

Apunte de Clase 2 - Java

En el presente apunte vamos a abordar dos temas clave en la introducción de la asignatura. Por un lado vamos realizar una revisión de los elementos fundantes de la Plataforma Java, vamos a realizar una breve recopilación histórica de cómo Java llegó a ser una de las plataformas de desarrollo de software más utilizadas del mundo hoy en día, enumeraremos las versiones o ediciones disponibles de java y describiremos brevemente su función en la plataforma.

En segundo lugar vamos a desarrollar un [HolaMundo](#) con Java, pero no con el objetivo de realizar un análisis del código requerido que vamos caracterizar brevemente, sino con el objetivo de comprender los componentes necesarios para programar en java, qué debemos instalar en nuestra máquina para desarrollar y qué función cumplen estos instaladores.

Introducción a la Plataforma Java

Breve referencia histórica de la Plataforma Java

A comienzos de la década de 1990, el lenguaje C++ era el preferido por los desarrolladores para la creación de aplicaciones. Este lenguaje (cuya especificación fue definida por B. Stroustrup) reunía la potencia del C estándar con la programación orientada a objetos, y aunque no fue el primer lenguaje de objetos, fue el primero en ser usado para desarrollos profesionales.

Hacia 1991 un equipo de ingenieros de la firma Sun Microsystems generaron un proyecto llamado de The Green Project, liderado por James Gosling y Patrick Naughton, este equipo comenzó a trabajar en el diseño de un lenguaje de programación que pudiera usarse para programar dispositivos electrodomésticos. Ese lenguaje debía tener la capacidad de adaptarse a distintos tipos de dispositivos y procesadores, siendo capaz de generar programas que pudieran correr en cualquier tipo de dispositivo que soportara al lenguaje. Originalmente ese lenguaje se llamó *7 (léase: star seven), pero ese nombre no fue muy bien recibido por la comunidad de programadores y se propuso cambiarlo al nombre Oak por un roble que había fuera de la oficina.

Sin embargo, ya existía una marca de software con ese nombre por lo que debió ser cambiado otra vez y luego de pasar por el nombre Green (del proyecto en el que estaban) terminó llamándose Java. Se dice que algunos miembros del equipo de trabajo de Gosling y Naughton estaban tomando un café discutiendo sobre el nuevo nombre del lenguaje, y alguien notó que estaban tomando "café de la cafetería Java". Sin más, se propuso el nombre Java para el nuevo lenguaje, y eso explica también por qué el logotipo del lenguaje es una taza de café humeante (Aunque no hay fundamento real acerca de esta creencia).

Por los motivos ya indicados, el lenguaje Java se diseñó para poder generar programas que sean portables de una plataforma a otra. Vale decir: un programa desarrollado en un tipo de computadora con cierto tipo de sistema operativo debería poder llevarse a otro tipo de computadora con un sistema operativo diferente, sin tener que cambiar el programa y sin tener que volver a compilar - vamos a volver sobre este tema cuando hagamos el *Hola Mundo*. La idea era buena y el lenguaje tenía elementos que lo hacían muy flexible y

poderoso. Entre sus características, se decidió que sea basado en objetos, y que su núcleo de instrucciones fuera similar al de C++ en la mayor medida posible, para facilitar la migración de programadores de ese lenguaje hacia Java. Sin embargo, el mercado no estaba listo para un lenguaje con esas características: aún no era masiva la cantidad de electrodomésticos programables disponibles, ni la necesidad de contar con un lenguaje avanzado para programarlos.

También en 1991, la Internet fue liberada para su uso comercial y eso provocó un cambio profundo en las estrategias de comercialización, transmisión de datos, comunicaciones y (por supuesto) en el diseño y desarrollo de sistemas informáticos. Hasta ese momento Internet era usada sólo en ámbitos académicos y gubernamentales de los Estados Unidos, y era controlada por la Fundación Nacional de Ciencia (NSF: National Science Foundation). Internet ofrece numerosos servicios, entre los cuales se encuentra la World Wide Web (o simplemente, la web) a través de la cual todas las computadoras enlazadas a la red pueden acceder a información gratuita sobre prácticamente cualquier tema que haya sido publicado, hoy en día, incluso sobre cualquier dispositivo conectado como lo son centrales meteorológicas, o sensores sísmicos o infinidad de dispositivos de lo que se denomina Internet de las Cosas o IoT por su sigla en inglés.

Con el dramático incremento en el uso de la Internet, pronto se vio que Java podría ser muy útil para programación de páginas web más atractivas que las que hasta ese momento se producían. El lenguaje HTML de las páginas web no es un lenguaje de programación sino un lenguaje de formateo de texto, y no se podía hacer con HTML que una página incluyera gráficas o animaciones sofisticadas, ni procesamiento de datos. Pero con Java podían crearse pequeños programas llamados applets, los cuales podrían incluirse dentro de una página web y descargarse en la computadora del usuario. Como Java es portable, no importaba si ese applet se desarrollaba en un contexto Windows y luego se descargaba en una computadora con Linux (o viceversa): el applet se ejecutaría sin problemas si el navegador web usado soporte Java, hoy en día esa batalla se perdió a manos de Javascript pero los primeros juegos en línea se programaron como Applets Java. En aquel momento mediante applets, un programador web adquiría mayor flexibilidad en la programación y muchos más elementos gráficos que con HTML, y Java se relanzó como lenguaje adquiriendo de inmediato un éxito rotundo.

Con el tiempo, surgieron otros lenguajes capaces de introducir programación compleja en una página web. Los lenguajes de script (como JavaScript o Visual Basic Script) así como Flash, son un claro ejemplo de ello. Pero Java no era (ni es) un simple lenguaje para programación de applets, aunque les deba su popularidad inicial a los applets. Java es un lenguaje multipropósito: permite desarrollar sistemas de amplia gama y hoy en día es muy utilizado para programación de aplicaciones de negocios (soluciones informáticas integrales para la gestión de las actividades de una empresa de cualquier magnitud).

Estas aplicaciones son independientes de sus interfaces, es decir pueden ser consumidas por distintos tipos de interfaces de usuario, están alojadas en servidores para ser accedidas de manera remota y se conectan con todo tipo de sistemas en infinidad de tecnologías distintas, es a esto a lo que vamos a llamar backend y para lo que Java es uno de los principales exponentes del mercado en la actualidad.

Versiónes de Java

Introducción al esquema de versionado

Desde su lanzamiento en 1996, Java evolucionó a través de muchas versiones. Cada una de ellas incorporaba soluciones a problemas de la versión anterior, mejoras, y también librerías o paquetes de clases que no necesariamente eran desarrolladas desde Sun Microsystems sino también por terceras organizaciones

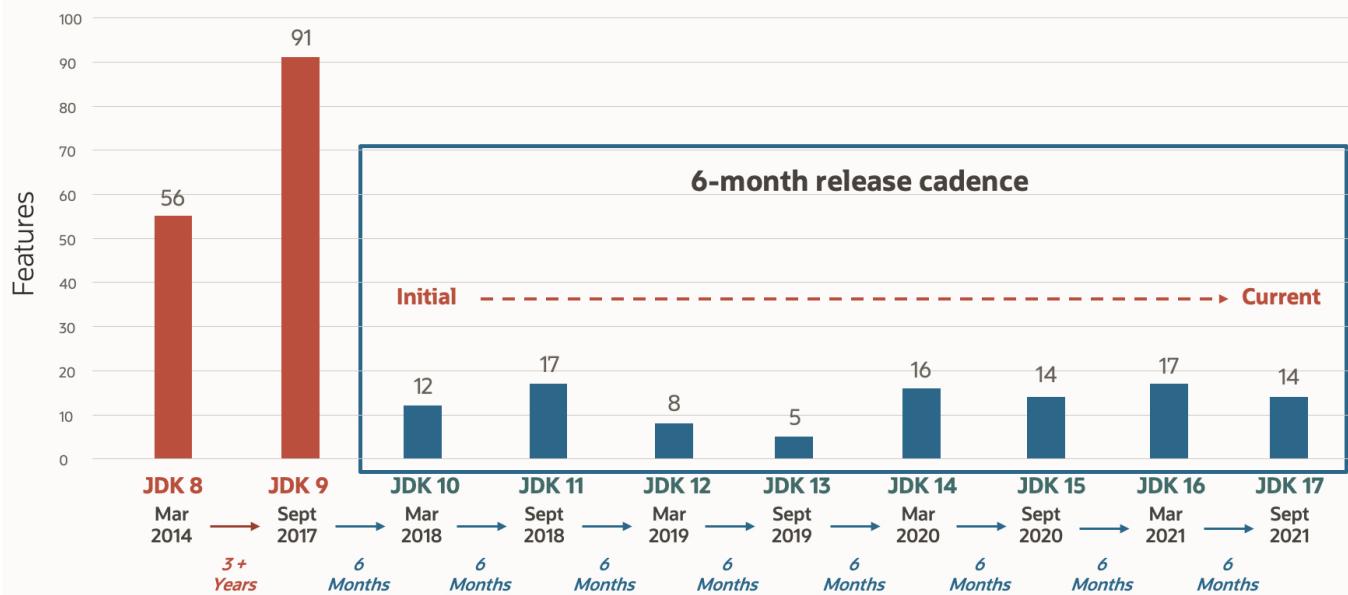
específicamente dedicadas a producir esas librerías (también llamadas APIs por Application Programming Interface o Interfaz de Programación de Aplicaciones) para Java. Cada versión de Java se identifica históricamente con números separados por puntos. Así que, si bien por allá por 1991 ya había algunos programas Java en ejecución, la primera versión de Java es de 1996 fue la versión Java 1.0 y hacia 1997 se lanzó la versión Java 1.1 que incluía una API (llamada Abstract Windowing Toolkit o AWT) para el desarrollo de interfaces de usuario basadas en ventanas. En 1998 lanzó la versión Java 1.2 agregaba una nueva y muy potente API (conocida como Swing, que era el nombre código del equipo de desarrollo que la diseñó) para el desarrollo de interfaces de usuario y muchas otras características que instalaron a Java definitivamente como uno de los lenguajes líderes para el desarrollo de aplicaciones de negocio.

A partir de esta versión comenzó a usarse el nombre general **Java 2**, específicamente **J2** para referirse a las distintas ediciones de Java 1.2 y a las subsiguientes versiones que aparecerán después: Java 1.3 en el año 2000 y Java 1.4 en el año 2002. De hecho, a partir de entonces el conjunto de herramientas para programación Java disponible desde la versión Java 2, se conoce también como J2SE (abreviatura de Java 2 Standard Edition). Existe aún hoy gran cantidad de documentación en Internet que hace referencia a Java desde J2XX donde las 'X' hacen referencia a la edición en cuestión. En el año 2004 apareció Java 5.0 (note que la versión comercial ya no se designa como Java 1.5 sino directamente como Java 5.0, aunque internamente el nombre Java 1.5 se usaba todavía. Quedó en la anécdota del marketing mundial el hecho que no haya existido una versión oficialmente llamada Java 3 ni una llamada Java 4...) Esta versión Java 5.0 es realmente una nueva versión del lenguaje y no una corrección o conjunto de mejoras a la versión anterior. La versión Java 5.0 incorpora elementos que cambian el núcleo del lenguaje, tales como clases genéricas, enumeraciones y una variante mejorada del ciclo for.

Aquí habría que abrir un importante debate acerca del concepto retro-compatibilidad hacia atrás que Java ha mantenido a lo largo de toda su historia y que trae dolores de cabeza a la comunidad por volverse el impedimento de avance y mejora continua de la plataforma, dejamos este punto para lectura de cada estudiante y debate grupal.

En diciembre de 2006 apareció Java 6 (sí, Java 6 y NO Java 6.0) imponiendo aún nuevos cambios y mejoras. A partir de esta versión, se elimina la designación J2SE y se cambia directamente por Java SE (lo cual era una necesidad obvia desde la versión Java 5.0) Entre los años 2009 y 2010, Sun fue adquirida por la empresa Oracle, y en 2011 apareció la versión Java SE 7. Luego también existió la versión 8, Java SE 8 con su nuevo conjunto de funcionalidades.

La versión 9 de java, Java SE 9 fue la última versión de java como originalmente se conoció, luego del cambio a manos de Oracle la última versión con el esquema desde la Java 5 es la 7, a partir de dicha versión y en sintonía con una gran cantidad de productos de software se pasó al esquema de versiones con soporte de largo término o LTS por su sigla en inglés y versiones de actualizaciones periódicas.



Luego del cambio a cadencia de 6 meses siguieron las versiones 10 y 11 esta última, versión LTS, tuvo un importante cambio pues pasó a ser Licenciada por Oracle y genero un gran movimiento de la comunidad para proponer máquinas virtuales alternativas que ya se venían desarrollando y que mantuvieron el concepto de uso libre, que no es lo mismo que Open Source o código abierto, sin embargo entre todas ellas la propia Oracle junto a Red Hat esponsporearon a OpenJDK que es quien mantienen la actualización de las versiones libres de Java.

Versión actual de Java (este apartado va a estar siempre desactualizado 😊)

Así la última versión LTS de java es la versión Java SE 21 que al ser una versión LTS garantiza el soporte hasta Septiembre de 2031, ya está disponible también la versión menor 22. La versión 25 que vendrá a ser la próxima LTS tiene fecha planificada de pasar a GA en Septiembre del 2025.

No siendo esta asignatura un curso de Java no tendremos posibilidad de revisar en detalle las novedades de la versión o versiones sucesivas pero basta con una búsqueda en internet para observar el nivel de detalle de los cambios que parecen en la versión.

Notas del Release 21 del sitio de Oracle

Ediciones de Java

Ahora nos encontramos con el principal escollo a la hora de conseguir documentación de Java ya que la propia palabra Java se asocia a conceptos muy diferentes, en el siguiente apartado intentaremos al menos nombrar los principales estratos a los que se asocia la palabra Java con una breve referencia a cada uno.

Java SE (Standard Edition): La edición estándar de Java es considerada la versión core o central de java, ella incluye las herramientas necesarias para poder programar en Java y otras herramientas para empaquetar nuestros componentes Java o Documentar los mismos entre otras muchas herramientas de soporte al proceso de desarrollo. Además incluye el conjunto principal de APIs o Librerías propias del lenguaje de programación Java que van desde Colecciones hasta acceso a Base de Datos y ORM pasando por entrada/salida, redes e hilos entre otras.

Java EE (Enterprise Edition): La edición empresarial de java o Java EE se ha utilizado históricamente para desarrollar aplicaciones empresariales en Java. Sin embargo, en la actualidad, Oracle transfirió la

responsabilidad de Java EE a la Fundación Eclipse, donde se ha convertido en el proyecto Jakarta EE.

Ya Siendo Jakarta EE, se organizó en Perfiles o conjuntos de Librerías y APIs orientados a diferentes propósitos:

- **Perfil Web (Web Profile):** Este perfil incluye tecnologías principales para el desarrollo de aplicaciones web empresariales, como Servlet API, JavaServer Faces (JSF), Java Persistence API (JPA) y Contexts and Dependency Injection (CDI).
- **Perfil de Plataforma Completa (Full Platform Profile):** Este perfil proporciona todas las APIs y especificaciones de Java EE, incluyendo las del perfil web, y agrega tecnologías adicionales como Enterprise JavaBeans (EJB), Java Message Service (JMS) y Java Transaction API (JTA).
- **MicroProfile:** Desarrollado por la comunidad, MicroProfile es un conjunto de especificaciones y APIs diseñadas para facilitar el desarrollo de microservicios basados en Java. Ofrece características como la configuración externa, tolerancia a fallos, métricas y seguridad. MicroProfile ha ganado popularidad debido a su enfoque ligero y su capacidad para impulsar la arquitectura de microservicios en Java.

Java ME (Micro Edition): La edición Micro de Java originalmente se creó para la programación de dispositivos móviles, sin embargo, esta edición se enfoca actualmente en la Internet de las cosas (IoT).

Java Card: Java Card se utiliza para desarrollar aplicaciones seguras en tarjetas inteligentes y dispositivos con recursos limitados.

Java FX: Java FX es una plataforma de desarrollo de interfaces de usuario (UI) rica y moderna para aplicaciones de escritorio, web y móviles. Proporciona un conjunto de bibliotecas y herramientas para crear interfaces de usuario interactivas y atractivas en modo declarativo. Java FX se ha convertido en un proyecto de código abierto y se ha trasladado a la Fundación OpenJFX.

El lenguaje de Programación Java

Unos de los principales sino el principal elemento de Java SE es el Lenguaje de Programación Java, el lenguaje de programación en el que se escriben todos los componentes en todas las ediciones más allá de otros lenguajes utilizados para configuración, intercambio o ejecución de los programas.

Como todos los lenguajes de programación está basado en ciertas características específicas y estas lo hacen una versión un lenguaje único que ocupa un espacio que ningún otro de los lenguajes existente puede a excepción quizás de C# que puede ser el que más se asemeja.

Características de Java

Entre las principales características del Lenguaje de Programación Java debemos citar:

- **De Alto Nivel:** es un lenguaje que está más cerca de ser legible por el ser humano que de las instrucciones del procesador y por otro lado contiene una amplia gama de librerías y código ya desarrollado que permite que el programador se pueda centrar en las funcionalidades a programar e independizarse de las cuestiones de infraestructura física de las computadoras.
- **Orientado a objetos:** Al contrario de otros lenguajes como C++, Java no es un lenguaje modificado para poder trabajar con objetos, sino que es un lenguaje creado originalmente para trabajar con objetos. De hecho, todo lo que hay en Java son objetos.

- Independiente de la plataforma: Debido a que existen máquinas virtuales para diversas plataformas de hardware, el mismo código Java puede funcionar prácticamente en cualquier dispositivo para el que exista una JVM.
- Compilado e Interpretado (la JVM es más que un intérprete y NO es correcto decir que es un intérprete, pero mantiene el mismo concepto y nos permitimos invitar al estudiante a revisar las diferencias específicas): La principal característica de Java es la de ser un lenguaje compilado e interpretado. De este modo se consigue la independencia de la máquina, el código compilado se ejecuta en máquinas virtuales que sí son dependientes de la plataforma o sistema operativo. Para cada sistema operativo distintos, ya sea Microsoft Windows, Linux, macOS, existe una máquina virtual específica que permite que el mismo programa Java pueda funcionar sin necesidad de ser recompilado.
- Gestiona la memoria automáticamente (Memory Safe): La máquina virtual de Java gestiona la memoria dinámicamente como veremos más adelante. Existe un recolector de basura que se encarga de liberar la memoria ocupada por los objetos que ya no están siendo utilizados.
- Multihilos (multithreading): Soporta la creación de partes de código que podrán ser ejecutadas de forma paralela y comunicarse entre sí.

Hola Mundo en Java

El programa **Hola Mundo**, que tiene su origen en el libro "*El Lenguaje de Programación C*"** de Brian Kernighan y Dennis Ritchie, tiene por objeto mostrar los elementos necesarios para escribir programas en un lenguaje de programación, en este caso en Java. No es su objetivo el estudio del Lenguaje propiamente dicho, sino poder conocer las herramientas necesarias para llevar ese programa desde el código fuente al programa en ejecución.

A continuación, y con el mismo objeto, vamos a desarrollar el programa Hola Mundo pero en Java.

1 El código fuente

Lo primero que vamos a tener que hacer es crear un archivo de texto, cosa que podemos hacer con cualquier editor de texto. Todos los sistemas operativos incluyen al menos una herramienta de edición de archivos de texto, en mi caso podría usar directamente `vim` por línea de comandos para hacerlo. También podemos usar una herramienta con interfaz de usuario como Notepad++, Atom o Sublime Text por nombrar algunos famosos.

Entonces en un archivo de código fuente vamos a escribir el siguiente código Java:

```
public class Programa {  
    public static void main(String[] args) {  
        System.out.println("Hola Mundo");  
    }  
}
```

La primera restricción del lenguaje de programación Java es que este archivo no puede tener cualquier nombre, se **debe** llamar `Programa.java` la primera parte debe coincidir con el nombre de la clase declarada dentro y debe tener la extensión `.java` estos elementos son obligatorios para que todo lo que sigue pueda funcionar.

Si bien no vamos a hacer una revisión exhaustiva del lenguaje Java, es imposible no mencionar algunos elementos generales y distintivos de este lenguaje:

- Es un lenguaje Case Sensitive, por lo que las letras minúsculas y mayúsculas son considerados símbolos diferentes es decir **Programa** no es lo mismo que **programa**.
- Divide los bloques con llaves, es decir que utiliza **{** para indicar el inicio de un bloque de código y **}** para indicar el final del bloque, esto lo diferencia de otros lenguajes que utilizan palabras como begin y end o tabulaciones para marcar el principio y final de los bloques.
- Utiliza **obligatoriamente ;** para terminar una sentencia de código, esto lo diferencia de otros lenguajes que utilizan el salto de línea para terminar una sentencia.

Finalmente, solo mencionar que este fragmento de código el método de clase main de la clase Programa es un método especial y es considerado el punto de inicio de nuestro programa Java y por lo tanto debe ser definido estrictamente como está en el ejemplo. Más adelante veremos el significado de cada una de las palabras reservadas en el bloque y qué porción de dicha definición se puede modificar.

2 El compilador

El compilador Java es una de las herramientas provistas por Java SE y es central en la programación Java, para disponer de esta herramienta será necesaria la instalación del Kit de Desarrollo de Software Java o Java SDK (Cabe destacar que para los programadores Java se sigue llamando y se llamará siempre Java JDK por Kit de Desarrollo Java simplemente).

Más adelante haremos un breve debate acerca de las diferentes opciones de instalación del JDK pero en general vamos a recomendar instalaciones diferentes para linux y windows basados en la simplicidad de instalación y sabiendo que en el uso que vamos a dar cualquiera de las dos versiones es compatible con todo lo que vamos a hacer en la asignatura

Instalación en Linux

En el caso de linux vamos a instalar el paquete de OpenJDK 17 disponible para todas las distribuciones, por ejemplo en el caso de alguna distribución que soporte apt deberíamos ejecutar los siguientes comandos:

```
$ sudo apt update -y && sudo apt upgrade -y  
...actualiza el Linux y el administrador de paquetes apt  
  
$ sudo apt install openjdk-21-jdk -y  
... instala java openjdk 21
```

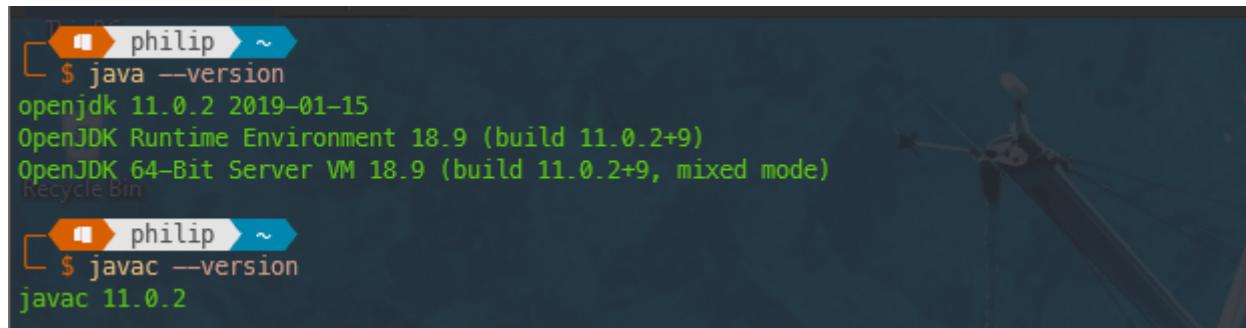
Instalación en Windows

En el caso de Windows vamos a recomendar la instalación de Oracle JDK 17 porque podemos usar un instalador no tendremos que configurar manualmente las variables de entornos (aunque siempre está la opción de descargar el zip del sitio de Open JDK y realizar la instalación manual)

[Descarga del JDK de Oracle](#)

Ejecutar el instalador y seguir los pasos.

Una vez realizada la instalación por alguno de los mecanismos expuestos, o algún otro mecanismo deseado, debería verificar que la instalación funcionó correctamente como sigue:



```
philip@hpZbook-Philip: ~
$ java --version
openjdk 11.0.2 2019-01-15
OpenJDK Runtime Environment 18.9 (build 11.0.2+9)
OpenJDK 64-Bit Server VM 18.9 (build 11.0.2+9, mixed mode)

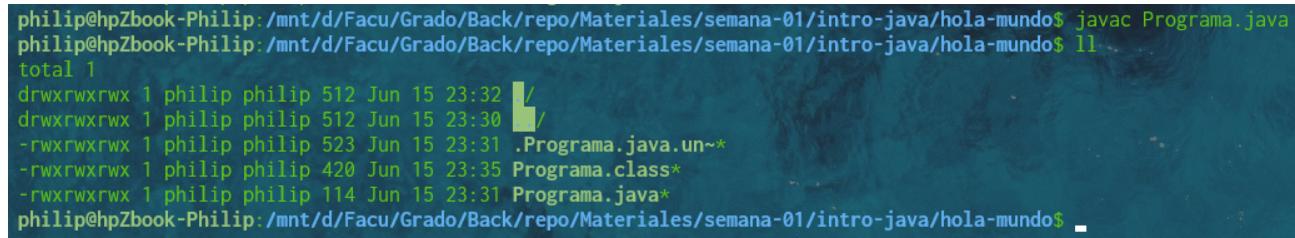
philip@hpZbook-Philip: ~
$ javac --version
javac 11.0.2
```

Habiendo validado la instalación de Java como mencionamos, estamos en condiciones de compilar nuestra clase. Para ello vamos a usar el compilador java este es el programa **javac** (por Java Compiler) y como entrada al compilador le vamos a dar el nombre del archivo que contiene la clase a compilar.

Entonces, estando ubicados en el directorio donde tenemos nuestra clase programa vamos a ejecutar:

```
> javac Programa.java
```

Si el resultado es simplemente la siguiente línea de comandos como vemos en la imagen significa que la compilación fue exitosa y que podemos ver el archivo de salida en el propio directorio.



```
philip@hpZbook-Philip:/mnt/d/Facu/Grado/Back/repo/Materiales/semana-01/intro-java/hola-mundo$ javac Programa.java
philip@hpZbook-Philip:/mnt/d/Facu/Grado/Back/repo/Materiales/semana-01/intro-java/hola-mundo$ ll
total 1
drwxrwxrwx 1 philip philip 512 Jun 15 23:32 /
drwxrwxrwx 1 philip philip 512 Jun 15 23:30 Programa.java
-rwxrwxrwx 1 philip philip 523 Jun 15 23:31 .Programa.java.un~
-rwxrwxrwx 1 philip philip 420 Jun 15 23:35 Programa.class*
-rwxrwxrwx 1 philip philip 114 Jun 15 23:31 Programa.java*
philip@hpZbook-Philip:/mnt/d/Facu/Grado/Back/repo/Materiales/semana-01/intro-java/hola-mundo$
```

En el caso en que exista un error de compilación, el proceso va a fallar, no se va a crear ningún archivo nuevo y vamos a poder ver una referencia al primer error encontrado por el compilador, a continuación un ejemplo donde falta el **;**:



```
philip@hpZbook-Philip:/mnt/d/Facu/Grado/Back/repo/Materiales/semana-01/intro-java/hola-mundo$ javac Programa.java
Programa.java:3: error: ';' expected
    System.out.println("Hola Mundo")
                           ^
1 error
philip@hpZbook-Philip:/mnt/d/Facu/Grado/Back/repo/Materiales/semana-01/intro-java/hola-mundo$ ll
total 0
drwxrwxrwx 1 philip philip 512 Jun 15 23:39 /
drwxrwxrwx 1 philip philip 512 Jun 15 23:35 Programa.java*
-rwxrwxrwx 1 philip philip 113 Jun 15 23:37 Programa.java*
philip@hpZbook-Philip:/mnt/d/Facu/Grado/Back/repo/Materiales/semana-01/intro-java/hola-mundo$
```

En este caso, deberemos volver al código fuente, corregir el error encontrado y repetir este proceso hasta lograr que la compilación sea exitosa.

3 El archivo de código binario Java o Java Byte Code

En el caso de que el proceso de compilación haya resultado exitoso, obtendremos como resultado un nuevo archivo, que tendrá siempre un nombre compuesto por el nombre de la clase, es decir, el mismo nombre que el archivo punto java, en nuestro ejemplo [Programa](#), pero ahora con la extensión .class, es decir [Programa.class](#) como podemos observar en la imagen anterior.

En el contexto de Java, el bytecode es el conjunto de instrucciones de bajo nivel que se genera cuando se compila un programa Java. A diferencia de otros lenguajes de programación, donde el código fuente se compila directamente a código máquina específico del sistema operativo, Java utiliza un enfoque diferente.

Cuando desarrollamos en Java, el código fuente (.java) se compila utilizando el compilador de Java (javac), que traduce el código fuente a bytecode. El bytecode resultante se almacena en archivos con extensión .class. Estos archivos .class contienen instrucciones que son ejecutadas por la Máquina Virtual de Java (JVM), que es un componente crucial en la plataforma Java.

El bytecode de Java es independiente de la plataforma, lo que significa que se puede ejecutar en cualquier sistema que tenga una implementación de la JVM. Esto se debe a que la JVM es responsable de interpretar y ejecutar el bytecode en tiempo de ejecución, sin importar el sistema operativo subyacente. De esta manera, el código Java es altamente portátil y puede ejecutarse en diferentes entornos sin necesidad de realizar modificaciones significativas.

El bytecode de Java es un conjunto de instrucciones orientadas a objetos. Cada instrucción es representada por un código de operación (opcode) seguido de sus operandos, si los tiene. Estas instrucciones son ejecutadas secuencialmente por la JVM durante la fase de ejecución del programa.

Una ventaja clave del bytecode es que proporciona una capa adicional de seguridad. Antes de que se ejecute el bytecode, la JVM realiza una verificación de seguridad para garantizar que no haya violaciones en tiempo de ejecución, como acceso a memoria no autorizada o violaciones en la manipulación de objetos. Esta verificación de seguridad ayuda a prevenir problemas graves y mejora la confiabilidad del programa.

Además, el bytecode también permite la implementación de características de nivel superior en Java, como la gestión automática de memoria (a través del recolector de basura), el soporte para reflexión (que permite inspeccionar y manipular clases y objetos en tiempo de ejecución) y la administración de excepciones.

En resumen, el bytecode de Java es un conjunto de instrucciones de bajo nivel que se genera al compilar el código fuente de Java. Este bytecode es interpretado y ejecutado por la JVM, lo que proporciona portabilidad y seguridad al lenguaje Java. Es gracias a esta combinación de bytecode y JVM que los programas Java pueden ejecutarse en múltiples plataformas sin requerir modificaciones significativas.

4 Ejecución

Habiendo obtenido el archivo de código binario de java a partir de un proceso de compilación exitosa, ahora nos resta ejecutar el programa, pero como ya hemos dicho antes, esta ejecución no es automática ni independiente pues este código binario java NO ES una aplicación ejecutable.

Va a ser necesaria la ya nombrada Máquina Virtual Java o JVM por su sigla en inglés, si hemos descargado e instalado el JDK ya contamos con dicha máquina virtual puesto que como explicaremos en el siguiente apartado está incluida para probar lo que desarrollamos pero si necesitáramos ejecutar nuestro programa en una máquina donde no vamos a programar, no es necesario descargar el JDK sino que tenemos la opción de descargar un JRE (por Java Runtime Environment) o Entorno de Ejecución Java que solo tiene los elementos necesarios para ejecutar una aplicación java y no, las herramientas de programación.

Entonces teniendo la máquina virtual solo nos queda ejecutar la aplicación, para ello vamos a ejecutar la JVM y le vamos a pasar como parámetro el **nombre de la clase** que contiene el método main:

```
philip@hpZbook-Philip:/mnt/d/Facu/Grado/Back/repo/Materiales/semana-01/intro-java/hola-mundo$ java Programa
Hola Mundo
philip@hpZbook-Philip:/mnt/d/Facu/Grado/Back/repo/Materiales/semana-01/intro-java/hola-mundo$
```

Notar que no escribimos .class sino solo el nombre de la clase, aquí podría surgir la pregunta: ¿cómo hace la máquina virtual para encontrar el archivo con el bytecode correspondiente a la clase?. Si hacemos un poco de memoria, cuando comenzamos planteamos una sola restricción que era que el nombre del archivo de código fuente debía llamarse como la clase que contenía; y luego dijimos, que el compilador siempre llama el resultado de la compilación con el mismo nombre que el archivo .java pero con la extensión .class.

Entonces, si nosotros ejecutamos invocando a la máquina virtual y diciéndole el nombre de la clase que tiene que ejecutar, la máquina virtual puede inferir que debe buscar un archivo con el nombre compuesto por <nombre de la clase> más la extensión .class.

Nota: a partir de la versión 11 Java incluye la posibilidad de ejecutar directamente el archivo de la clase, es decir el archivo **Programa.java**, sin embargo esto simplemente está provocando el proceso en memoria y sin escribir el bytecode de manera local y no es el esquema de ejecución cuando vamos a desplegar software de ejecución real.

```
philip@hpZbook-Philip:/mnt/d/Facu/Grado/Back/repo/Materiales/semana-01/intro-java/hola-mundo$ java Programa.java
Hola Mundo
philip@hpZbook-Philip:/mnt/d/Facu/Grado/Back/repo/Materiales/semana-01/intro-java/hola-mundo$
```

¿Por qué se dice que Java es multiplataforma?

Hasta ahora hemos recorrido el camino que nos llevó desde el código fuente original que escribimos para imprimir la cadena "Hola Mundo" en la pantalla, hasta el programa que comprobamos que efectivamente lo hizo.

Ahora bien, más allá de que algo mencionamos al respecto, ¿dónde explícitamente radica el hecho de la multiplataforma?, bueno, para lograr comprender este concepto primero debemos al menos revisar la diferencia que existe entre JDK y JRE específicamente y donde radica la máquina virtual. Respecto del JDK ya hemos explicado bastante y en resumidas cuentas sin descargar e instalar el JDK no podemos programar puesto que este paquete es quien me provee el compilador java entre otras muchas herramientas, y por otro lado también me provee un JRE para probar lo que yo programo en la misma PC, donde estoy programando.

El punto está en que también puedo descargar e instalar de forma separada solo el JRE, y en este caso no estaríamos en condiciones de programar, pero sí de ejecutar un programa ya compilado.

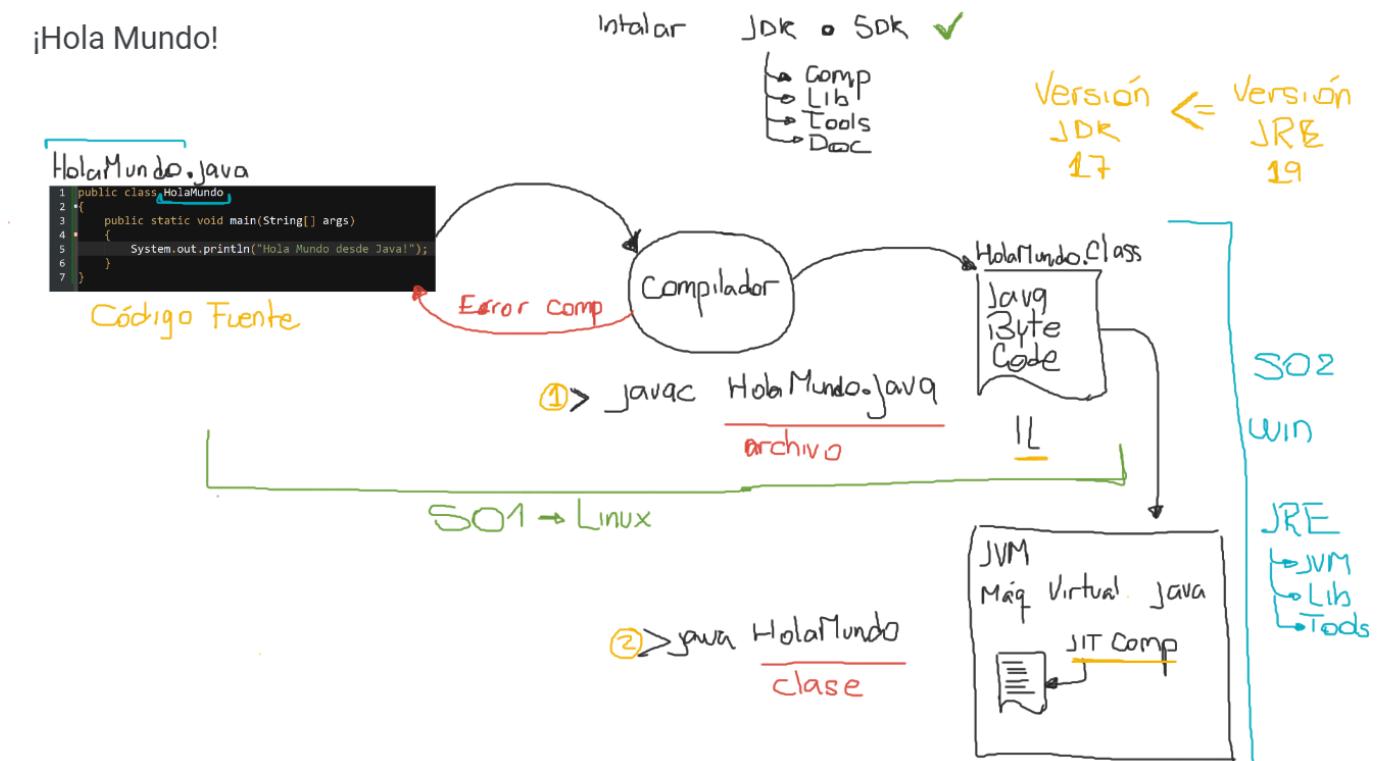
Entonces nosotros podemos realizar el proceso de escritura de código y compilación en cualquier plataforma o sistema operativo para el cual contemos con un JDK, es decir, para el cual exista una versión de JDK que podamos descargar e instalar. Con esto obtendremos como resultado de esta fase un archivo o conjunto de archivos Java Bytecode o .class, los cuales ya dijimos cumplen un rol importante aquí, ¿por qué?, es evidente que en la misma máquina donde programamos y compilamos ejecutamos la aplicación, podemos hacerlo, ya dijimos, el JDK incluye un JRE.

Pero si fuese necesario ejecutar la aplicación en una PC donde nunca he programado y no voy a programar jamás, por ejemplo la PC de mi mamá, o peor aún en una plataforma o sistema operativo para la cual no existe un JDK pero sí existe un JRE, yo podría tomar mis archivos de bytecode copiarlos a la plataforma

destino para la cual cuento con un JRE y simplemente ejecutarlos aquí, cosa que no solo es posible, sino que además se lleva a cabo de la manera óptima para esta plataforma puesto que el JRE y por añadidura la JVM contenida está optimizada para la plataforma para la que fue creada.

En este escenario que describimos estaríamos en presencia de un proceso multiplataforma real y donde aprovecharemos esta característica del lenguaje de programación Java brindada gracias a la existencia de Bytecode. Java en este caso estaría manteniendo el concepto inicial de escribir el código y compilarlo una vez en alguna plataforma para la que exista un JDK disponible y luego poder ejecutarlo, a partir de los archivos de bytecode, en cualquier plataforma para la que exista una Máquina Virtual o lo que es igual, cualquier plataforma para la que exista un JRE disponible y ese es, esencialmente, el esquema de funcionamiento de las aplicaciones Java.

En resumen



Algunas alternativas al esquema tradicional

Desde la versión 11 de Java podemos omitir el paso de compilación, es decir podemos directamente ejecutar la máquina virtual brindando un archivo **MiClase.java** como parámetro de entrada y la máquina virtual va a provocar una compilación en línea y va a ejecutar la clase. Por ejemplo si nos basamos en el archivo **Programa.java** construido antes:

```
philip ➤ D: > ... > intro-java > hola-mundo ➤ p main • ➤ ?6 ~3
$ java Programa.java
Hola Mundo

philip ➤ D: > ... > intro-java > hola-mundo ➤ p main • ➤ ?6 ~3
$
```

En este caso se omite la creación del archivo **MiClase.class** el cual se genera en memoria y es una versión útil para la ejecución de scripts o la realización de pruebas.

Unnamed Classes and Instance Main Methods (Preview)

Esta nueva versión de Java 21 agrega la posibilidad de construir un script sin la palabra reservada `class` y con un método main simplificado en línea con la particularidad anterior imitando un poco la idea de Python a la hora de escribir scripts de rápida ejecución / implementación.

Así podríamos por ejemplo escribir el archivo `Hola.java` con el código del método main exclusivamente:

```
void main() {
    // Notar que no hace falta el parámetro del método main y se omite public y
    static
    System.out.println("Hola Mundo");
}
```

El archivo `Hola.java` entonces, solo contendrá el código antes mencionado, y a partir de este luego podremos ejecutar directamente con el método abreviado existente desde la versión 11 de Java:

```
philip@D:...> intro-java> hola-mundo> p main • > ↵?7 ~3
$ java --enable-preview --source 21 Hola.java
Note: Hola.java uses preview features of Java SE 21.
Note: Recompile with -Xlint:preview for details.
Hola Mundo
```

Notar que para ello tendremos que agregar las opciones de máquina virtual `--enable-preview` para habilitar el uso del acceso temprano a las novedades de la versión y este parámetro requiere que le indiquemos sobre qué versión queremos hacer preview para lo que vamos a usar el parámetro `--source` con la versión 21 para configurar que queremos los preview de la versión 21 de Java.

Sin embargo, más allá de la aclaración de la versión y del preview se puede observar como esta evolución del lenguaje permite la creación de scripts livianos de ejecución directa a partir del archivo de código.

Breve debate sobre las diferentes JVMs existentes

Lo prometido es deuda aquí un brevísimo repaso sobre las distintas máquinas virtuales java existentes:

1. HotSpot VM (Oracle/OpenJDK):

Ventajas: HotSpot VM es una de las máquinas virtuales más utilizadas y optimizadas en la actualidad. Ofrece un alto rendimiento y una amplia gama de características avanzadas, como la optimización Just-In-Time (JIT) y la recolección de basura adaptativa.

Desventajas: Algunas de las desventajas incluyen que puede tener un mayor consumo de memoria y una latencia de inicio ligeramente más larga en comparación con algunas otras máquinas virtuales más ligeras.

2. OpenJ9 (Eclipse Adoptium, anteriormente conocido como IBM J9):

Ventajas: OpenJ9 es conocido por su eficiencia y bajo consumo de memoria. Es una excelente opción para aplicaciones que requieren una huella reducida en términos de recursos.

Desventajas: Algunas aplicaciones pueden requerir ajustes y pruebas adicionales para garantizar una compatibilidad total con OpenJ9.

3. GraalVM:

Ventajas: GraalVM es innovadora debido a su capacidad para ejecutar aplicaciones Java, JavaScript y otros lenguajes en una sola máquina virtual. También ofrece la posibilidad de compilar aplicaciones Java en imágenes nativas.

Desventajas: Aunque es prometedora, GraalVM puede ser más compleja de configurar y usar en comparación con las máquinas virtuales Java más tradicionales.

4. Azul Zing:

Ventajas: Zing se centra en la eliminación de los tiempos de latencia y la reducción de los problemas de rendimiento en tiempo real, lo que lo convierte en una opción preferida para aplicaciones críticas en tiempo real.

Desventajas: Zing suele ser utilizado en casos específicos y puede ser más costoso que otras opciones.

5. SAP Machine:

Ventajas: SAP Machine se ha optimizado para ejecutar aplicaciones SAP y es conocida por su estabilidad y soporte de nivel empresarial.

Desventajas: Su uso está más centrado en aplicaciones SAP y puede no ser tan ampliamente utilizado como otras opciones generales.

La elección de la máquina virtual dependerá de las necesidades específicas del proyecto que se esté desarrollando, como el rendimiento, el consumo de recursos, la estabilidad y las características únicas que ofrezcan. Se recomienda investigar más sobre cada una de estas máquinas virtuales y considerar cómo se ajustan a las demandas de tu aplicación antes de tomar una decisión.

⌚ ¿Qué es realmente una Máquina Virtual? ¿Y por qué Java la necesita?

Antes de cerrar este apunte, vale la pena entender un poco más en profundidad qué es una **Máquina Virtual** y por qué Java se apoya en esta arquitectura para lograr su promesa de "escribir una vez, ejecutar en cualquier lado".

Máquina Virtual: concepto general

Una **Máquina Virtual (VM)** es un entorno de ejecución que simula un sistema operativo o una computadora real. Permite ejecutar programas en un contexto controlado, **independiente del hardware físico**. En este entorno:

- Se aísla la ejecución del sistema operativo base (host)
- Se puede tener más de una VM por equipo físico
- Cada VM puede tener su propio sistema operativo, memoria, procesos, etc.

En el caso de Java, no hablamos de simular un sistema operativo entero, sino de algo más específico...

⌚ La Java Virtual Machine (JVM)

Java no compila directamente a código nativo del procesador (como lo hace C o C++), sino que compila a **bytecode**, un lenguaje intermedio portable.

Ese bytecode **es interpretado o compilado en tiempo de ejecución** por la **Java Virtual Machine (JVM)**.

↗ Esto es lo que hace posible que un `.class` compilado en una máquina Linux con Intel, funcione en una Mac con ARM sin recompilar nada.

Arquitectura básica de la JVM

La JVM está compuesta por:

Componente	Función principal
Class Loader	Carga los <code>.class</code> al iniciar
Runtime Data Areas	Espacios de memoria para objetos, métodos, pilas de ejecución
Execution Engine	Interpreta o compila a código nativo el bytecode
Native Interface (JNI)	Permite invocar código en C/C++ si es necesario
Garbage Collector	Libera automáticamente objetos que ya no se usan

Detalles interesantes (nivel introductorio)

- **ClassLoader** carga clases dinámicamente. Si no encuentra una clase, lanza `ClassNotFoundException` o `NoClassDefFoundError`.
- **Heap y Stack**: los objetos viven en el heap; las variables locales y llamadas de métodos viven en la pila (stack).
- **Execution Engine** puede usar un **intérprete** (más lento pero inmediato) o un **JIT (Just-In-Time) compiler**, que convierte bytecode en código nativo más eficiente para métodos que se usan mucho.
- **Garbage Collector** se encarga de liberar memoria automáticamente; Java es "memory-safe" gracias a esto.

¿Por qué todo esto importa?

Porque este diseño:

- Asegura **portabilidad**
- Permite a Java ser **más seguro** (al estar aislado del sistema)
- Facilita **la optimización en tiempo de ejecución**
- Desacopla el lenguaje del hardware

Y sobre todo, nos da contexto para comprender **qué pasa realmente cuando ejecutamos un programa Java**.

En síntesis: *el bytecode es universal, y la JVM es el traductor local que lo entiende en cada máquina donde se ejecute**...

Enlaces relacionados

- [The 2024 Java Programmer RoadMap \[UPDATED\]](#) si bien es un artículo originalmente de 2021 que fue actualizado muestra un panorama completo del ecosistema Java y sus expectativas a futuro.

Apunte de clase 3 - Maven

Introducción

Maven es una poderosa herramienta de construcción y gestión de proyectos que simplifica y automatiza muchas de las tareas comunes en el desarrollo de aplicaciones Java y otros lenguajes de programación. Su enfoque principal es proporcionar una forma coherente de estructurar, construir y administrar proyectos por un lado, y centralizar, administrar y disponer las dependencias de estos proyectos por el otro; lo que mejora la eficiencia y la colaboración en equipos de desarrollo.

Podemos entender a Maven desde 3 puntos de vista medianamente distintos, por un lado Maven como generador de estructuras de proyecto, por el otro lado maven como administrador y distribuidor de dependencias y finalmente, maven como orquestador del ciclo de vida del proyecto.

Generador de Estructuras de Proyecto: Maven ofrece la posibilidad de utilizar [archetypes](#), que son plantillas predefinidas para crear las estructuras básicas de los proyectos. Estos arquetipos definen la organización de directorios y archivos en un proyecto, incluyendo el archivo [pom.xml](#) (Project Object Model), que es el archivo de configuración del proyecto. Esto ayuda a los desarrolladores a seguir un patrón consistente. Al crear un nuevo proyecto, puedes elegir entre varios arquetipos predefinidos según el tipo de aplicación que estás construyendo o crear un arquetipo base para todos tus proyectos en base a un proyecto tipo construido especialmente. Esto garantiza que los proyectos tengan una estructura coherente desde el principio.

Administrador de Dependencias: Una de las características más destacadas de Maven es su capacidad para administrar dependencias automáticamente. En lugar de descargar manualmente bibliotecas externas y gestionar sus versiones, Maven permite declarar las dependencias del proyecto en el archivo llamado [pom.xml](#). Maven se encarga de descargar automáticamente las dependencias necesarias desde repositorios remotos, como Maven Central, y las incorpora en el proyecto. Esto simplifica en gran medida el proceso de gestión de dependencias y garantiza que todas las partes del proyecto utilicen las mismas versiones de bibliotecas.

Orquestador del Ciclo de Vida del Proyecto: Maven se encarga de coordinar el ciclo completo de construcción de un proyecto, desde la compilación y la ejecución de pruebas hasta la generación de artefactos binarios, como archivos JAR o WAR. A través de comandos simples, como [mvn compile](#), [mvn test](#) o [mvn package](#). Maven automatiza estas tareas y garantiza que el proceso de construcción sea consistente y predecible. También permite configurar y personalizar las fases de construcción según las necesidades del proyecto.

En resumen, Maven es una herramienta esencial en el desarrollo de aplicaciones Java que ofrece un enfoque coherente para generar estructuras de proyecto, administrar dependencias y coordinar el ciclo de construcción. Su capacidad para simplificar tareas comunes y mejorar la eficiencia en el desarrollo hace que sea ampliamente utilizado en la comunidad de desarrollo y en entornos DevOps.

Un poco más de detalle sobre las Funciones de Maven

Generador de Estructuras de Proyecto

Maven proporciona una amplia variedad de arquetipos (plantillas) que te permiten crear proyectos Java de diferentes tipos, como aplicaciones web, aplicaciones de consola, bibliotecas, etc. Los arquetipos establecen una estructura de directorios y archivos estándar para tu proyecto. Algunos conceptos clave relacionados con los arquetipos son:

Artefacto: Un producto generado por Maven, como un archivo JAR o un archivo WAR, que contiene el código compilado y las dependencias del proyecto. Es la unidad de compilación o componente indivisible dentro de la infraestructura Java y recibe un identificador que en conjunto con el GroupId y la Versión deben ser únicos.

Esencialmente un artefacto está asociado al proyecto y se compone de un directorio contenedor con el archivo `pom.xml` y la estructura de directorios y archivos del proyecto.

Para poder crear un proyecto maven a partir de un arquetipo será necesario especificar:

GroupId: Identificador del grupo al que pertenece el proyecto. Suele ser el nombre del dominio de la organización al revés, por ejemplo, `ar.edu.utnfc.backend`.

ArtifactId: Identificador único del artefacto. Es el nombre del proyecto. El nombre de artefacto normalmente se define utilizando spinal case, por ejemplo: `hola-mundo`.

Version: Número de versión del artefacto. Según la convención es un conjunto de 3 números separados por punto seguidos de un guion y las palabras SNAPSHOT o RELEASE determinando si hablamos de una versión en desarrollo o de una versión definitiva del artefacto.

Estos datos requeridos al crear el proyecto como ya veremos, luego pueden ser editados en el archivo `pom.xml` donde quedan especificados.

Lectura complementaria: [Creación de proyectos con Maven Archetypes](#)

Administrador de Dependencias

Dependencias en un Proyecto: En el contexto del desarrollo de software, una dependencia es una entidad externa (como una biblioteca, una API o un módulo) que es utilizada por una aplicación o un proyecto para cumplir ciertas funciones. Las dependencias permiten reutilizar código y funcionalidades existentes en lugar de reinventar la rueda en cada proyecto. Sin embargo, la gestión de dependencias puede volverse complicada si no se controla adecuadamente.

El "DLL Hell" (infierno de las DLL) es un término que se originó en el mundo de Windows y se refiere a los problemas que surgían cuando múltiples aplicaciones dependían de diferentes versiones de una misma librería de vínculos dinámicos (DLL, Dynamic Link Library). Esto llevaba a conflictos y errores cuando las aplicaciones no podían encontrar o utilizar las versiones correctas de las DLL que necesitaban. Este problema surgió debido a la falta de un mecanismo efectivo para gestionar y aislar las dependencias en el sistema operativo.

Evolución hacia la Administración de Dependencias de Maven: La administración de dependencias evolucionó para abordar el problema del "DLL Hell" y los desafíos asociados con la gestión manual de bibliotecas y componentes en proyectos de software. Maven, introducido en 2004, revolucionó la forma en que se manejan las dependencias en proyectos Java y estableció un enfoque coherente y automatizado.

Maven resuelve automáticamente las dependencias del proyecto, descargando las bibliotecas requeridas desde repositorios remotos y configurándolas para su uso en el proyecto. Algunos conceptos clave relacionados con la administración de dependencias son:

Dependency Management: Declaración de dependencias en el archivo pom.xml utilizando la etiqueta <dependency>. Maven descargará y gestionará estas dependencias automáticamente.

Repositorios: Maven utiliza repositorios remotos (como Maven Central) para buscar y descargar dependencias. También puedes configurar repositorios locales o privados.

Transitive Dependencies: Maven resuelve automáticamente las dependencias transitivas, es decir, las bibliotecas que son necesarias debido a otras dependencias.

Lectura complementaria: [Gestión de Dependencias en Maven](#)

Características Clave de la Administración de Dependencias en Maven

Declaración Declarativa: En lugar de descargar y administrar dependencias manualmente, Maven permite declarar las dependencias en un archivo pom.xml, especificando el grupo, el artefacto y la versión. Maven se encarga de descargar y gestionar estas dependencias.

Resolución Automática: Maven resuelve automáticamente las dependencias transitivas, es decir, las bibliotecas requeridas por otras dependencias. Esto evita problemas de compatibilidad y asegura que las versiones correctas se utilicen en todo el proyecto.

Repositorios Centrales y Remotos: Maven utiliza repositorios centralizados (como Maven Central) para almacenar y distribuir dependencias. Esto asegura que las bibliotecas estén disponibles y se descarguen de fuentes confiables.

Repositorio Local: Las dependencias descargadas se almacenan en un repositorio local (por defecto, la carpeta .m2), lo que evita descargas repetidas y facilita el desarrollo sin conexión.

Finalmente como veremos más adelante Maven tiene la capacidad a través de la orquestación del ciclo de vida del proyecto de construir los artefactos y enviarlos al repositorio que corresponda.

Lecturas complementarias:

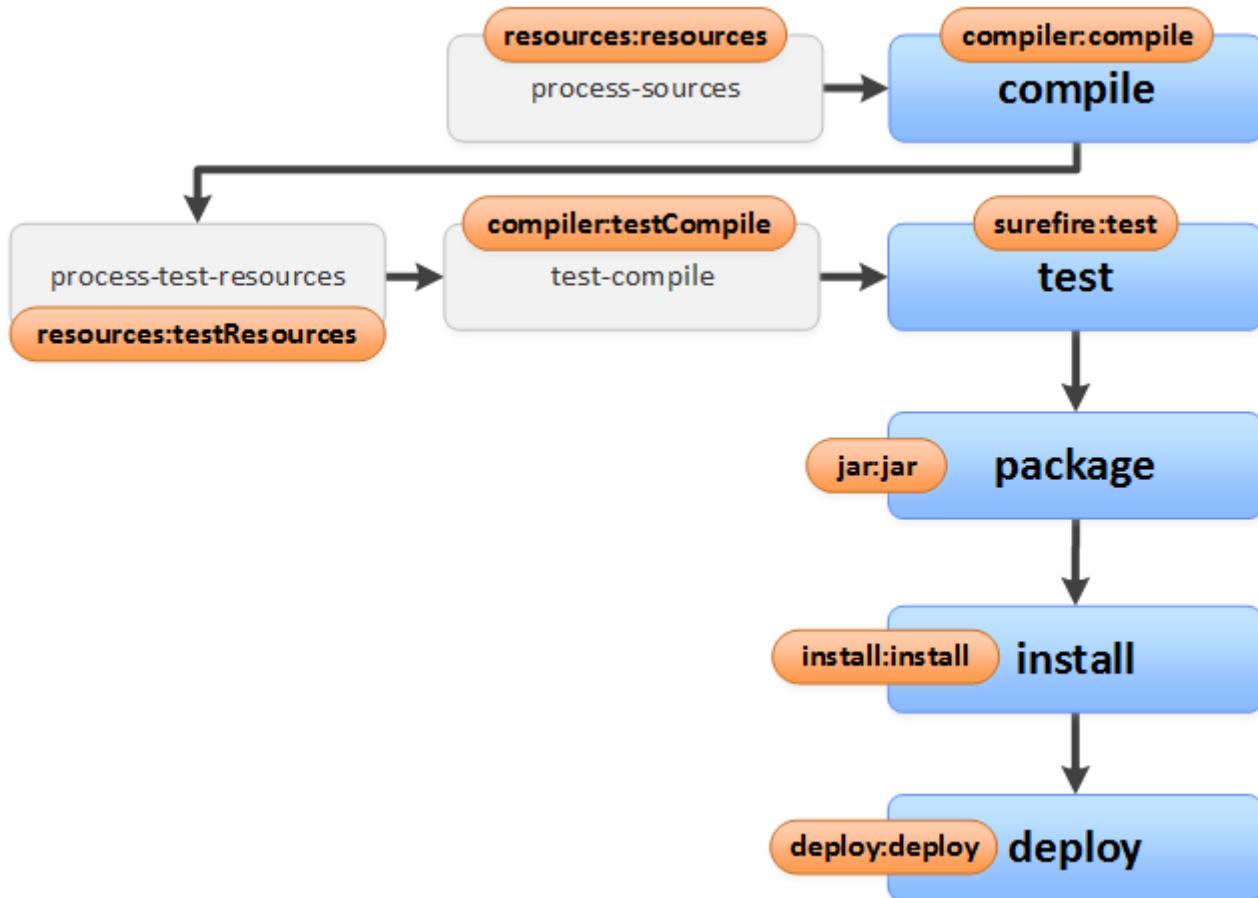
- [Historia y evolución de Maven](#)
- [Maven: A Tale of Two Builds](#)

Orquestación del Ciclo de Construcción del Proyecto

Maven sigue un ciclo de construcción estándar con diferentes fases, cada una de las cuales se encarga de una tarea específica en el proceso de desarrollo y construcción del proyecto. Algunas fases clave del ciclo de construcción son:

- **clean:** Borra todos lo generado por procesos anteriores y deja el proyecto limpio para volver a comenzar
- **validate:** Verifica que el proyecto esté correcto y todas las dependencias estén disponibles.
- **compile:** Compila el código fuente del proyecto.
- **test:** Ejecuta las pruebas automatizadas.

- **package**: Empaque el código compilado en un artefacto (como un JAR o WAR).
- **install**: Instala el artefacto en el repositorio local para su uso en otros proyectos.
- **deploy**: Copia el artefacto final a un repositorio remoto para compartirlo con otros desarrolladores.



Lectura complementaria: [Ciclo de Construcción de Maven](#)

El archivo `pom.xml`

La estructura básica de un proyecto Maven en Java sigue una convención estándar. Aquí un ejemplo de esta estructura de directorios:

```
hola-mundo/
└── src/
    ├── main/
    │   ├── java/
    │   └── resources/
    └── test/
        ├── java/
        └── resources/
    pom.xml
```

Dónde:

- **src/main/java**: Contiene el código fuente principal de la aplicación.
- **src/main/resources**: Contiene recursos (archivos de configuración, archivos de propiedades, etc.) utilizados en la aplicación principal.

- `src/test/java`: Contiene las pruebas unitarias y de integración.
- `src/test/resources`: Contiene recursos utilizados en las pruebas.

Además de los directorios podemos observar el archivo `pom.xml` que venimos nombrando desde el comienzo de las presentes notas de clase y ahora toca revisar en detalle.

El archivo pom.xml (Project Object Model) es el corazón de un proyecto de Maven. Es un archivo de texto en formato XML ([Extensible Markup Language](#)) que contiene la configuración del proyecto y a partir del cual con solo disponer del directorio de proyecto con los archivos de código y este archivo, el proyecto puede ser abierto y trabajado en cualquier IDE que admita el trabajo con proyecto Maven. Aquí están los elementos clave del `pom.xml` y su explicación:

```
<project>
    <modelVersion>4.0.0</modelVersion>
    <groupId>ar.edu.utnfc.backend</groupId>
    <artifactId>holamundo</artifactId>
    <version>1.0.0-SNAPSHOT</version>
    <name>Hola Mundo Maven</name>
    <description>A simple Maven project</description>

    <!-- Especificación de la versión de Java -->
    <properties>
        <maven.compiler.source>21</maven.compiler.source>
        <maven.compiler.target>21</maven.compiler.target>
    </properties>

    <dependencies>
        <!-- Dependencias del proyecto -->
    </dependencies>

    <build>
        <plugins>
            <!-- Plugin de compilación -->
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.8.1</version>
                <configuration>
                    <source>${maven.compiler.source}</source>
                    <target>${maven.compiler.target}</target>
                </configuration>
            </plugin>

            <!-- Otros plugins de construcción -->
        </plugins>
    </build>
</project>
```

- `<modelVersion>`: Versión del modelo del proyecto de Maven (normalmente 4.0.0). -`<groupId>`: Identificador del grupo al que pertenece el proyecto (usualmente el dominio de la empresa al revés).

- <artifactId>: Identificador único del artefacto (nombre del proyecto).
- <version>: Versión del artefacto.
- <name>: Nombre descriptivo del proyecto.
- <description>: Descripción del proyecto.
- <dependencies>: Sección donde se declaran las dependencias del proyecto.
- <properties>: Este bloque permite definir propiedades utilizadas en el proyecto. Aquí, se establece la versión de Java para la compilación y ejecución del proyecto. En este ejemplo, se utiliza Java 8.
- <build>: Este bloque contiene la configuración para la construcción del proyecto.
 - <plugins>: Aquí se definen los plugins de construcción que se utilizarán en el ciclo de construcción del proyecto.
 - <plugin>: Cada plugin de construcción se define dentro de este elemento.
 - <groupId>: Identificador del grupo del plugin.
 - <artifactId>: Identificador del artefacto del plugin.
 - <version>: Versión del plugin. -<configuration>: Permite configurar opciones específicas del plugin. En este caso, se configura el plugin maven-compiler-plugin para que utilice la versión de Java definida en las propiedades.

Otros elementos opcionales: Puedes configurar repositorios, perfiles o credenciales de acuerdo con lo necesario para el proyecto.

Lecturas Complementarias:

- [Configuración del Compilador en POM](#)
- [Plugins de Construcción en POM](#)
- [Configuración del Ciclo de Construcción en POM](#)

Instalación de Maven

Instalación de Maven en Windows

1. Descarga Maven:

- Ve al sitio oficial de Apache Maven: <https://maven.apache.org/download.cgi>
- Descarga el archivo ZIP de la última versión de Maven. La versión actual es la versión 3.9.10, sin embargo cualquier versión posterior a la versión 3.8.3 es apta para trabajar con Java 21. Aquí queda explicado el proceso con la versión 3.9.4 con la que se documentó en 2023.
- Extrae el contenido del archivo ZIP en una ubicación conveniente, el lugar ideal de instalación depende del sistema operativo utilizado, pero lo recomendable es ubicarlo en un directorio que no requiera elevación de permisos para ser utilizado, por ejemplo, el directorio de archivos del usuario `c:\Users\miUsuario\tools\maven-3.9.4`.

2. Configuración de Variables de Entorno:

- Abre el menú de Inicio y busca "Editar las variables de entorno del sistema".
- En la pestaña "Opciones avanzadas", haz clic en "Variables de entorno".
- En "Variables del sistema", crea una nueva variable con el nombre `M2_HOME` y el valor de la ruta a la carpeta donde extrajiste Maven (por ejemplo, `c:\Users\miUsuario\tools\maven-3.9.4`).
- Edita la variable Path y agrega `%M2_HOME%\bin` al final.

Instalación de Maven en Linux o Mac OS

1. Descarga Maven:

- Abre una terminal.
- Navegar hasta el directorio donde vayamos a instalar Maven, una alternativa sería /opt pero requeriría ser sudoer del sistema operativo, otra alternativa sigue siendo el directorio del usuario
- Ejecuta el siguiente comando para descargar el archivo tar.gz de la última versión de Maven:

```
wget https://apache.claz.org/maven/maven-3/3.9.4/binaries/apache-maven-3.9.4-bin.tar.gz
```

2. Descomprimir el archivo:

- Ejecuta el siguiente comando.
- Ejecuta el siguiente comando para descargar el archivo tar.gz de la última versión de Maven:

```
tar -xf apache-maven-3.8.4-bin.tar.gz
```

3. Configurar las variables de entorno:

- Abre el archivo .bashrc para bash shell en linux o .zshrc para z shell en macos con un editor de texto.
- Agrega las siguientes líneas al final del archivo.

```
export M2_HOME=/opt/apache-maven-3.8.4  
export PATH=$PATH:$M2_HOME/bin
```

4. Actualiza las variables de entorno:

- En la terminal, ejecuta:

```
source ~/.bashrc
```

o

```
source ~/.zshrc
```

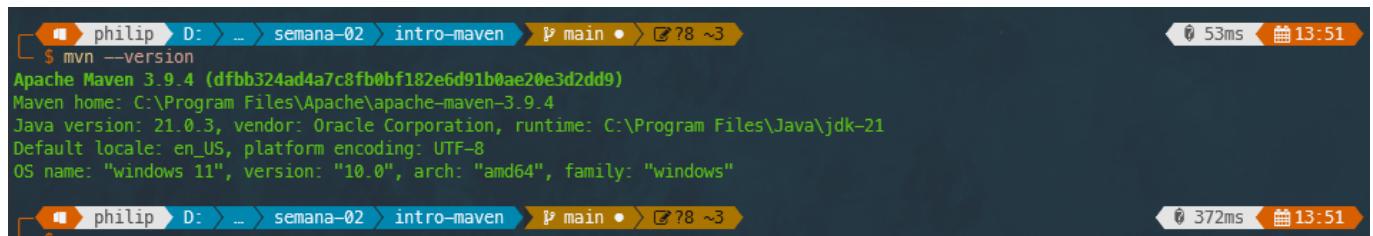
Según tu sistema operativo para cargar los cambios de las variables de entorno.

Verificación de la Instalación

Independientemente del mecanismo utilizado para verificar la instalación y configuración deberemos abrir una terminal o usar la terminal de la instalación y ejecutar:

```
$ mvn --version
Apache Maven 3.9.4
Maven home: <raiz de instalación de maven>/maven-3.9.4
Java version: 17.0.6, ...
```

Si vemos aquí la versión de maven que hemos instalado es porque todo está funcionando Ok y si no lo vemos es porque algo no ha ido bien.



```
philip@D:\...> mvn --version
Apache Maven 3.9.4 (dfbb324ad4a7c8fb0bf182e6d91b0ae20e3d2dd9)
Maven home: C:\Program Files\Apache\apache-maven-3.9.4
Java version: 21.0.3, vendor: Oracle Corporation, runtime: C:\Program Files\Java\jdk-21
Default locale: en_US, platform encoding: UTF-8
OS name: "windows 11", version: "10.0", arch: "amd64", family: "windows"
```

Ubicación del directorio .m2

Como veremos en el último apartado de estas notas de clase, la instalación de maven se completa con un directorio especial que es el directorio `.m2`

La ubicación del directorio `.m2` (repositorio local) es la misma en todos los sistemas operativos. Por defecto, se encuentra en el directorio de usuario. Por ejemplo, en Windows, sería `C:\Users\miUsuario\.m2`, y en Linux o macOS, sería `/home/miUsuario/.m2`.

En el caso de requerir agregar alguna configuración específica o personalizada a maven el lugar para dicha configuración es un archivo llamado `settings.xml` ubicado directamente dentro de la carpeta `.m2`; sin embargo, maven funciona sin la necesidad de dicho archivo con la configuración por defecto.

Siguiendo estos pasos, habrás instalado y configurado correctamente la última versión de Maven en Windows, Linux y macOS, y estarás listo para comenzar a desarrollar proyectos con esta herramienta.

Hola mundo con Maven

A continuación, nos proponemos retomar la tarea del apunte de clases anterior en el que creamos un programa "Hola Mundo Java" con su consecuente compilación y ejecución por línea de comandos, pero ahora utilizando la herramienta Maven para llevar a cabo la creación del proyecto y todas las tareas asociadas a la construcción y ejecución de dicho artefacto de software.

1. Vamos a crear el proyecto:

- Ahora vamos a crear el proyecto y para ello vamos a utilizar un archetype.
- En Maven Central existe un archetype predefinido para crear una aplicación java básica, el nombre de este archetype es: `maven-archetype-quickstart`
- Creamos el proyecto utilizando la herramienta mvn

```
mvn archetype:generate "-DgroupId=ar.edu.utnfc.backend"
"-DartifactId=hola-mundo"
"-DarchetypeGroupId=org.apache.maven.archetypes"
"-DarchetypeArtifactId=maven-archetype-quickstart"
"-DinteractiveMode=false"
```

Nota: en Power Shell cada uno de los argumentos debe ser encerrado entre comillas dobles sino maven va a producir un error

- También es posible utilizar la versión interactiva que nos irá preguntando cada uno de los parámetros que hemos comentado en el presente material.
- Si todo funcionó Ok vamos a ver la siguiente salida:

```
philip@philip-Dell-E5430: ~> mvn archetype:generate "-DgroupId=ar.edu.utnfc.backend" "-DartifactId=hola-mundo" "-DarchetypeGroupId=org.apache.maven.archetypes" "-DarchetypeArtifactId=maven-archetype-simple" "-DinteractiveMode=false"
[INFO] Scanning for projects...
[INFO]
[INFO] < org.apache.maven:standalone-pom >
[INFO] Building Maven Stub Project (No POM) 1
[INFO] [ pom ]
[INFO]
[INFO] >>> archetype:3.2.1:generate (default-cli) > generate-sources @ standalone-pom >>>
[INFO]
[INFO] <<< archetype:3.2.1:generate (default-cli) < generate-sources @ standalone-pom <<<
[INFO]
[INFO]
[INFO] --- archetype:3.2.1:generate (default-cli) @ standalone-pom ---
[INFO] Generating project in Batch mode
[INFO] Archetype [org.apache.maven.archetypes:maven-archetype-simple:1.4] found in catalog remote
[INFO]
[INFO] Using following parameters for creating project from Archetype: maven-archetype-simple:1.4
[INFO]
[INFO] Parameter: groupId, Value: ar.edu.utnfc.backend
[INFO] Parameter: artifactId, Value: hola-mundo
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Parameter: package, Value: ar.edu.utnfc.backend
[INFO] Parameter: packageInPathFormat, Value: ar/edu/utnfc/backend
[INFO] Parameter: package, Value: ar.edu.utnfc.backend
[INFO] Parameter: groupId, Value: ar.edu.utnfc.backend
[INFO] Parameter: artifactId, Value: hola-mundo
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Project created from Archetype in dir: D:\Facu\Grado\Back\repo\Materiales\semana-02\intro-maven\temp\hola-mundo
[INFO]
[INFO] BUILD SUCCESS
[INFO]
[INFO] Total time:  3.430 s
[INFO] Finished at: 2024-08-19T13:54:30-03:00
[INFO]
```

2. Revisamos lo creado por Maven:

- En primer lugar vamos a encontrar un directorio con el nombre del artefacto, en este ejemplo **hola-mundo**
- y dentro de este directorio vamos a encontrar la siguiente estructura de directorios y archivos:

```

Folder PATH listing for volume Data
Volume serial number is 7CAF-8DFA
D:.
    README.md

    hola-mundo
        pom.xml

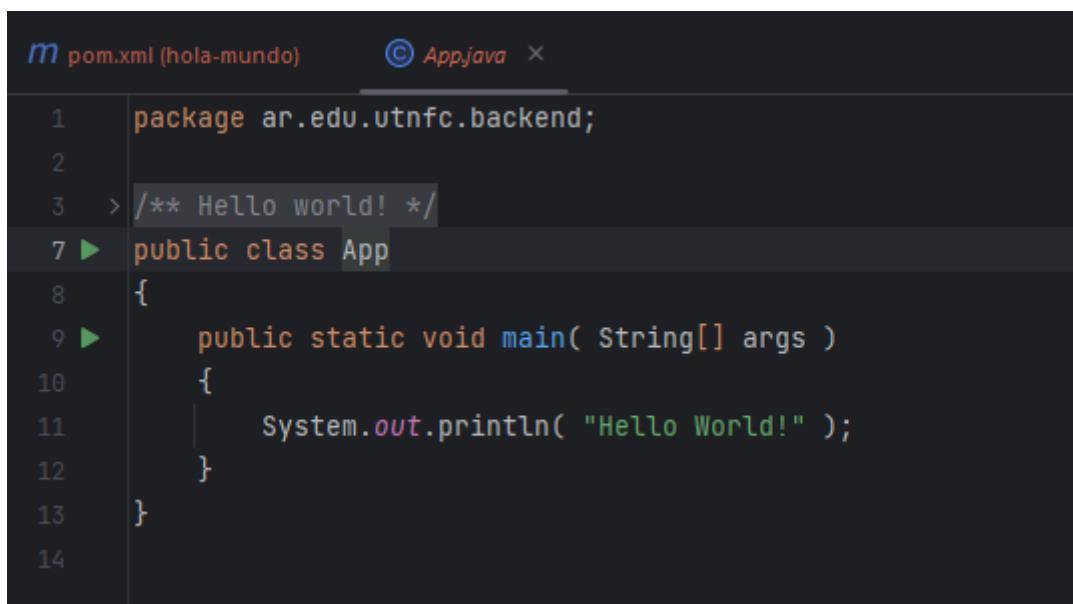
        src
            main
                java
                    ar
                        edu
                            utnfc
                                backend
                                    App.java

            site
                site.xml

            test
                java
                    ar
                        edu
                            utnfc
                                backend
                                    AppTest.java

```

- Aquí hay varias cosas para revisar, en primer lugar el archivo `pom.xml` que se encuentra directamente en la raiz del directorio creado para el proyecto.
- Luego la convención de los directorios `src` para archivos del proyecto y `test` para los archivos de las pruebas y un directorio `site` con un archivo de información del proyecto que es utilizada por el plugin `site` para crear un sitio asociado al proyecto (No vamos a usar dicho plugin en Backend).
- Ahora bien, además del archivo `pom.xml` también vamos a encontrar dentro del directorio `main/java/<paquetes asociados al groupId>` el archivo `App.java` que no es ni más ni menos que un "Hola Mundo" Similar al que hicimos de forma manual en el apunte de clases anterior.



```

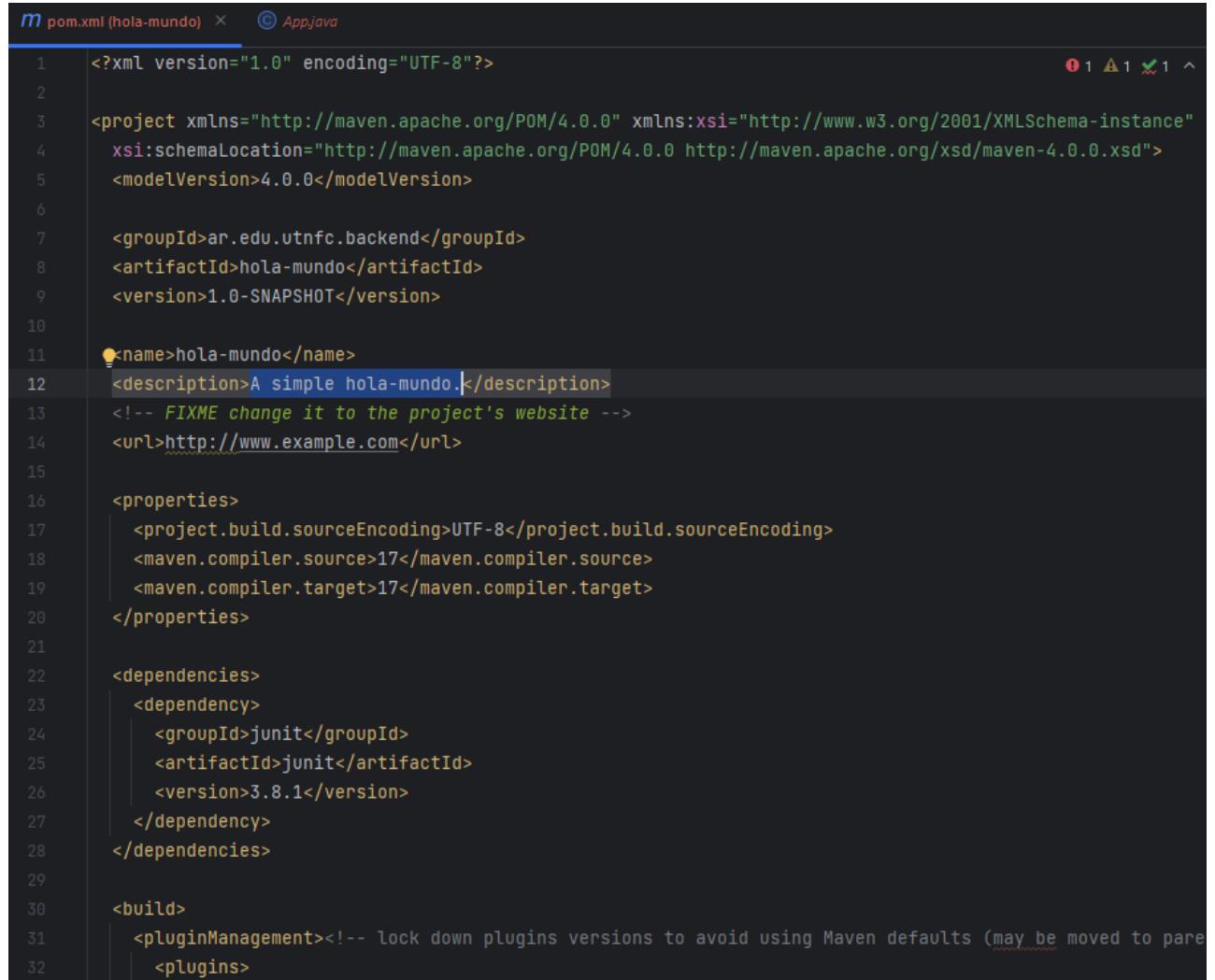
m pom.xml (hola-mundo)  © App.java ×

1 package ar.edu.utnfc.backend;
2
3 > /**
4  * Hello world!
5  */
6
7 ► public class App
8 {
9     ►     public static void main( String[] args )
10    {
11        System.out.println( "Hello World!" );
12    }
13 }
14

```

Nota: los paquetes o **packages** en java son una estructura de directorios para organizar las clases asociadas a la sentencia **package** que podemos observar en la primera línea de código del archivo **.java**.

- Pero finalmente nos enfocaremos en el corazón del proyecto maven que es el archivo **pom.xml**



```

m pom.xml (hola-mundo) × ⚡ App.java
1 <?xml version="1.0" encoding="UTF-8"?>
2
3 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
5   <modelVersion>4.0.0</modelVersion>
6
7   <groupId>ar.edu.utnfc.backend</groupId>
8   <artifactId>hola-mundo</artifactId>
9   <version>1.0-SNAPSHOT</version>
10
11  <name>hola-mundo</name>
12  <description>A simple hola-mundo.</description>
13  <!-- FIXME change it to the project's website -->
14  <url>http://www.example.com</url>
15
16  <properties>
17    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
18    <maven.compiler.source>17</maven.compiler.source>
19    <maven.compiler.target>17</maven.compiler.target>
20  </properties>
21
22  <dependencies>
23    <dependency>
24      <groupId>junit</groupId>
25      <artifactId>junit</artifactId>
26      <version>3.8.1</version>
27    </dependency>
28  </dependencies>
29
30  <build>
31    <pluginManagement><!-- lock down plugins versions to avoid using Maven defaults (may be moved to parent -->
32    <plugins>

```

- Aquí podemos identificar los elementos de configuración del proyecto, como también la versión de Java y del código Fuente que por defecto es establecida a 1.7 y personalmente la cambié a 17. La dependencia de la librería JUnit de tests unitarios y en adelante la configuración de plugins para la construcción del proyecto.

3. Bien ahora construyamos este *Hola Mundo*

- Para ello podemos abrir el directorio anterior con IntelliJ y utilizar la terminal del IDE o quedarnos en la terminal del sistema operativo.
- Estando entonces dentro del directorio del proyecto vamos a ejecutar:

```
> mvn compile
```

- Si todo fue bien vamos a observar una salida similar a esta:

```
[INFO] Scanning for projects...
[INFO] --- maven-compiler-plugin:3.8.0:compile (default-compile) @ hola-mundo ---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 1 source file to D:\Facu\Grado\Back\repo\Materiales\semana-02\intro-maven\hola-mundo\target\classes
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 1.209 s
[INFO] Finished at: 2023-08-08T11:24:38-03:00
[INFO] -----
```

- Y como lo indica la propia salida la particularidad más visible es que aparece la carpeta target dentro del proyecto contenido el resultado de la compilación de todas las clases. De hecho, si buscamos en target/classes/<paquetes asociados al groupId>/ vamos a encontrar el archivo App.class que podemos usar para ejecutar la aplicación como lo hicimos en el apunte de clases anterior.
- Sin embargo, al trabajar con maven tenemos una manera más elegante de empaquetar y ejecutar nuestra aplicación java. Aunque, para lograrlo vamos a tener primero que agregar al archivo pom.xml la configuración que le indicará a Maven cuál es el archivo que tiene el método main y debe ser utilizado como punto de inicio de nuestra aplicación, que por ahora es lo único que tiene.
- Para ello vamos a tener que modificar la configuración del plugin que lleva a cabo el empaquetado en el archivo .jar para indicarle que agregue el archivo MANIFEST.MF que esencialmente el que diferencia un archivo .jar que es solo una librería de un archivo .jar que es una aplicación que podemos ejecutar como veremos más adelante.
- Para lograrlo vamos a ubicar el plugin encargado de generar el archivo .jar <artifactId>maven-jar-plugin</artifactId> y luego de la versión que ya tiene el archivo pom.xml especificada, agregamos lo siguiente (dicho plugin debe quedar como sigue):

```
<plugin>
  <artifactId>maven-jar-plugin</artifactId>
  <version>3.0.2</version>
  <configuration>
    <archive>
      <manifest>
        <mainClass>ar.edu.utnfc.backend.App</mainClass>
      </manifest>
    </archive>
  </configuration>
</plugin>
```

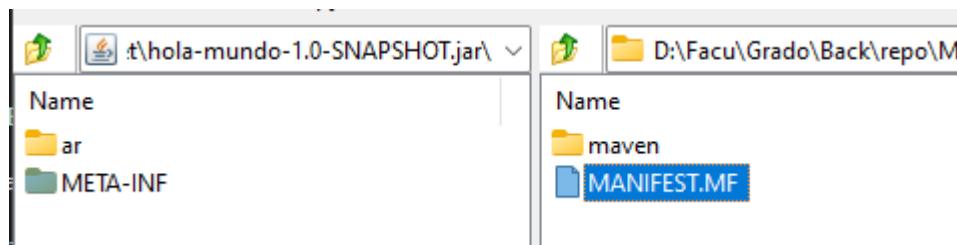
- Una vez realizada esta configuración tenemos que construir el proyecto usando:

```
> mvn package
```

- Si todo resulta Ok la salida será más o menos así:

```
[INFO] Scanning for projects...
[INFO]
[INFO] -----< ar.edu.utnfc.backend:hola-mundo >-----
[INFO] Building hola-mundo 1.0-SNAPSHOT
[INFO] -----[ jar ]-----
[INFO]
[INFO] --- maven-resources-plugin:3.0.2:resources (default-resources) @ hola-mundo ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory D:\Facu\Grado\Back\repo\Materiales\semana-02\intro-maven\hola-mundo\src\main\resources
[INFO]
[INFO] --- maven-compiler-plugin:3.8.0:compile (default-compile) @ hola-mundo ---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 1 source file to D:\Facu\Grado\Back\repo\Materiales\semana-02\intro-maven\hola-mundo\target\classes
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO]
[INFO] --- maven-jar-plugin:3.0.2:jar (default-jar) @ hola-mundo ---
[INFO] Building jar: D:\Facu\Grado\Back\repo\Materiales\semana-02\intro-maven\hola-mundo\target\hola-mundo-1.0-SNAPSHOT.jar
[INFO]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO]
[INFO] -----
[INFO] Total time: 2.735 s
[INFO] Finished at: 2023-08-08T13:01:51-03:00
[INFO] -----
```

- Ahora el directorio **target** contiene el archivo **hola-mundo-1.0-SNAPSHOT.jar** que corresponde con el nombre del artefacto más la versión con la extensión .jar
- Si abrimos este archivo con una herramienta para descomprimir archivos dentro vamos a encontrar el mencionado archivo **MANIFEST.MF** que indica el punto de inicio de la aplicación



- Con el texto indicando la clase main, como se puede observar:

```
1 Manifest-Version: 1.0
2 Created-By: Apache Maven 3.9.4
3 Built-By: philip
4 Build-Jdk: 17
5 Main-Class: ar.edu.utnfc.backend.App
6
```

4. Ejecutando nuestra aplicación

- Ahora tenemos un componente Java ejecutable...
- Como sabemos, y ya hemos comentado en el apunte de clases anterior, no se puede generar código ejecutable con Java sino que lo que tenemos en el conjunto de archivos .class, empaquetado junto con su configuración de manera tal de poder transportar y contener todo lo necesario en un solo archivo, eso es un archivo .jar por Java Archive.
- Para ejecutar un archivo .jar vamos a usar la máquina virtual java como lo hicimos en el apunte de clases anterior pero le vamos a agregar el parámetro **-jar** para indicarle que lo que le estamos dando como entrada es un empaquetado de una aplicación java.
- Entonces, para ejecutar usamos la línea de comandos:

```
> java -jar hola-mundo-1.0-SNAPSHOT.jar
```

- Y allí finalmente tenemos nuestra aplicación ejecutando.

```
[08-08 13:16] philip@HPZBOOK-PHILIP | D:\Facu\Grado\Back\repo\Materiales\s> java -jar .\hola-mundo-1.0-SNAPSHOT.jar
Hello World!
[08-08 13:16] philip@HPZBOOK-PHILIP | D:\Facu\Grado\Back\repo\Materiales\s> -
```

Alternativa de ejecución utilizando el propio administrador mvn

¿Por qué y cómo se puede ejecutar una aplicación Java con mvn?

Maven como orquestador de tareas

Maven no es un compilador ni un intérprete. Es un framework de automatización del ciclo de vida de un proyecto Java (y de otros lenguajes, aunque su ecosistema está centrado en Java). Sus tareas no son ejecutadas por él mismo, sino por plugins, que son componentes específicos que saben hacer una acción: compilar, testear, empaquetar, etc.

Cuando ejecutás un comando como:

```
mvn compile
```

Maven no llama directamente a `javac`, sino que invoca al maven-compiler-plugin, que tiene configurado cómo y con qué parámetros debe invocar `javac`.

De la misma manera, para ejecutar tenemos la alternativa:

```
mvn exec:java
```

Que no ejecuta directamente java, sino que invoca al plugin exec-maven-plugin, el cual ejecuta un comando Java real bajo ciertas configuraciones (como qué clase tiene el método main y qué classpath usar).

Este plugin debe ser referenciado y configurado en el archivo `pom.xml` por ejemplo para el ejemplo del archetype `maven-archetype-quickstart` deberíamos agregar al archivo `pom.xml` lo siguiente:

Agregar dentro de `<build><plugins>`:

```
...
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>exec-maven-plugin</artifactId>
```

```
<version>3.1.0</version>
<configuration>
    <mainClass>ar.edu.utnfc.backend.App</mainClass>
</configuration>
</plugin>

...

```

💡 ¿Por qué esto es más que un atajo?

Usar `mvn exec:java` tiene ventajas claras:

- 1. Usa el classpath resuelto por Maven Ejecuta el programa con todas las dependencias descargadas correctamente. No necesitas escribir ni construir manualmente un classpath.
- 2. Se integra con el ciclo de vida Podés decir: `mvn clean compile exec:java` y asegurarte de compilar desde cero. En proyectos grandes, esto se vuelve un flujo confiable.
- 3. Funciona igual para proyectos simples o complejos (como Spring Boot) Spring Boot usa `spring-boot-maven-plugin`, que extiende el ciclo con un `spring-boot:run`, que hace lo mismo: resuelve dependencias, configura classpath y ejecuta.

📋 Entonces, ¿cuándo y por qué usar mvn exec:java?

- Durante el desarrollo, para correr aplicaciones pequeñas, sin necesidad de construir jar.
- Para scripts que se ejecutan en etapas de CI/CD sin necesidad de empaquetado.
- Para proyectos educativos o pruebas, donde se quiere evitar fricción con la terminal y comandos manuales.
- Porque es consistente con todo el ciclo Maven: dependencias, compilación, plugins, etc.

Repositorios de dependencias

Para cerrar el presente Apunte de Clases, no podemos pasar por alto una breve descripción del concepto de repositorio y las diferencias entre los diferentes repositorios con los que interactúa maven durante el ciclo de vida del proyecto.

Por ello en primer lugar tenemos Maven Central o repositorio central global de maven, además de otros repositorios globales de dependencias específicas que podemos configurar. Luego tenemos los repositorios privados o propios, esto es instalar el software del repositorio como por ejemplo Nexus en los servidores de nuestra organización para compartir allí los componentes de los distintos equipos de trabajo de la organización y finalmente, el repositorio local o también llamado cache que es esencialmente la carpeta `.m2` de la instalación local de maven.

1. **Repositorio Central:** El repositorio central es un repositorio de dependencias mantenido por la comunidad Maven. Es el repositorio predeterminado desde el cual Maven descarga dependencias cuando las declaras en tu archivo `pom.xml`. El repositorio central almacena una gran cantidad de bibliotecas y artefactos Java de uso común, lo que facilita la incorporación de estas dependencias en tus proyectos sin necesidad de descargarlas manualmente.

2. **Repositorio Remoto:** Un repositorio remoto es un servidor que almacena dependencias y artefactos de proyectos. Puedes configurar Maven para usar repositorios remotos adicionales además del repositorio central. Esto es útil cuando trabajas con bibliotecas específicas o en entornos con restricciones de red. Maven descarga dependencias desde repositorios remotos según sea necesario y las almacena en tu repositorio local para su reutilización. Para lograr que maven acceda a repositorios remotos específicos será necesario configurar el acceso a dichos repositorios o bien en el archivo `pom.xml` para un proyecto en particular o bien en el archivo `settings.xml` para todos los proyectos en mi pc.

3. **Repositorio Local o Cache:** El repositorio local es un directorio en tu sistema donde Maven almacena las dependencias descargadas. Por defecto, este repositorio está ubicado en la carpeta `.m2` en tu directorio de usuario. Cuando Maven descarga una dependencia por primera vez desde un repositorio remoto, la almacena en el repositorio local. Esto evita tener que descargar la misma dependencia nuevamente en el futuro y permite trabajar sin conexión a Internet.

Comparación con la Carpeta Local `.m2`: Repositorio Local y Caché de Dependencias Remotas

El directorio `.m2` en tu sistema actúa como el repositorio local y también como una caché de dependencias remotas descargadas. Aquí hay una comparación detallada de su papel:

Repositorio Local (`.m2`):

- Almacena todas las dependencias descargadas en tu sistema.
- Evita descargas repetidas al permitir que Maven reutilice dependencias ya descargadas.
- Facilita la construcción de proyectos en entornos sin acceso a Internet al proporcionar dependencias previamente descargadas.
- Permite la colaboración en equipos, ya que todos los miembros del equipo pueden utilizar las mismas dependencias almacenadas localmente.

Caché de Dependencias Remotas:

- Actúa como una caché temporal de dependencias descargadas desde repositorios remotos.
- Cuando Maven necesita una dependencia, primero verifica si está en la caché local antes de intentar descargarla nuevamente.
- Optimiza el rendimiento al reducir la necesidad de descargar repetidamente las mismas dependencias desde repositorios remotos.

Lectura complementaria: [Repositorios en Maven](#)

En resumen, los repositorios de dependencias (central, remoto y local) y la carpeta local `.m2` desempeñan un papel crucial en la gestión y distribución eficiente de las dependencias en proyectos Maven. Estos conceptos son fundamentales para garantizar que tus proyectos tengan acceso a las bibliotecas necesarias y funcionen de manera confiable en diferentes entornos de desarrollo.

Ciclo de Vida de Construcción en Maven

A modo de Resumen, agregamos aquí una tabla con los distintos pasos del ciclo de vida de la administración de proyectos con Maven:

Fase	Descripción
------	-------------

Fase	Descripción
validate	Verifica que el proyecto está bien estructurado y toda la información es válida.
compile	Compila el código fuente del proyecto.
test	Ejecuta las pruebas unitarias utilizando un marco de pruebas como JUnit.
package	Empaque el código compilado en un formato distribuible, como .jar o .war.
verify	Ejecuta cualquier verificación necesaria sobre el resultado del empaquetado.
install	Instala el artefacto en el repositorio local (.m2) para ser usado por otros proyectos.
deploy	Copia el artefacto final a un repositorio remoto para compartirlo con otros desarrolladores.

❖ Anexo 1: Incluir dependencias dentro del .jar (Fat JAR o Uber JAR)

El .jar que genera Maven por defecto incluye solo las clases propias del proyecto. Si el programa requiere bibliotecas externas (como JUnit, Gson, etc.), al ejecutar con java -jar puede fallar con errores de clase no encontrada.

Para evitar esto, podemos generar un .jar **con todas las dependencias incluidas**, usando el plugin maven-assembly-plugin. Esto se conoce como **Fat JAR** o **Uber JAR**.

Configuración en el pom.xml

Agregar dentro de <build><plugins>:

```
<plugin>
  <artifactId>maven-assembly-plugin</artifactId>
  <version>3.3.0</version>
  <configuration>
    <archive>
      <manifest>
        <mainClass>ar.edu.utnfc.backend.App</mainClass>
      </manifest>
    </archive>
    <descriptorRefs>
      <descriptorRef>jar-with-dependencies</descriptorRef>
    </descriptorRefs>
  </configuration>
  <executions>
    <execution>
      <id>make-assembly</id>
      <phase>package</phase>
      <goals>
        <goal>single</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

Entonces, ¿Qué es un Uber Jar (o Fat Jar) y qué produce exactamente?

Cuando hablamos de construir un Uber Jar con Maven, nos referimos a generar un archivo .jar autocontenido, que incluye:

- Las clases compiladas del proyecto (las tuyas, en src/main/java)
- Los recursos del proyecto (archivos en src/main/resources)
- Y lo más importante: TODAS las dependencias que el proyecto necesita para ejecutarse (archivos .jar descargados por Maven)

Todo eso se empaqueta en un solo .jar ejecutable, que puede correrse sin necesitar nada más instalado o configurado (más allá de tener Java).

💡 ¿Qué hace Maven por defecto al ejecutar mvn package?

Sin configuración adicional:

- Toma tus .class compilados
- Los empaqueta en un archivo .jar
- Crea un archivo target/mi-proyecto-1.0.jar

Pero no incluye las dependencias. Si usás bibliotecas externas (como Gson, JUnit, Apache Commons, etc.), ese jar fallará si las clases no están disponibles en el classpath cuando lo ejecutes.

💡 ¿Qué hace un Uber Jar que no hace un .jar normal?

Con un plugin como maven-assembly-plugin, Maven:

1. Combina tu código con las dependencias:

- Descomprime cada .jar de dependencia
- Extrae sus clases y recursos
- Los mete dentro de tu .jar

2. Agrega un archivo especial META-INF/MANIFEST.MF que declara:

```
Main-Class: ar.edu.utnfc.backend.App
```

Esto le dice a java cuál es la clase de inicio al ejecutar el jar

3. Genera un .jar nuevo con sufijo -jar-with-dependencies:

Ejemplo: target/hola-mundo-1.0-SNAPSHOT-jar-with-dependencies.jar Este nuevo .jar puede ejecutarse directamente como lo vimos anteriormente al agregar las configuración del [jar-plugin](#):

```
java -jar target/hola-mundo-1.0-SNAPSHOT-jar-with-dependencies.jar
```

Y se ejecutará correctamente incluso si no tenés ninguna dependencia instalada en tu sistema ni configuraste ningún classpath.

En este punto con proyectos sin demasiadas dependencias no es algo que se vuelva crítico ni necesario pero más adelante cuando los proyectos dependan de las dependencias requeridas va a ser esencial.

📝 ¿Cómo lo logra? (detalle técnico básico)

Internamente, un .jar es solo un archivo .zip con clases y un manifiesto. Cuando hacés un Uber Jar, Maven toma:

- target/classes/... → tu código compilado
- dependency.jar!/com/ejemplo/... → el contenido de cada .jar de dependencia
- Los fusiona en un único archivo .jar
- Y asegura que haya un único META-INF/MANIFEST.MF válido

⌚ ¿Por qué usar un Uber Jar?

Caso de uso	Beneficio
Ejecutar en otra máquina sin necesidad de Maven instalado	<input checked="" type="checkbox"/> El .jar ya contiene todo lo necesario
Desplegar una app sencilla (CLI, microservicio, tarea batch)	<input checked="" type="checkbox"/> Portabilidad absoluta y ejecución directa
Evitar conflictos de classpath en entornos complejos	<input checked="" type="checkbox"/> Las versiones ya están resueltas y empaquetadas
Aprender cómo funciona una app Java real	<input checked="" type="checkbox"/> Claridad total sobre clases, dependencias y ejecución

🚀 ¿Y después qué? Introducción a `mvn deploy` y repositorios remotos

Hasta ahora vimos cómo usar Maven para compilar, testear y empaquetar un proyecto localmente. Pero, ¿qué pasa cuando necesitamos **compartir ese artefacto (.jar) con otros equipos o proyectos?**

Ahí entra en juego la acción `deploy`.

¿Qué es `mvn deploy`?

El comando `mvn deploy` es el paso final del ciclo de vida de Maven. Sirve para **subir (desplegar) el artefacto generado a un repositorio remoto**, desde donde otros proyectos pueden descargarlo como dependencia.

```
mvn deploy
```

Esto solo tiene sentido cuando se configura un repositorio remoto accesible por Maven (por ejemplo, un servidor privado de la organización).

¿Qué es un repositorio remoto?

Un **repositorio Maven remoto** es un servidor especializado que almacena artefactos Java (como `.jar`, `.war`, `.pom`, etc.) y los organiza por `groupId`, `artifactId` y `version`.

Los proyectos Maven pueden **publicar artefactos** en estos repositorios y **consumir artefactos** de ellos como dependencias.

💡 Software popular para repositorios remotos

- **Nexus Repository Manager** (de Sonatype)

Muy utilizado en empresas, gratuito en su versión OSS.

- **JFrog Artifactory**

Profesional, muy completo, con versión gratuita.

- **Apache Archiva**

Alternativa open-source más simple.

- **GitLab Package Registry**

Permite subir artefactos `.jar` a proyectos GitLab, integrando la gestión de dependencias con el control de versiones y CI/CD. Compatible con Maven mediante configuración del `pom.xml` y `settings.xml`.

💡 ¿Cómo se usa `deploy`?

Para desplegar tu `.jar` al repositorio remoto necesitas:

1. Un `pom.xml` con información del servidor remoto (`<distributionManagement>`)
2. Configurar las credenciales en el archivo `~/.m2/settings.xml`
3. Un servidor como Nexus/Artifactory ejecutándose
4. Ejecutar:

```
mvn deploy
```

Maven empaqueta y sube tu artefacto al servidor remoto, junto con su `pom.xml` y metadatos.

¿Por qué es importante esto?

Caso de uso	Beneficio
Compartir librerías internas entre equipos	<input checked="" type="checkbox"/> Centralización, control de versiones, reutilización
Automatizar pipelines CI/CD	<input checked="" type="checkbox"/> Build + deploy automático desde Jenkins, GitLab CI, etc.
Versionar y publicar tus artefactos	<input checked="" type="checkbox"/> Historial de builds accesibles y trazables

Esta es una **punta de ovillo** 🎁: no lo vamos a usar en la materia, pero dejamos el concepto para completar el ciclo y articular cómo encaja en proyectos reales de desarrollo profesional.

Apunte de clase 4 - Sintaxis

En el apunte de clases 2, hicimos ya una breve identificación del Lenguaje de Programación Java en cuanto a sus características generales específicas. Mencionamos allí que Java es un lenguaje de programación basado en *Objetos*, *Case Sensitive*, que utiliza *Llaves* para delimitar los bloques de código y que finaliza las sentencias de programación con un *Punto y coma*. Esto no es poco decir porque en un simple párrafo estamos delineando los elementos fundantes del lenguaje.

En este apunte de clases vamos a sumar los conceptos iniciales y necesarios para poder programar en Java como son los *Tipos de datos*, *Operaciones* tanto de asignación como operaciones propiamente dichas, *Estructuras de control* y finalizaremos el presente apunte con alguna herramienta introductoria para *Lectura o Ingreso de datos*. Allá vamos.

Tipos, Variables y Asignaciones

Todo programa Java necesitará en algún momento recibir, operar, almacenar y mostrar o devolver valores o datos. Para ello como en todos los lenguajes de programación vistos hasta aquí vamos a utilizar variables. Sin embargo, con java será la primera vez que nos encontraremos frente a un lenguaje de programación *tipado*, es decir que toda variable tiene un tipo de datos y este tipo de datos en la mayoría de los casos debe ser especificado por el programador y no puede cambiar en el futuro.

Estas variables ocuparán en memoria un cierto número de bytes, que depende del tipo de valor del que se trate (por ejemplo, en general en distintos lenguajes de programación un valor de tipo entero ocupa entre uno y ocho bytes en memoria, un valor de tipo real o de coma flotante, con punto y parte decimal, ocupa entre cuatro y ocho bytes, y un carácter ocupa uno o dos bytes dependiendo de la codificación empleada) y Java no es diferente en este aspecto.

Por ello antes de poder utilizar una variable deberemos *declarar* esa variable y qué hacemos cuando declaramos una variable, específicamente unimos los tres componentes de esta: el nombre o identificador a partir del cual luego nos referiremos a la variable por un lado, el tipo de datos que la variable puede contener por el otro y finalmente, aunque transparente para nosotros, la memoria que esa variable dispondrá para almacenar los valores que asociemos a ella.

Bien vamos por partes, cuáles son los tipos de variables existentes en java:

- Tipos enteros

Tipo	Descripción	Bytes ocupados	Literales	Rango de Valores
byte	número entero de un byte	1	número entre -128 y 127	[-128 a 127]
short	número entero de dos bytes	2	número entero en el rango	[-32768 a 32767]

Tipo	Descripción	Bytes ocupados	Literales	Rango de Valores
int	número entero de cuatro bytes	4	número entre en el rango	[-2^31 a 2^31-1]
long	número entero de ocho bytes	8	número entero	

- Tipos decimales

Tipo	Descripción	Bytes ocupados	Literales	Rango de Valores
float	número decimal simple	4	3.5f	simple presición 6 o 7 decimales
double	número decimal preciso	8	0.123456789	doble presición 14 o 15 decimales

- Otros tipos

Tipo	Descripción	Bytes ocupados	Literales	Rango de Valores
char	número sin signo que representa el código de un carácter	dependiente de la codificación	'A'	dependiente de la codificación
boolean	valor lógico	1	true o false	[true:false]
String	cadena de caracteres	dependiente de la cantidad de caracteres de la cadena y la codificación	"Hola"	

Ahora bien, conocemos algunos posibles tipos de datos pero cómo declaramos variables, simplemente anteponemos el tipo al nombre de la variable y con eso hemos declarado una variable y cubierto los tres vértices del triángulo que antes mencionamos tipo - nombre - memoria.

Por ejemplo algunas sentencias de declaración de variables:

```
int x1;
float x2, x3;
char c;
String nom;
```

Nota: Un elemento a notar en las sentencias anteriores es que **String** no solo comienza con mayúscula sino que además no está marcada de color como **int**, **float** o **char**, la razón de esto es que String es una clase y como más adelante veremos en ese caso estamos declarando una referencia.

Sin embargo, y debido al uso generalizado de las cadenas de caracteres en los lenguajes de programación String se vuelve necesario y por eso tiene un esquema especial de instanciación en Java.

Con las variables ya declaradas se pueden comenzar a usar asignando valores como hasta ahora lo hemos hecho en los demás lenguajes de programación.

Algunos ejemplos:

```
// Declaración de variables
char c;
int a;
String nom1, nom2;

// Asignaciones
c = '+';
a = c; // Notar la conversión, ya que como habíamos dicho un carácter es una
variable que almacena el código de la letra contenida.
a = 'c'; // Igual al caso anterior.
nom1 = "José";
nom2 = n1; // Notar que estoy haciendo que nom2 tenga el mismo valor que nom1
nom2 = "nom1"; // Aquí en cambio estoy dando a nom2 la cadena "nom1" y no el
valor de la variable

// Notar que también se podría hacer ambas cosas en la misma sentencia
int x = 10;
```

Nomenclatura de identificadores para variables, el nombre de las variables es elección del desarrollador pero debe cumplir ciertas restricciones y convenciones (en el Apunte 2 hicimos una aclaración de la diferencia de ambos conceptos), a continuación las principales restricciones y convenciones:

- **Restricción:** El nombre o identificador de una variable en Java, solo puede contener letras (mayúsculas y/o minúsculas), o también dígitos (0 al 9), o también el guion bajo (`) (también llamado guion de subrayado).
- **Restricción:** El nombre de una variable no debe comenzar con un dígito.
- **Restricción:** El nombre de una variable no puede ser una palabra reservada del lenguaje Java.
- **Nota:** Java es case sensitive, es decir, hace diferencia entre minúsculas y mayúsculas, por lo que toma como diferentes a dos nombres de variables que no sean exactamente iguales. El identificador contador no es igual al identificador Contador y Java tomará a ambos como dos variables diferentes.
- **Convención:** Java utiliza para los nombres de variable (como para casi todo), la notación camel case, es decir los identificadores se definen comenzando con minúscula siempre y se utiliza una sola letra mayúscula para indicar el comienzo de cada nueva palabra en el caso de un nombre compuesto.

Inferencia de tipos en literales y conversión de tipos en java

Como dijimos Java es un lenguaje tipado y por lo tanto realiza un constante chequeo de tipos de variables en cada momento de asignación u operación de valores en las variables mencionadas. Este chequeo se realiza a

tal punto que cuando asignamos un literal a una variable como por ejemplo:

```
int a;  
a = 15;
```

Java antes de hacer la asignación del 15 en la variable a, está infiriendo el tipo del 15 (para java todo número entero es de tipo int) y luego al evaluar que la asignación es posible sin pérdida de precisión se realiza.

Pero en este otro caso:

```
float b;  
// b = 3.5; // esta línea provocaría un error de compilación de posible  
pérdida de precisión  
b = 3.5f
```

Java infiere el tipo de 3.5 como double ya que para java cualquier valor decimal es double, por ello tengo que intervenir y expresar que asumo que ese valor va a ser tratado como float agregando la letra f pegada al final del número.

Ahora bien, el problema se complica cuando queremos hacer asignaciones de valores de variables en otras variables, y aquí pueden ocurrir 3 casos diferentes. El primero y más feliz es que las variables a ambos lados del signo igual sean del mismo tipo, por ejemplo:

```
int a, b;  
a = 10;  
b = a;
```

Java realiza el chequeo de tipos y entiende que no hay pérdida de precisión y por lo tanto realiza la asignación sin advertencia alguna. El segundo caso es que la variable a la izquierda del signo igual tenga mayor cantidad de memoria disponible que la variable a la derecha del signo igual, por ejemplo:

```
short c = 15;  
int a;  
a = c;
```

Nota: notar la situación contradictoria que se da en la asignación del 15, anteriormente dijimos que java asume los literales enteros como valores int y aquí estaríamos asignando un int en un short, sin embargo al ser un literal java puede hacer el control en tiempo de compilación y por lo tanto determina que la asignación no implica pérdida de precisión y permitirla.

Lo que en este caso sucede es que java detecta que no hay pérdida de precisión debido a que el short ocupa menos memoria que el int y por lo tanto, realiza una promoción implícita del valor de c a int y realiza la asignación con valores del mismo tipo.

Finalmente llegamos al caso crítico, donde el tamaño en memoria de la variable ubicada a la izquierda del signo igual es **menor** que el tamaño de memoria de la variable ubicada a la derecha. En este caso se podría producir una pérdida de precisión porque podríamos tener un valor más grande en la variable de la derecha y java no podrá resolver qué poner en la variable de la izquierda, aquí el compilador Java nos va a requerir una acción explícita que se denomina **Casting** o conversión de tipo explícito, e implica anteponer el nombre del tipo al que quiero cambiar entre paréntesis al nombre de la variable a la que quiero cambiar el tipo en la asignación u operación. Por ejemplo:

```
int a = 123456;
short b;
// b = a; // esto provocaría error de compilación Possible lost of precision.
// Entonces tenemos que castear a la variable a
b = (short) a; // Esto sí compila
```

Sin embargo, si tenemos que adivinar ¿qué valor queda en la variable b?, lo primero que se nos podría ocurrir es que b queda valiendo el máximo para el tipo short: 32767, pero si probamos el bloque anterior nos encontramos con que no es así, b queda valiendo: -7616. ¿Qué pasó aquí?

En realidad lo que pasó es que nosotros le dimos expresa instrucción a Java de que el valor de a debía ser entendido como entero, y eso hizo que java simplemente tome los dos bytes menos significativos de a y copie su contenido en b, con lo que copió los bits que estaban en esos bytes y esos bits conforman un número negativo.

Caso particular, uso de var

La palabra reservada **var** es una característica introducida en Java 10 (LTS Java 11), que permite declarar variables locales de manera más concisa y flexible. Con var, el tipo de la variable se infiere automáticamente por el compilador en función del valor asignado a la variable.

Cuando utilizas la palabra clave **var** para declarar una variable, el compilador determina automáticamente el tipo de datos de esa variable basándose en el valor al que se le asigna. Esto significa que no es necesario especificar explícitamente el tipo de datos, lo que puede hacer que el código sea más claro y menos repetitivo. Por ejemplo

```
var edad = 25; // La variable "edad" es de tipo int
var nombre = "John"; // La variable "nombre" es de tipo String

System.out.println("Nombre: " + nombre + ", Edad: " + edad);
```

En este ejemplo, las variables `edad` y `nombre` son declaradas utilizando la palabra clave `var`. El compilador infiere automáticamente que `edad` es de tipo `int` debido a su valor numérico, y que `nombre` es de tipo `String` debido a su valor de texto.

`var` solo puede usarse para declarar variables locales dentro de métodos o bloques. No se puede usar para declarar parámetros de métodos, variables de instancia o variables de clase. **var no significa "variant"** (variable que puede cambiar de tipo). Una vez que se infiere un tipo para una variable con `var`, ese tipo es fijo. Se recomienda usar `var` con moderación y solo cuando el tipo de datos es obvio o cuando la inferencia de tipo mejora la legibilidad del código.

Ventajas de var:

- Reduce la repetición de tipos verbosos.
- Hace que el código sea más conciso y limpio.
- Facilita el mantenimiento del código al cambiar los tipos de variables sin necesidad de modificar las declaraciones de tipo.

Desventajas de var:

- Puede reducir la claridad si se usa de manera excesiva o en contextos donde el tipo no es obvio.
- Puede dificultar la comprensión del tipo de datos si el valor inicial es nulo o ambiguo.
- En resumen, la palabra clave `var` en Java permite la inferencia automática de tipos para variables locales. Ayuda a hacer el código más conciso y limpio, pero debe usarse con moderación y en situaciones donde el tipo sea obvio para mejorar la legibilidad.

Operadores aritméticos en Java

Hemos visto que se puede asignar en una variable un valor, también se puede asignar el resultado de una expresión. Una expresión es una fórmula en la cual se usan operadores (como suma o resta) sobre diversas variables y constantes (que reciben el nombre de operandos de la expresión). Si el resultado de la expresión es un número, entonces la expresión se dice expresión aritmética. Los siguientes es un ejemplo de una expresión aritmética en Java:

```
int suma, num1 = 10, num2 = 7;  
suma = num1 + num2;
```

En la última línea se está asignando en la variable `suma` el resultado de la expresión `num1 + num2`; y obviamente la variable `suma` quedará valiendo 17. Note que en una asignación primero se evalúa cualquier expresión que se encuentre a la derecha del signo `=`, y luego se asigna el resultado obtenido en la variable que esté a la izquierda del signo `=`. La siguiente tabla muestra los principales operadores aritméticos del lenguaje Java (volveremos más adelante con un estudio más detallado sobre la aplicación de estos operadores):

Operador	Significado	Ejemplo de uso
<code>+</code>	suma	<code>a = b + c;</code>

Operador	Significado	Ejemplo de uso
-	resta	$a = b - c;$
*	producto	$a = b * c;$
/	división	$a = b / c;$
%	resto de una división	$a = b \% c;$

En Java los distintos operadores aritméticos actúan de acuerdo al tipo de las variables o constantes sobre las que operan. Así, si se usa el operador suma (+) para sumar dos variables int, el resultado será un valor int, y lo mismo ocurrirá con los demás.

Aunque lo anterior es lo lógico, no siempre el resultado obtenido será el que hubiésemos esperado: si se usa el operador división (/) y los dos números que se dividen son de tipo int, entonces el operador calculará la llamada división entera (o cociente entero) entre ambos, lo cual significa que la parte decimal del resultado será truncada. Veamos el siguiente ejemplo:

```
int a, b, c;
a = 5;
b = 2;
c = a / b;
```

El valor de c, será de 2, y no de 2.5, ya que la división se calculó en forma entera al ser de tipo int las variables a y b. Lo que java hace se denomina inferencia de tipo resultante de la operación y se basa en algunas reglas simples:

- Si los operadores son byte, short o int, el resultado será int aunque todos los operadores sean byte o short.
- A partir de int java asume el mayor tipo de los operadores involucrados según el siguiente orden:
 - `int < long < float < double`

En ese sentido, el operador resto (%) es muy útil porque permite calcular el resto de una división entera (y esto a su vez es muy valioso en casos en que se quiere aplicar conceptos de divisibilidad): la expresión $r = x \% y$; calcula el resto de dividir en forma entera a x por y, y asigna ese resto en la variable r. En el caso citado antes, si los valores ingresados fueran x = 11 y y = 2, el resto calculado sería 1.

Visualización por pantalla

Al ejecutar un programa, lo normal es que antes de finalizar el mismo muestre por pantalla los resultados obtenidos. En el lenguaje Java la instrucción más básica para hacer eso es `System.out.print()`. Esta instrucción permite mostrar en pantalla tanto el contenido de una variable como también mensajes formados por cadenas de caracteres, lo cual es útil para lograr salidas de pantalla “amigables” para quien use nuestros programas. La forma de usar la instrucción se muestra en los siguientes ejemplos:

```
public class App {  
    public static void main(String[] args) {  
        // declaramos e inicializamos variables  
        int a, b;  
        a = 3;  
        b = a + 1;  
        // mostramos el resultado por pantalla  
        System.out.println(b);  
  
    } // fin del main  
} // fin del class
```

Aquí, la instrucción de la línea 12), muestra en pantalla el valor contenido en la variable b, o sea, el número 4. Sin embargo, una salida más elegante sería acompañar al valor en pantalla con mensaje aclaratorio:

```
System.out.print( "El resultado es: " + b );
```

Observar que ahora no sólo aparecerá el valor contenido en b, sino también la cadena "El resultado es: ", precediendo al valor. Notar también que para mostrar el valor de una variable, el nombre de la misma no lleva comillas, pues en ese caso se tomaría al nombre en forma literal. La instrucción que sigue, muestra literalmente la letra 'b' en pantalla: `System.out.print("b");`

Entrada de Datos por teclado en Java

Esta operación permite mayor generalidad en la carga de datos de un programa. En el ejemplo anterior, se analizó un programa que permitía sumar dos números, en base al esquema de asignación directa de valores.

Podemos darnos cuenta rápidamente que un programa así planteado es muy poco útil, pues indefectiblemente el resultado mostrado en pantalla será 4... El programa tiene muy poca flexibilidad debido a que el valor inicial de la variable a es siempre 5 y el de b es siempre 3. Lo ideal sería que mientras el programa se ejecuta, pueda pedir que el usuario ingrese por teclado un valor para la variable a, luego otro para b, y que luego se haga la suma (en forma similar a como permite hacerlo una calculadora...)

Java a partir de la versión 5 provee una clase que está preparada para capturar una entrada por teclado en la consola estándar, de forma simple y sin necesidad de mayores conceptos previos. Java provee la herramienta que analizaremos a continuación como alternativa básica. Esta es la clase Scanner, la cual entre otras varias y amplias funcionalidades que implementa agrega también la posibilidad de realizar de manera simple y concreta la lectura de valores numéricos, caracteres o cadenas de caracteres desde teclado a través de la consola estándar.

Para utilizar la clase Scanner lo único especial que hay que agregar es la siguiente línea de código que debe quedar antes de la declaración de la clase en el archivo de código. `import java.util.Scanner;`

Los métodos de la clase Scanner que nos interesan en este punto, los presentamos a continuación y cabe aclarar que la clase Scanner tiene varios métodos más, que no nos interesa analizar por ahora:

Tipo de retorno	Método	Descripción
<code>String</code>	<code>nextLine()</code>	Retorna la próxima carga de cadena de caracteres.

Tipo de retorno	Método	Descripción
boolean	nextBoolean()	Retorna la próxima carga como un valor booleano.
byte	nextByte()	Retorna la próxima carga como un valor de tipo byte.
double	nextDouble()	Retorna la próxima carga como un valor de tipo double.
float	nextFloat()	Retorna la próxima carga como un valor de tipo float.
int	nextInt()	Retorna la próxima carga como un valor de tipo int.
long	nextLong()	Retorna la próxima carga como un valor de tipo long.
short	nextShort()	Retorna la próxima carga como un valor de tipo short.

Para utilizar estos métodos es necesario realizar algunas tareas previas, en primer lugar, debemos crear el escáner, es decir debemos crear el objeto que va a escanear la entrada estándar para luego utilizar los métodos, ese bloque de código quedaría como sigue:

```
import java.util.Scanner;

public class App {
    public static void main(String[] args) {
        // Declarar la referencia a Scanner y crear la instancia que tome la
        // entrada estándar
        Scanner miEscaner = new Scanner(System.in);
        // Declara las variables a y b
        int a, b;

        System.out.print("Ingrese el valor de a: ");
        a = miEscaner.nextInt();
        System.out.print("Ingrese el valor de b: ");
        a = miEscaner.nextInt();

        // mostrar el resultado
        System.out.println("La suma es: " + (a+b));
    }
}
```

Cada uno de esos métodos se invoca escribiendo primero el nombre del objeto de la clase Scanner que creamos en la primera línea (en el ejemplo referenciado por `miEscaner`) seguido de un punto, y luego el nombre del método. Cada método, al ejecutarse, provoca que el programa entre en modo de espera, sin ejecutar ninguna otra instrucción hasta que alguien ingrese por teclado un valor y se presione la tecla `Enter`.

El valor así ingresado, se asigna en la variable indicada en la instrucción a la izquierda del signo igual, y luego prosigue normalmente el programa. En el ejemplo anterior, el segmento de programa mostrado provoca la carga por teclado de un número entero en la variable `a`, y otro en la variable `b` para hacer la suma pero ahora de los valores ingresados por teclado.

Estructura alternativa o condicional

En problemas que no sean absolutamente triviales, es muy común que en algún punto se requiera comprobar el valor de alguna condición, y en función de ello proceder a dividir la lógica del algoritmo en dos o más ramas o caminos de ejecución. Por ejemplo, en un programa de control de acceso a un lugar seguro se debe pedir a cada usuario que cargue su clave de identificación. El programa entonces debería controlar si la clave cargada es correcta y sólo en ese caso habilitar el paso a esa persona. Pero si la clave fuese incorrecta, el programa debería tomar alguna medida alternativa, como sacar un mensaje de alerta por la consola de salida, bloquear una puerta, dar aviso a un supervisor, etc. Pero el hecho es que si sólo se emplean estructuras secuenciales de instrucciones, la situación anterior no podría resolverse.

Para casos así, los lenguajes de programación proveen instrucciones específicamente diseñadas para el chequeo de una o más condiciones, permitiendo que el programador indique con sencillez lo que debe hacer el programa en cada caso. Esas instrucciones se denominan estructuras condicionales, o bien, instrucciones condicionales.

En general, una instrucción condicional contiene una expresión lógica que puede ser evaluada por verdadera o por falsa, y dos bloques de instrucciones adicionales designados en general como la salida o rama verdadera y la salida o rama falsa. Si un programa alcanza una instrucción condicional y en ese momento la expresión lógica es verdadera, el programa ejecutará las instrucciones de la rama verdadera (y sólo esas). Pero si la expresión es falsa, el programa ejecutará las instrucciones de la rama falsa (y sólo esas).

En el lenguaje Java, una instrucción condicional típica como la que mostramos en la figura anterior, se escribe (esquemáticamente) así:

```
if (expresión lógica) {  
    // instrucciones de la rama verdadera  
}  
else {  
    // instrucciones de la rama falsa  
}
```

La palabra reservada **if** da inicio a la estructura condicional que posee básicamente dos salidas o ramas: la rama o salida por verdadero y la rama o salida por falso. A continuación se escribe la condición que se evalúa, **siempre** entre paréntesis, y es la *expresión lógica* que se quiere evaluar por verdadero o falso, recordemos en este contexto, que una expresión lógica es una fórmula cuyo resultado es un valor lógico (o valor de verdad).

La "rama verdadera" se escribe entre llaves, inmediatamente después de cerrar el paréntesis de la condición, y la "rama falsa" va después de la rama verdadera, también envuelta entre llaves, pero precedida de la palabra reservada **else**. Si al evaluar la condición la misma es cierta, se ejecuta únicamente el bloque de instrucciones encerrado entre llaves de la rama verdadera, y se ignora la rama falsa. Si la condición fuera evaluada por falso, se ejecutaría únicamente el bloque de la rama **else**, y será ignorada la rama verdadera.

Expresiones lógicas, operadores relacionales y conectores lógicos en Java

En una lección anterior hemos visto que en general, una **expresión** es una fórmula compuesta por variables y constantes (llamados operando) y por símbolos que indican la aplicación de una acción (llamados operadores). Hemos analizado también el uso de los llamados operadores aritméticos básicos de Java (suma, resta, producto, etc.) y sabemos que en función de esto, una expresión aritmética es una expresión cuyo resultado es un número.

Ahora bien, el hecho de que una expresión entregue como resultado un número, se debe a que los operadores que aparecen en ella son operadores aritméticos y por lo tanto llevan a la realización de alguna operación cuyo resultado será numérico. Sin embargo, en todo lenguaje existen operadores cuya acción no implica la obtención de un número como resultado, sino, por ejemplo, valores lógicos de la forma verdadero o falso (**true** o **false** en Java) y algunos otros operadores entregarán resultados de otros tipos (cadenas de caracteres, por ejemplo). En ese sentido, como ya hemos indicado, una expresión lógica es una expresión cuyo resultado esperado es un valor de verdad (**true** o **false**).

Nota: en Java, a diferencia de otros lenguajes de programación, las sentencias que requieren un valor lógico, **SOLO** aceptan valores **boolean** es decir, no es válido utilizar valores de otro tipo para que el lenguaje los asuma verdaderos o falsos.

Como vimos, las instrucciones condicionales (y otros tipos de instrucciones que veremos, como las instrucciones repetitivas) se basan típicamente en chequear el valor de una expresión lógica para determinar el camino que seguirá el programa en su ejecución.

Para el planteo de expresiones lógicas, todo lenguaje de programación provee operadores que implican la obtención de un valor de verdad como resultado. Los más elementales son los llamados operadores relacionales u operadores de comparación, que en Java son los siguientes:

Operador	Significado	Ejemplo	Observaciones
<code>==</code>	igual que	<code>a == b</code>	retorna true si <code>a</code> es igual que <code>b</code> , o false en caso contrario
<code>!=</code>	distinto de	<code>a != b</code>	retorna true si <code>a</code> es distinto de <code>b</code> , o false en caso contrario
<code>></code>	mayor que	<code>a > b</code>	retorna true si <code>a</code> es mayor que <code>b</code> , o false en caso contrario
<code><</code>	menor que	<code>a < b</code>	retorna true si <code>a</code> es menor que <code>b</code> , o false en caso contrario
<code>>=</code>	mayor o igual que	<code>a >= b</code>	retorna true si <code>a</code> es mayor o igual que <code>b</code> , o false en caso contrario
<code><=</code>	menor o igual que	<code>a <= b</code>	retorna true si <code>a</code> es menor o igual que <code>b</code> , o false en caso contrario

Los operadores de la tabla anterior permiten plantear instrucciones condicionales en java para comparar de distintas formas dos valores.

También es posible emplear operadores conocidos como conectores lógicos para poder chequear varias expresiones lógicas a la vez. En general, cada una de las expresiones encadenadas por un conector lógico se designa como una proposición lógica. Por lo tanto, una proposición lógica es una expresión formada por variables y/o constantes relacionadas entre sí mediante operadores de comparación (o relacionales), de tal forma que el resultado de la expresión será un verdadero o un falso. Los tres principales conectores lógicos en Java se ven en la tabla siguiente

Operador	Significado	Ejemplo	Observaciones
<code>&&</code>	conjunción lógica (y)	<code>a == b && y != x</code>	ver revisar tablas de verdad para estimar el resultado
<code> </code>	disyunción lógica (o)	<code>n == 1 n == 2</code>	ver revisar tablas de verdad para estimar el resultado
<code>!</code>	negación lógica (no)	<code>! x > 7</code>	ver revisar tablas de verdad para estimar el resultado

Un conector lógico u operador booleano es un operador que permite encadenar la comprobación de dos o más expresiones lógicas y obtener un resultado único. En general, cada una de las expresiones lógicas encadenadas por un conector lógico se designa como una proposición lógica. En la columna **Ejemplo** de la tabla anterior, las expresiones `a == b`, `y != x`, `n == 1`, `n == 2` y `x > 7` son proposiciones lógicas.

Variantes de la expresión condicional en Java

Condicional Doble

Condicional Simple

También puede ocurrir (y de hecho es muy común) que para una condición sólo se especifique la realización de una acción si la respuesta es verdadera y no se requiera hacer nada en caso de responder por falso. Para estos casos, en Java y otros lenguajes es perfectamente válido escribir una instrucción condicional que sólo tenga la rama verdadera, omitiendo por completo la falsa. Una instrucción condicional de ese tipo se suele designar como condición simple, y en ella no se especifica la rama else: la instrucción condicional termina cuando termina la rama verdadera. La forma general típica de una instrucción condicional simple` en Java es la siguiente:

```
if (expresión lógica)
    //instrucciones de la rama verdadera
    //continuación del programa
```

Nota: si revisamos hay una sutil omisión en el fragmento anterior y es que no escribimos las llaves, eso es porque en el caso que el bloque de instrucciones a ejecutar incluya solo una instrucción las llaves se pueden omitir.

En este caso puntual, el bloque condicional termina después de la primera sentencia de código. Esto aplica también a los condicionales tradicionales tanto para la rama verdadera como para la rama falsa.

Las instrucciones de la rama verdadera se encolumnan en un bloque hacia la derecha encerradas entre llaves si son más de una, del mismo modo que en un condicional tradicional, pero al terminar este bloque se escriben directamente las instrucciones para continuar con el programa, en la misma columna del `if` inicial, sin escribir la rama `else`.

Condicional Múltiple

Como vimos en el ejemplo anterior, cuando es necesario evaluar un valor con más de dos elecciones posibles, se puede resolver con estructuras alternativas anidadas o en cascada, o si se quiere con estructuras simples en secuencia. Sin embargo, si se tiene un problema donde el número de alternativas es grande, usar estos métodos, puede plantear serios problemas de escritura del algoritmo y naturalmente de legibilidad. Es por eso que existe una estructura llamada selección, decisión o condicional múltiple que considera estos casos en particular.

La estructura de decisión múltiple evaluará una expresión que podrá tomar n valores distintos: 1, 2, 3,..., n. Según que elija uno de estos valores en la condición, se realizará una de las n acciones, o lo que es igual, el flujo del algoritmo seguirá un determinado camino entre los n posibles.

En Java, la sintaxis para esta estructura es la siguiente:

```
switch (expresión_entera) {  
  
    case (valor1) :  
        // instrucciones para valor 1  
        break;  
  
    case (valor2) :  
        // instrucciones para valor 2  
        break;  
  
    ...  
  
    case (valorN) :  
        // instrucciones para valor N  
        break;  
  
    default:  
        // instrucciones si no cumple ninguna de las anteriores  
}
```

Nota: este es el caso tradicional de switch, a partir de la versión 7 de java también se pueden usar cadenas de caracteres lo que, si bien agrega overhead por la comparación de cadenas, hace extremadamente más legibles los fragmentos de código.

Además, a partir de la próxima versión LTS (Java 21), también se incluirá la posibilidad de utilizar expresiones regulares en las ramas del switch

Creo sin embargo, que una de las particularidades clave a destacar aquí es que la estructura switch en java funciona en cascada, esto es, si la expresión coincide con **valor2** pero yo omito intencionalmente codificar los break, entonces java ejecutará todas las sentencias de la rama **case (valor2)** y todas las sentencias de las demás ramas hasta encontrar un break o el final de la estructura.

Operador ternario o condicional inline

El operador ternario, también conocido como operador condicional inline, es una estructura que permite tomar decisiones basadas en una condición y asignar valores diferentes a una variable en función de si la condición es verdadera o falsa. A diferencia de las estructuras if-else, el operador ternario es más conciso y se utiliza para expresar lógica condicional de manera más compacta.

El operador ternario es útil y debería ser utilizado siempre que tengamos condicionales que en cada una de sus ramas asignan a una misma variable un valor diferente. Por ejemplo:

```
int edad = 18;
String mensaje;

if (edad >= 18) {
    mensaje = "Mayor de edad";
}
else {
    mensaje = "Menor de edad";
}
```

En estos casos como veremos el operador ternario logra un fragmento de código más claro y legible. El operador ternario se compone de tres partes: la condición, el valor si la condición es verdadera y el valor si la condición es falsa. La sintaxis general es:

```
variable = (condición) ? valor_verdadero : valor_falso;
```

```
int edad = 18;
String mensaje = (edad >= 18) ? "Mayor de edad" : "Menor de edad";
```

En este ejemplo, se utiliza el operador ternario para asignar el valor de la variable status basado en la edad. Si la edad es mayor o igual a 18, la variable status tendrá el valor "Mayor de edad"; de lo contrario, tendrá el valor "Menor de edad". Y, como podemos ver, este fragmento cuando nos acostumbramos es mucho más legible y flexible que la versión original.

Ventajas del Operador Ternario:

- Concisión: El operador ternario permite expresar lógica condicional en una sola línea de código, lo que puede hacer que el código sea más limpio y conciso.
- Legibilidad: En algunos casos, el operador ternario puede hacer que la lógica condicional sea más legible que una estructura if-else.

Desventajas del Operador Ternario:

- Complejidad Limitada: El operador ternario es útil para decisiones simples, pero puede volverse menos legible y más complejo cuando se manejan decisiones más complicadas.

- Dificultad en el Mantenimiento: Un uso excesivo o mal considerado del operador ternario puede dificultar el mantenimiento del código y la comprensión de otros desarrolladores.

Consideraciones al Usar el Operador Ternario:

- Utiliza el operador ternario cuando la lógica condicional sea simple y fácilmente comprensible.
- No abuses del operador ternario para evitar comprometer la legibilidad del código.
- Si la lógica condicional es compleja, considera usar una estructura if-else en su lugar.
- En resumen, el operador ternario en Java es una forma concisa de expresar lógica condicional en una sola línea de código. Sin embargo, debe usarse con moderación y solo en casos donde la lógica sea simple y fácil de entender.

Operadores resumidos en Java

A medida que se avanza en el estudio y planteo de algoritmos y programas para problemas cada vez más complejos, se verá que en la mayoría de esos programas será necesario eventualmente llevar a cabo procesos de **conteo** (por ejemplo, determinar cuántas veces apareció un número negativo), o de **sumarización** (por ejemplo, determinar cuánto vale la suma de todos los valores que tomó la variable **x** a lo largo de la ejecución de programa, suponiendo que **x** cambia de valor durante esa ejecución). El primer caso se resuelve incorporando una variable de conteo (o simplemente un contador), y el segundo, incorporando una variable de acumulación (o simplemente un acumulador).

En ambas situaciones se trata de variables que en una expresión de asignación aparecen en ambos miembros: la misma variable se usa para hacer un cálculo y para recibir la asignación del resultado de ese cálculo. Los siguientes son dos ejemplos de expresiones de conteo o acumulación (en el primero, la variable **a** se usa como un contador, y en el segundo la variable **b** se usa como un acumulador):

```
a = a + 1  
b = b + x
```

En general, un contador es una variable que sirve para contar ciertos eventos que ocurren durante la ejecución de un programa. Intuitivamente, se trata de una variable a la cual se le suma el valor 1 cada vez que se ejecuta la expresión. Esto es así porque contar normalmente significa sumar 1.

Entonces, técnicamente, un contador es una variable que actualiza su valor en términos de su propio valor anterior y de 1.

La sentencia resumida para expresar la expresión de conteo en java, es similar a otros lenguajes como python y puede expresarse así: **a += 1** la cual funciona así:

- En primer lugar se ejecuta la parte derecha de la asignación, con el valor actual de **a**. Si **a** comenzó valiendo cero, entonces la primera vez que se ejecute la expresión **a + 1** se obtiene un uno.
- En segundo lugar se asigna el valor así obtenido en la misma variable **a**, con lo cual se cambia el valor original.

Del mismo modo que en la expresión `a = a + 1` la variable `a` funciona como contador, de forma similar la variable `c` en la expresión `c = c - 1` funciona como decrementador: va restando de `a` uno a partir del valor original de la variable `c`. En los siguientes ejemplos mostramos contadores de formas diversas (asegúrese de entender lo que cada instrucción hace cada vez que se ejecuta):

```
a = a + 1  
b = b - 1  
c = c + 2  
d = d * 4  
e = e / 3
```

Por otra parte, un acumulador o variable de acumulación es básicamente una variable que permite sumar los valores que va asumiendo otra variable o bien otra expresión en un proceso cualquiera. Técnicamente, y en general, un acumulador es una variable que actualiza su valor en términos de su propio valor anterior y el valor de otra variable u otra expresión.

Según la definición dada de un acumulador, las siguientes expresiones también son expresiones de acumulación:

```
s = s + x  
b = b * z  
a = a - y  
p = p / t  
c = c + 2*x
```

Es interesante notar que en Java (como en otros lenguajes) cualquier expresión de conteo o de acumulación_responde a la forma general siguiente:

`variable = variable operador expresión`

donde `variable` es la variable cuyo valor se actualiza (y aparece en ambos miembros de la expresión de asignación) y `expresión` es una constante, una variable o una expresión propiamente dicha (formada a su vez por constantes, variables y operadores). Y el hecho es que en el lenguaje Java cualquier expresión que venga escrita en la forma general anterior, se puede escribir también en la forma resumida siguiente:

`variable operador= expresión`

A modo de ejemplo, veamos las siguientes equivalencias:

Forma general	Forma resumida
<code>a = a + 1</code>	<code>a += 1</code>
<code>b = b - 1</code>	<code>b -= 1</code>
<code>c = c + 2</code>	<code>c += 2</code>

Forma general	Forma resumida
----------------------	-----------------------

<code>d = d * 3</code>	<code>d *= 3</code>
<code>e = e / 4</code>	<code>e /= 4</code>
<code>s = s + x</code>	<code>s += x</code>
<code>b = b * z</code>	<code>b *= z</code>
<code>a = a - x</code>	<code>a -= x</code>
<code>p = p / t</code>	<code>p /= t</code>
<code>c = c + 2*x</code>	<code>c += 2*x</code>

Operadores unarios, pre y post incremento / decremento

Siguiendo el esquema anterior pero yendo más allá aún, tenemos dos casos particulares que se dan para el contador, es decir el incremento de una variable en términos de su propio valor y 1 más, o el decremento de una variable en términos de su propio valor y 1 menos, según lo visto en esos casos tendríamos:

```
a += 1
b -= 1
```

Para estos casos particulares java agrega 4 operadores extra, en realidad dos operadores que pueden ser utilizados en dos formas distintas cada uno. A saber:

Forma resumida	Operador unario
-----------------------	------------------------

<code>a += 1</code>	<code>a++ o ++a</code>
<code>b -= 1</code>	<code>b-- o --b</code>

Estoas dos formas son denominadas pre y post incremento para el operador `++` y pre y post decremento para el operador `--`, la primera pregunta que surge es: ¿qué sentido tiene que existan dos? y la primera respuesta que surge es que más allá de alguna discusión muy fina que por ahí leí, si están como una sentencia independiente no tienen diferencia alguna.

Ahora la cosa cambia cuando la expresión de incremento está a la derecha del signo igual o como parámetro de un método. En esos casos el resultado puede no ser tan claro:

Valores iniciales	Sentencia	Valores finales
--------------------------	------------------	------------------------

<code>a = 5</code>	<code>c = ++a</code>	<code>a => 6 ; C => 6</code>
<code>a = 5</code>	<code>c = a++</code>	<code>a => 6 ; C => 5</code>
<code>b = 5</code>	<code>c = --b</code>	<code>b => 4 ; C => 4</code>
<code>b = 5</code>	<code>c = b--</code>	<code>b => 4 ; C => 5</code>

La posición del operador define el orden de ejecución de la sentencia, si es pre incremento o pre decremento, primero se va a llevar la modificación de la variable incrementada y luego la asignación y en caso de ser post incremento o post decremento primero se va a llevar a cabo la asignación y luego la modificación de la variable incrementada.

Estructura repetitiva

Existen dos tipos generales de ciclos, los cuales se designan con los siguientes nombres genéricos (el motivo de estos nombres se verá oportunamente en esta misma lección):

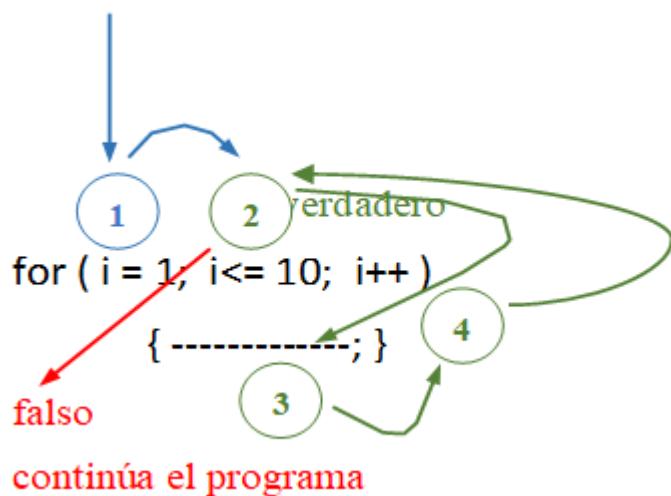
- Ciclos "0 – N"
- Ciclos "1 – N"

Estos ciclos son implementados por los diversos lenguajes en formas que varían ligeramente de un lenguaje a otro. El lenguaje Java implementa los dos tipos de ciclos mediante tres instrucciones específicas, designándolos respectivamente con las palabras reservadas que se marcan a continuación en letra negrita:

- Ciclo **for** (ciclo del tipo "0-N")
- Ciclo **while** (ciclo del tipo "0-N")
- Ciclo **do while** (ciclo del tipo "1-N")

El ciclo for en Java

El ciclo for comúnmente se usa en circunstancias en las cuales se conoce de antemano la cantidad exacta de repeticiones que deben realizarse, aunque en el lenguaje Java la sintaxis del ciclo es tan amplia que admite ser usado en cualquier circunstancia que requiera repetición de acciones. De hecho, veremos que un ciclo for es una variante de un ciclo while_. Un ejemplo con la estructura de un for se ve en el modelo siguiente, suponiendo que se desea repetir 10 veces la ejecución de ciertas instrucciones (otros casos requieren variantes directas sobre la estructura mostrada):



En la cabecera del ciclo existen tres secciones: la primera (`i=1` en este caso) se llama sección de inicialización, y se usa para indicar el valor inicial de las variables de control del ciclo. La segunda (`i<=10` en este caso) es la condición de control del ciclo: si la condición es verdadera el ciclo ejecuta una repetición del bloque de acciones, pero si es falsa corta el ciclo y continúa el programa. La tercera sección (`i++` en este caso) es la

sección de incremento, en la cual se indica la forma en que cambiará el valor de cada variable de control del ciclo.

Cuando el ciclo comienza, se ejecuta primero (y por única vez) la sección de inicialización, dando un valor inicial a la variable de control (ver acción (1) en el gráfico). Luego, se verifica la condición de control (2). Si la misma da "verdadero", se ejecuta entonces el bloque de acciones del ciclo (3), y al finalizar, se produce un retorno automático a la cabecera, específicamente a la sección de incremento, para cambiar el valor de las variables de control (4). Finalmente, se vuelve a verificar la condición de control. Si vuelve a dar "verdadero", se repite lo anterior, y si da "falso", el ciclo se detiene.

Observar que si la condición de control fuera falsa la primera vez que se evalúa, entonces el bloque de acciones no sería ejecutado. Esta es la característica básica de los ciclos tipo "0-N": el cero hace referencia a la cantidad mínima de veces que se espera que el bloque del ciclo sea ejecutado, y la N es un valor genérico que se refiere a que una vez comenzado el ciclo, se espera que el bloque de acciones se ejecute N veces, siendo N un valor indefinido pero que se supone será conocido al momento de ejecutar el ciclo.

A modo de ejemplo el siguiente fragmento imprime por pantalla los números del 1 al 5 entre llaves y separados por comas:

```
System.out.print('{');
for (int i = 1; i <= 5; i++) {
    System.out.print(i);
    if (i < 5)
        System.out.print(',');
}
System.out.println('}'');
```

La salida sería:

```
{1, 2, 3, 4, 5}
```

El ciclo while en Java

En muchas ocasiones necesitamos plantear un ciclo que ejecute en forma repetida un bloque de acciones pero sin conocer previamente la cantidad de vueltas a realizar. Para estos casos la mayoría de los lenguajes de programación, y en particular Java, proveen un ciclo designado como ciclo **while**.

Como todo ciclo, un ciclo while está formado por una cabecera y un bloque o cuerpo de acciones, y trabaja en forma general de una manera muy simple. La cabecera del ciclo contiene una expresión lógica que es evaluada en la misma forma en que lo hace una instrucción condicional if, pero con la diferencia que el ciclo while ejecuta su bloque de acciones en forma repetida siempre que la expresión lógica arroje un valor verdadero. Así como un if hace una única evaluación de la expresión lógica para saber si es verdadera o falsa, un ciclo while realiza múltiples evaluaciones: cada vez que termina de ejecutar el bloque de acciones vuelve a

evaluar la expresión lógica y si nuevamente obtiene un valor verdadero repite la ejecución del bloque y así continúa hasta que se obtenga un falso.

La característica principal del ciclo while es que la condición de control se evalúa por primera vez antes de la primera ejecución del bloque de acciones. Al igual que en el ciclo for_, esto implica que si en la primera evaluación de la condición de control se obtiene un valor falso, entonces el bloque del ciclo no será ejecutado y por esta causa el ciclo es del tipo "0-N".

Por ejemplo, el siguiente fragmento implementa un ciclo que se ejecuta mientras el valor de la variable cont se mantenga menor o igual a 5. En cada vuelta, se muestra un mensaje que contiene el número de vuelta en que se encuentra el ciclo:

```
int cont = 0
while (cont <= 5) {
    cont++;
    System.out.println("Vuelta número: " + cont);
}
```

La salida por consola estándar si se ejecuta este programa, sería la siguiente:

```
Vuelta número: 1
Vuelta número: 2
Vuelta número: 3
Vuelta número: 4
Vuelta número: 5
```

La variable cont recibió un valor inicial (el cero en este caso) antes de comenzar la ejecución del ciclo (de no ser así, la evaluación de la condición la primera vez no tendría sentido...). El bloque del ciclo contiene varias instrucciones y figura encerrado entre llaves. Cuando el programa llega al ciclo por primera vez, evalúa la condición. En este caso, el valor inicial de cont es cero y la condición es cierta: cont es menor que 5. Siendo cierta la condición, se ejecuta el bloque de acciones, con lo cual cont suma 1 y luego se muestra el mensaje: Vuelta número: 1. Termina allí el bloque de acciones, pero como se trata de un ciclo, automáticamente el programa regresa a verificar otra vez la condición: si el nuevo valor de cont sigue siendo menor que 5, el bloque se ejecuta otra vez y así seguirá hasta que en algún retorno la condición sea falsa. Cuando cont tenga el valor 5, demostrará el mensaje Vuelta número: 5 y el ciclo se detendrá. En ese momento se ejecutarán las instrucciones que se hayan escrito debajo del ciclo.

El ciclo do while en Java

La característica básica del ciclo do while es que la condición de control se evalúa por primera vez después de ejecutar por primera vez el bloque de acciones. Esto implica que al menos una vez el bloque de acciones del ciclo siempre será ejecutado, independientemente del valor inicial de la condición de corte (por eso este ciclo es del tipo "1-N": una vez como mínimo se ejecuta el bloque, y luego puede llegar a N repeticiones). La estructura de un ciclo do while es la siguiente:

El esquema de funcionamiento del ciclo do while es el siguiente: cuando se llega a una línea que comienza con do_, se ejecuta el bloque de acciones que sigue a continuación, sin importar el valor de la condición de control. Luego de ejecutar ese bloque, se evalúa la condición de control. Si dicha condición arroja un falso, el ciclo corta y continúa el programa con la instrucción que esté debajo del ciclo. Si la condición se valúa en verdadero, se provoca un retorno automático a la línea marcada con do_ y se vuelve a ejecutar el bloque de acciones del ciclo. El proceso continuará de esta forma, hasta obtener un falso en la condición de control.

Un ejemplo general de ciclo **do ... while**

```
do {  
    // acciones a repetir  
} while(cond);
```

Elementos de control de ciclo Java

El bloque de acciones de un ciclo (*while o for*) en Java puede incluir una instrucción **break** para cortar el ciclo de inmediato sin retornar a la cabecera para evaluar la expresión lógica de control o incluir la sentencia **continue** para volver desde ese punto a la cabecera sin terminar el bloque de sentencias repetitivas.

El siguiente segmento tiene el objetivo mostrar el uso de **break** y **continue**:

```
System.out.print('{');  
for (int i = 1; i <= 10; i++) {  
    if (i % 2 == 0)  
        continue;  
    System.out.print(i);  
    if (i == 7)  
        break;  
    if (i < 10)  
        System.out.print(',');  
  
}  
System.out.println('}'');
```

Ahora es un poco menos intuitiva la salida esperada:

```
{1, 3, 5, 7}
```

Uso de Scanner para leer un archivo de texto

Como vimos en el apartado anterior la clase Scanner es capaz de configurarse con la entrada estándar de java para tomar lo que el usuario carga por teclado sin embargo el uso más común de la clase Scanner es el de la lectura de archivos o flujos de texto.

Es decir nosotros podemos tomar un archivo de texto com por ejemplo el archivo llamado datos.txt con el siguiente contenido:

```
1  
2  
3  
4  
5
```

Y procesarlo con a partir de una instancia de la clase Scanner de la siguiente manera:

```
import java.io.FileNotFoundException;  
import java.util.Scanner;  
import java.io.File;  
  
public class App  
{  
    public static void main( String[] args ) throws FileNotFoundException  
    {  
        File f = new File("datos.txt");  
        Scanner miEscaner = new Scanner(f);  
  
        while (miEscaner.hasNext()) {  
            System.out.println(miEscaner.nextInt());  
        }  
    }  
}
```

Con la salida que efectivamente pensamos que obtendríamos de manera intuitiva:

```
1  
2  
3  
4  
5
```

Novedades de Java moderno: `switch ->` y bloques de texto `"""`

Desde Java 14 en adelante (y consolidado en Java 21), Java ha incorporado mejoras que simplifican la escritura y lectura del código, sin perder tipado fuerte ni claridad.

🔗 Switch mejorado con `->`

La sintaxis tradicional de `switch` puede ser verbosa y propensa a errores si se omiten los `break`. A partir de Java 14 se puede usar una forma más compacta y segura:

```
String dia = "LUNES";

switch (dia) {
    case "LUNES", "MIERCOLES", "VIERNES" -> System.out.println("Día de cursado");
    case "SABADO", "DOMINGO" -> System.out.println("Fin de semana");
    default -> System.out.println("Día normal");
}
```

 No es necesario escribir `break` y se pueden agrupar múltiples valores en una sola rama con comas.

Bloques de texto multilínea con `"""`

Otra novedad muy útil es el uso de *text blocks* para definir cadenas de texto multilínea de forma clara:

```
String html = """
<html>
    <body>
        <h1>Bienvenidos</h1>
    </body>
</html>
""";
```



```
System.out.println(html);
```

 Este formato es ideal para definir código HTML, JSON, mensajes, o cadenas largas sin concatenaciones ni `\n`.

¿Por qué las usamos?

Característica	Desde versión	Ventaja didáctica
<code>switch -></code>	Java 14	Evita errores con <code>break</code> , más legible
Bloques <code>"""</code>	Java 15	Facilita trabajar con texto multilínea

 En esta materia vamos a usar estas características modernas porque hacen el código más legible, más seguro y más conciso. ¡Aprovechamos que Java 21 es LTS y lo permite!