".

7.4. Revisión de Deuda Técnica y Seguridad

Usa la IA como un auditor de calidad, no solo como un escritor.

- **La Regla**: Antes de hacer un *commit*, pega tu código en la IA y pregunta: ¿*Cómo puedo hacer este código más eficiente usando Streams de Java 21*? o ¿*Hay alguna vulnerabilidad de seguridad en este método*?.

7.5. Documentación Inteligente (JavaDoc)

En 2005, escribir documentación era tedioso. En 2026, es imperdonable no tenerla.

- **Práctica**: Pide a la IA que genere los comentarios en formato **JavaDoc** basándose en el código. Un buen ingeniero entrega código que otros humanos (y otras IAs) pueden entender fácilmente.

Herramientas de IA que deben estar en el Laboratorio

Herramienta	Uso Sugerido
GitHub Copilot / Cursor	Autocompletado inteligente dentro de **IntelliJ**.
JetBrains AI Assistant	Integración nativa con el IDE para refactorizar código antiguo a Java 21.
SonarQube (con IA)	Para analizar automáticamente si el código cumple con las reglas de *Clean Code*.

8. Hoja de Comandos Rápidos

Estos son comandos rápidos para dejar su estación de trabajo lista para la acción en 2026.

Si usas **Windows**, te recomiendo instalar primero **WSL2** (*Windows Subsystem for Linux*), pero estos comandos funcionarán en cualquier terminal moderna.

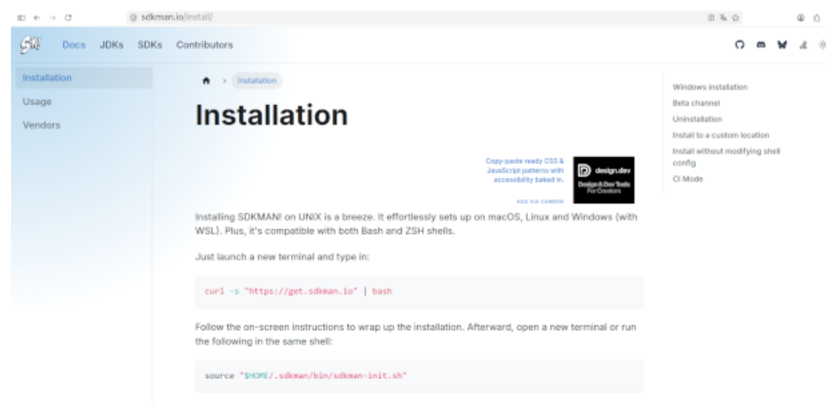
8.1. Gestión de Versiones (La forma profesional)

No instales **Java** descargando un `.exe` o `.pkg`. Usa **SDKMAN!**, que es el estándar para ingenieros. Le permite saltar entre versiones de **Java** en segundos.

- **Instalar SDKMAN!** (en Linux/Mac/WSL):

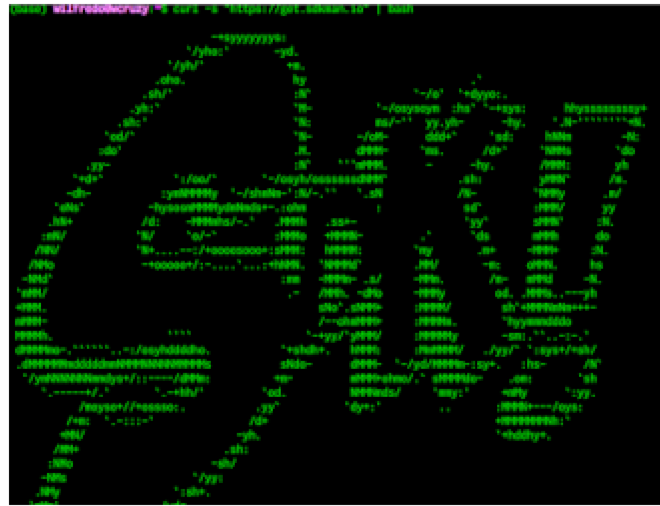
  1. Ingrese a **SDKman**:
     https://sdkman.io/

Para instalar **Java**, hacemos clic en **Docs** (https://sdkman.io/install/)



2. Abrimos una ventana de terminal en Linux o PowerShell en Windows, y escribimos:

```
curl -s "https://get.sdkman.io" | bash
```



```
Looking for a previous installation of SDKMAN...
Checking Bash version...
Looking for unzip...
Looking for zip...
Looking for tar...
Looking for curl...
Looking for sed...
Installing SDKMAN scripts...
Create distribution directories...
Getting available candidates...
Prime platform file...
Prime the config file...
Installing script cli archive...
* Downloading...
######################################################################## 100,0%
* Checking archive integrity...
* Extracting archive...
* Copying archive contents...
* Cleaning up...

Installing script cli archive...
* Downloading...
######################################################################## 100,0%
* Checking archive integrity...
* Extracting archive...
* Copying archive contents...
* Cleaning up...

Set version to 5.22.4 ...
Set native version to 0.7.32 ...
Attempt update of interactive bash profile on regular UNIX...
Added sdkman init snippet to /home/wilfredo/.bashrc
Attempt update of zsh profile...
Updated existing /home/wilfredo/.zshrc


All done!


You are subscribed to the STABLE channel.

Please open a new terminal, or run the following in the existing one:

    source "/home/wilfredo/.sdkman/bin/sdkman-init.sh"

Then issue the following command:

    sdk help

Enjoy!!!
(base) wilfredo@wcruzy:~$
```

Y para que se aplique al sistema ejecutamos:

```
source "$HOME/.sdkman/bin/sdkman-init.sh"
```

```
(base) wilfredo@wcruzy:~$ source "$HOME/.sdkman/bin/sdkman-init.sh"
(base) wilfredo@wcruzy:~$
```

3. Para verificar que la instalación está completa, ejecutamos en la terminal:

   ```
   sdk version
   ```

   ```
   (base) wilfredo@wcruzy:~$ sdk version

   SDKMAN!
   script: 5.22.4
   native: 0.7.32 (linux x86_64)

   (base) wilfredo@wcruzy:~$
   ```

4. Para instalar las versiones de Java que requiera, por ejemplo, **Java 17** y **Java 21**, ejecute primero

   ```
   sdk list java
   ```

   para listar las versiones disponibles de **Java** que tiene **SDKman**

   ```
   ================================================================================
   Available Java Versions for Linux 64bit
   ================================================================================
    Vendor        | Use | Version      | Dist    | Status     | Identifier
   --------------------------------------------------------------------------------
    Corretto      |     | 26           | amzn    |            | 26-amzn
                  |     | 25.0.2       | amzn    |            | 25.0.2-amzn
                  |     | 24.0.2       | amzn    |            | 24.0.2-amzn
                  |     | 23.0.2       | amzn    |            | 23.0.2-amzn
                  |     | 21.0.10      | amzn    |            | 21.0.10-amzn
                  |     | 17.0.18      | amzn    |            | 17.0.18-amzn
                  |     | 11.0.30      | amzn    |            | 11.0.30-amzn
                  |     | 8.0.472      | amzn    |            | 8.0.472-amzn
    Dragonwell    |     | 21.0.10      | albba   |            | 21.0.10-albba
                  |     | 17.0.18      | albba   |            | 17.0.18-albba
                  |     | 11.0.30      | albba   |            | 11.0.30-albba
                  |     | 8.0.482      | albba   |            | 8.0.482-albba
    Gluon         |     | 22.1.0.1.r17 | gln     |            | 22.1.0.1.r17-gln
                  |     | 22.1.0.1.r11 | gln     |            | 22.1.0.1.r11-gln
    GraalVM CE    |     | 25.0.2       | graalce |            | 25.0.2-graalce
                  |     | 24.0.2       | graalce |            | 24.0.2-graalce
                  |     | 23.0.2       | graalce |            | 23.0.2-graalce
                  |     | 21.0.2       | graalce |            | 21.0.2-graalce
                  |     | 17.0.9       | graalce |            | 17.0.9-graalce
    GraalVM Oracle|     | 26.ea.13     | graal   |            | 26.ea.13-graal
                  |     | 25.0.2       | graal   |            | 25.0.2-graal
                  |     | 24.0.2       | graal   |            | 24.0.2-graal
                  |     | 23.0.2       | graal   |            | 23.0.2-graal
                  |     | 21.0.10      | graal   |            | 21.0.10-graal
                  |     | 17.0.12      | graal   |            | 17.0.12-graal
    Huawei        |     | 21.0.10      | bisheng |            | 21.0.10-bisheng
                  |     | 17.0.18      | bisheng |            | 17.0.18-bisheng
                  |     | 11.0.30      | bisheng |            | 11.0.30-bisheng
                  |     | 8.0.482      | bisheng |            | 8.0.482-bisheng
   :
   ```

5. Nos fijamos en la columna **Vendor** (*Fabricante*), **Version** e **Identifier** para seleccionar la versión de Java que requerimos según el fabricante.

   Si queremos la versión **17.0.15 de Corretto**, por ejemplo, debemos fijarnos en el identificador: **17.0.15-amzn**

   Escribimos en la terminal:

```
sdk install java 17.0.15-amzn
```

```
(base) wilfredo@wcruzy:~$ sdk install java 17.0.15-amzn

Downloading: java 17.0.15-amzn

In progress...

############################################################ 100,0%

Repackaging Java 17.0.15-amzn...

Done repackaging...

Installing: java 17.0.15-amzn
Done installing!


Setting java 17.0.15-amzn as default.
(base) wilfredo@wcruzy:~$
```

Si el la **única** versión de Java, SDK la instalará como versión de Java por defecto. Pero si tenemos otra versión de Java y queremos que esta última sea la versión por **defecto**, escribimos:

```
sdk default java 17.0.15-amzn
```

Para verificar la instalación, escribimos:

```
java --version
```

```
(base) wilfredo@wcruzy:~$ java --version
java 17.0.6 2023-01-17 LTS
Java(TM) SE Runtime Environment (build 17.0.6+9-LTS-190)
Java HotSpot(TM) 64-Bit Server VM (build 17.0.6+9-LTS-190, mixed mode, sharing)
(base) wilfredo@wcruzy:~$
```

6. Para instalar **Java 21 de Temurin** más reciente, ejecutamos:

```
sdl java list
```

Y vamos a **Temurin** y nos fijamos en el **Identifier** de la más reciente versión 21 (la **21.0.10-tem**). Escribimos:

```
sdk install java 21.0.10-tem
```

```
(base) wilfredo@wcruzy:~$ sdk install java 21.0.10-tem

Downloading: java 21.0.10-tem

In progress...

############################################################ 100,0%

Repackaging Java 21.0.10-tem...

Done repackaging...

Installing: java 21.0.10-tem
Done installing!


Setting java 21.0.10-tem as default.
(base) wilfredo@wcruzy:~$
```

Podemos tener **una terminar ejecutando Java 17 por defecto** y abrimos otra terminal donde podamos ejecutar **Java 21**, para ello, abrimos una nueva terminal y escribimos:

```
sdk use java 21.0.10-tem
```

Vídeo:

Cómo Instalar SDKMAN y Usar Múltiples Versiones de Java (Java 17 y Java 21)

8.2. Comandos Esenciales de Docker

Para que el *ambiente portátil* funcione, estos son los tres comandos que deben tener en cuenta:

- **Construir la imagen** (estando dentro de la carpeta del proyecto):

```
docker build -t mi-proyecto-java .
```

- **Correr el contenedor**:

```
docker run --rm mi-proyecto-java
```

- **Limpiar imágenes antiguas** (para no llenar el disco duro del laboratorio):

```
docker system prune -f
```

8.3. Git: El flujo de trabajo en 2026

Antes de subir nada, asegúrate de que el autor está bien identificado (útil para calificar tareas):

- **Configurar identidad**:

```
git config --global user.name "Tu Nombre"
git config --global user.email "tu@email.com"
```

- **El ciclo de vida del código**:

```
git init                              # Inicia el repositorio
git add .                             # Prepara los cambios
git commit -m "feat: mi record" # Guarda con mensaje descriptivo
git push origin main                  # Sube a GitHub
```

8.4. Prueba de Fuego

Si quieres saber si tu **Java 21** está realmente bien instalado y soporta **Virtual Threads**, ejecuta este comando directamente en tu terminal:

```
jshell --button
```

(Esto abrirá la consola interactiva de Java.) Pega esto:

```
Thread.startVirtualThread(() -> System.out.println("¡Hola desde el futuro (2026)!"));
```

Si ves el mensaje, ¡tu equipo está oficialmente en el siglo XXI!