

LUCAS EGA
ACADEMY

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

Java



Curso Completo de Java

© 2026 Esteban Gutiérrez · Colibrí Editorial EGA · Hecho en México

lucasegaacademy.com

ÍNDICE

PRÓLOGO.....	4
UNIDAD 1: Tus primeros pasos con Java.....	8
UNIDAD 2: Datos, operaciones y conversación.....	31
UNIDAD 3: Tu programa aprende a decidir.....	55
UNIDAD 4: Repite sin repetirte - Los ciclos.....	79
UNIDAD 5: Guarda muchos datos y organiza tu código.....	100
UNIDAD 6: Piensa en objetos - POO parte 1.....	123
UNIDAD 7: Hereda, adapta y contrata - POO parte 2.....	147
UNIDAD 8: Colecciones que crecen solas.....	169
UNIDAD 9: Que no se caiga, que se guarde, que corra en paralelo.....	191
UNIDAD 10: Proyectos reales y herramientas profesionales.....	216
APÉNDICE A: Referencia rápida de Java.....	240
APÉNDICE B: Catálogo de errores comunes.....	244
APÉNDICE C: 30 retos extra con soluciones.....	247
APÉNDICE D: Glosario general.....	252
APÉNDICE E: Instalación paso a paso (JDK, IntelliJ, MySQL, Maven).....	256
EXAMEN FINAL.....	260

PRÓLOGO

Bienvenido a tu libro de Java

Hola, programador. Si tienes este libro en tus manos (o en tu pantalla), es porque tomaste una de las mejores decisiones de tu vida: aprender Java. No exagero. Java es el lenguaje que mueve bancos, sistemas de gobierno, aplicaciones de Android, servidores de grandes empresas y hasta la sonda espacial que la NASA tiene rondando por ahí. Aprender Java no es aprender un lenguaje de moda: es aprender el lenguaje que paga las cuentas de millones de familias programadoras en todo el mundo.

Este libro nace del curso gratuito "Curso Completo de Java" de Lucas EGA Academy (lucasegaacademy.com). Si ya viste los videos del curso, este libro es tu compañero perfecto: aquí cada tema se explica a fondo, con el detalle que un video corto no alcanza a dar. Y si no has visto el curso, no te preocupes: el libro funciona por si solo, de principio a fin, sin necesidad de nada más que una computadora y ganas de aprender.

Mi nombre es Esteban, soy el creador de Lucas EGA Academy, y escribi este libro pensando en ti: en la persona que está sentada frente a su computadora a las 2 de la mañana porque quiere cambiar de vida. Te hablo de tu, directo, sin rodeos, como colega. Aquí no hay paja academica ni definiciones de diccionario que nadie entiende. Hay explicaciones claras, código que funciona, errores reales (los que yo cometi y los que tú vas a cometer) y mucha práctica.

Que vas a lograr con este libro

Al terminar este libro vas a poder:

- Instalar y configurar tu entorno de desarrollo Java como un profesional.
- Escribir programas de consola que pidan datos, tomen decisiones y repitan tareas.
- Organizar tu código en métodos, clases y objetos: la famosa Programación Orientada a Objetos (POO).
- Usar colecciones como ArrayList y HashMap para manejar datos de verdad.
- Manejar errores con excepciones para que tus programas no se caigan.
- Leer y escribir archivos, y correr tareas en paralelo con hilos.

- Armar proyectos completos: una calculadora, una agenda de contactos y una tienda con POO.
- Conectar Java con una base de datos MySQL usando JDBC.
- Usar Maven para manejar dependencias como en los trabajos reales.

No es una lista de deseos: es el temario real de las 10 unidades que siguen. Cada unidad termina con práctica resuelta y explicada, y al final del libro hay un examen de 50 preguntas para que pongas a prueba todo lo aprendido.

Cómo está organizado el libro

El libro tiene 10 unidades. Cada unidad agrupa 3 clases del curso de video, pero aquí el contenido esta expandido muchísimo más:

1. **Introducción de la unidad:** que vas a aprender y por qué importa en el mundo real.
2. **Explicación detallada:** la teoría con analogías, errores comunes y buenas prácticas. Esta es la parte larga, leela con calma.
3. **Ejemplos paso a paso:** código comentado que puedes copiar, correr y modificar.
4. **Práctica:** al menos 5 ejercicios por unidad, todos con solución explicada. Hazlos, no solo los leas.
5. **Resumen de la unidad:** lo esencial en una página, para repasar antes del examen.

Después de las 10 unidades viene el **Examen Final:** 50 preguntas de opción múltiple, verdadero/falso y código, con todas las respuestas explicadas.

Como estudiar este libro (leelo, es importante)

Te doy mi método, el que le doy a todos mis estudiantes:

1. **No leas como novela.** La programación se aprende con las manos en el teclado. Por cada ejemplo del libro, abre tu editor, escribe el código tú mismo (no copies y pegues) y corrélo. Verlo correr en tu pantalla vale más que leerlo 10 veces.
2. **Rompe el código a proposito.** Una vez que un ejemplo funciona, cambiale algo: un número, un signo, el orden. Mira que error sale. Leer mensajes de error es una habilidad de programador, y se entrena rompiendo cosas.
3. **Haz TODOS los ejercicios.** Los de la sección de práctica no son opcionales. La diferencia entre "ya entendí" y "ya se hacerlo" es hacer los ejercicios. Las soluciones

están explicadas, pero intenta primero sin verlas. Date al menos 20 minutos por ejercicio antes de rendirte.

4. **Una unidad por sentada.** No intentes leer las 10 unidades en un día. Tu cerebro necesita digerir. Una unidad por día (o por cada dos días) es un ritmo perfecto. En 2 o 3 semanas terminas el libro sabiendo Java de verdad.
5. **El examen final es tu graduación.** No lo hagas a la mitad del libro. Terminalo todo, descansa un día, y luego haz el examen completo como si fuera el real: sin ver las respuestas, con tiempo. Si sacas menos de 40 de 50, repasa las unidades donde fallaste y repítelo.
6. **Programa todos los días, aunque sea 30 minutos.** La constancia le gana al talento. 30 minutos diarios durante un mes te ponen por encima del 90% de la gente que "quiere aprender a programar".

Que necesitas para empezar

- Una computadora (Windows, Mac o Linux, cualquiera sirve).
- Conexión a internet (para descargar el JDK una sola vez).
- Un editor: te recomiendo **IntelliJ IDEA Community** (gratis) o **Visual Studio Code** con la extensión de Java (gratis). En la Unidad 1 te explicó paso a paso cómo instalar todo.
- Ganas. Eso no se descarga, pero tu ya las tienes porque estas leyendo esto.

Una última cosa antes de empezar

Java tiene fama de ser "difícil" o "verboso". Mentira. Java es explícito: te pide que digas las cosas claras, y eso al principio cansa pero después te salva. Cuando lleves 3 unidades vas a entender por qué los bancos confían sus millones a este lenguaje: porque Java no deja nada al azar.

Así que respira, abre tu editor, y empecemos. La Unidad 1 te espera, y con ella tu primer programa.

Nos vemos en el código.

Esteban Creador de Lucas EGA Academy lucasegaacademy.com

Convenciones usadas en este libro

Para que no te pierdas, así presento las cosas:

- Los **terminos importantes** aparecen en negritas la primera vez que los explicó.
- El ``codigo dentro del texto`` aparece entre comillas invertidas, como ``System.out.println``.
- Los programas completos van en bloques de código separados, listos para copiar.
- Las **notas de error comun** te avisan de las trampas donde todos caemos.
- Las **buenas prácticas** te enseñan a escribir código como un profesional, no como un estudiante.

Listo. A programar.

UNIDAD 1: Tus primeros pasos con Java

Introducción

Esta unidad es tu puerta de entrada. Aquí vas a entender qué es Java y por qué es tan importante, vas a instalar todo lo necesario en tu computadora, y vas a escribir y correr tu primer programa. Al terminar esta unidad, tendrás un entorno de desarrollo funcionando y ya habrás compilado código real. Parece poco, pero es el paso que separa a los que "quieren aprender" de los que ya están aprendiendo.

Java nació en 1995 en Sun Microsystems (hoy es de Oracle) y su lema lo dice todo: **"escribe una vez, corre en todas partes"**. Esto significa que un programa escrito en Java puede correr en Windows, Mac, Linux, celulares y servidores sin que tengas que reescribirlo. Esa promesa se cumple gracias a la **JVM (Java Virtual Machine)**, la máquina virtual que ejecuta tus programas. Tu código se compila a un formato intermedio llamado **bytecode**, y la JVM de cada sistema lo traduce a instrucciones de esa máquina. Piensalo así: tu escribes en un idioma universal, y cada país (cada sistema operativo) tiene su traductor.

Donde se usa Java hoy: sistemas bancarios (casi todos los bancos grandes del mundo corren Java en su núcleo), aplicaciones empresariales, Android (el sistema operativo de celular más usado del planeta nació con Java como su lenguaje principal), servidores web, Big Data (Hadoop y Spark están escritos en Java) y videojuegos como Minecraft. Cuando aprendes Java, aprendes el lenguaje de los sistemas que no se pueden caer.

Explicación detallada

Que es Java realmente

Java es un lenguaje de programación **orientado a objetos**, **tipado estático** y **compilado a bytecode**. Vamos por partes, porque cada una de esas palabras es importante:

Orientado a objetos significa que en Java todo se organiza en "objetos": piezas de software que tienen datos (atributos) y comportamientos (métodos). Un objeto `CuentaBancaria` tiene un saldo (dato) y puede depositar o retirar (comportamientos). En la Unidad 6 vas a dominar esto a fondo; por ahora quédate con la idea de que Java piensa en objetos.

Tipado estático significa que cada variable tiene un tipo fijo que se declara desde el principio: un `int` siempre guarda números enteros, un `String` siempre guarda texto. Esto parece

tedioso al principio, pero es una red de seguridad: el compilador te avisa de errores antes de que el programa corra. En lenguajes sin tipos, esos errores aparecen cuando el programa ya está corriendo frente al usuario. Los bancos prefieren que el error salga en la computadora del programador, no en la del cliente.

Compilado a bytecode significa que hay dos pasos: primero compilas tu código fuente (archivos `.java`) con el compilador `javac`, que produce archivos `.class` con bytecode; después la JVM ejecuta ese bytecode. Este proceso de dos pasos es lo que permite el "escribe una vez, corre en todas partes".

El ecosistema: JDK, JRE y JVM (no los confundas)

Esta es la confusión número uno de los principiantes, así que vamos a dejarla clara de una vez:

- **JVM (Java Virtual Machine):** la máquina virtual que ejecuta el bytecode. Es el motor.
- **JRE (Java Runtime Environment):** la JVM más las librerías básicas para correr programas. Es lo que necesita alguien que solo quiere *usar* un programa Java, sin programar.
- **JDK (Java Development Kit):** el JRE más el compilador `javac` y las herramientas de desarrollo. Es lo que necesitas TU, que vas a *programar*.

Regla de oro: **para programar en Java, instala el JDK**. El JDK ya incluye todo lo demás. Cuando descargues, busca "JDK 17" o "JDK 21" (versiones LTS, de soporte largo, las más estables para aprender y trabajar). Oracle ofrece el JDK gratis, y también existen distribuciones gratuitas como Temurin (de Eclipse Adoptium) o Amazon Corretto. Cualquiera sirve para este libro.

Cómo funciona un programa Java por dentro

El ciclo de vida de tu programa tiene 4 etapas:

1. **Escribes** el código fuente en un archivo con extensión `.java`, por ejemplo `HolaMundo.java`.
2. **Compilas** con `javac HolaMundo.java`. El compilador revisa que tu código este bien escrito (sintaxis, tipos) y genera `HolaMundo.class` con bytecode.
3. **Ejecutas** con `java HolaMundo`. La JVM carga el bytecode, lo verifica por seguridad y lo ejecuta.

4. Si hay un error de sintaxis, `javac` te lo dice en la etapa 2 y no se genera el `.class`. Si hay un error de lógica (el programa corre pero hace lo incorrecto), lo descubres en la etapa 3.

Esta separación entre compilar y ejecutar es una ventaja enorme: muchos errores se atrapan antes de correr nada. Acostumbrate a leer los mensajes del compilador; al principio parecen jeroglíficos, pero con práctica se vuelven tus mejores amigos.

Anatomía de tu primer programa

Mira el programa más famoso de la historia de la programación:

```
public class HolaMundo {  
    public static void main(String[] args) {  
        System.out.println("Hola, mundo!");  
    }  
}
```

Vamos línea por línea, porque cada palabra cuenta en Java:

- `public class HolaMundo {` - En Java **todo vive dentro de una clase**. Una clase es como el contenedor principal de tu código. `public` significa que es visible para todos. El nombre de la clase (`HolaMundo`) **debe coincidir exactamente** con el nombre del archivo: `HolaMundo.java`. Si el archivo se llama `holamundo.java` o `HolaMundo2.java`, el compilador se queja. Error común número uno: no respetar mayúsculas y minúsculas en el nombre del archivo.
- `public static void main(String[] args) {` - Este es el **método main**, el punto de arranque de todo programa Java. La JVM busca exactamente este método para empezar. Desglosemoslo:
 - `public`: visible para la JVM.
 - `static`: se puede llamar sin crear un objeto (en la Unidad 6 entenderas esto a fondo).
 - `void`: no devuelve ningún valor.
 - `main`: el nombre mágico que la JVM busca.
 - `(String[] args)`: recibe los argumentos de la línea de comandos como un arreglo de textos. Por ahora no lo usaremos, pero debe estar ahí.

Si escribes `Main` con mayúscula, o te falta `static`, o pones `string` en minúscula, el programa compila pero al correrlo la JVM dice "no encuentre el método main". Es el error clásico del principiante. Copia esta línea exacta hasta que te la sepas de memoria.

- `System.out.println("Hola, mundo!");` - La instrucción que imprime texto en la consola.
- `System` es una clase del sistema, `out` es la "salida estandar" (tu pantalla), y `println` es el método que imprime una línea (el `\n` es de "line", salto de línea).
- El texto va entre comillas dobles `" "`.
- **Toda instrucción en Java termina con punto y coma `;`**. Si lo olvidas, el compilador te marca error. Piensa el `;` como el punto final de una oración: sin el, Java no sabe donde termina tu instrucción.
- Las llaves `{ }` delimitan bloques de código. Cada llave que abre debe cerrarse. Los editores modernos las cierran solos, pero en la terminal tienes que cuidarlas tu.

Errores comunes de esta unidad (leelos, te van a pasar)

1. **Nombre de archivo distinto al de la clase.** `public class HolaMundo` debe vivir en `HolaMundo.java`. Punto.
2. **Olvidar el punto y coma.** El compilador dirá algo como `';' expected`. Cuando veas ese mensaje, revisa la línea anterior: casi siempre falta el `;` ahí.
3. **`main` mal escrito.** Debe ser exactamente `public static void main(String[] args)`. Ni `Main`, ni sin `static`, ni `String args[]`... bueno, `String args[]` si funciona, pero no lo uses todavía.
4. **Compilar con `javac` y correr con `java` al revés.** Se compila el archivo (`javac HolaMundo.java`, con extensión) y se ejecuta la clase (`java HolaMundo`, sin extensión y sin `.class`).
5. **Acentos y caracteres raros en el código.** Java acepta texto con acentos dentro de las comillas, pero para evitar problemas de codificación en la terminal, en este libro usaremos texto sin acentos dentro del código. En tus programas reales puedes usarlos, pero configura tu editor en UTF-8.

Buenas prácticas desde el día uno

- **Nombres de clases en MayusculaCamelCase:** `HolaMundo`, `MiPrimerPrograma`, `CalculadoraSimple`. Así lo hacen los profesionales.
- **Un archivo, una clase publica.** Cada clase publica vive en su propio archivo `.java`.
- **Indentar tu código** (4 espacios por nivel). No es obligatorio, pero el código sin indentación es ilegible.

- **Compila seguido.** No escribas 200 líneas y luego compiles. Escribe poco, compila, prueba, sigue. Así los errores son pequeños y fáciles de encontrar.

Ejemplos paso a paso

Ejemplo 1: Tu primer Hola Mundo, paso a paso

Paso 1: Crea una carpeta para tus prácticas, por ejemplo `C:\java-practicas` en Windows o `~/java-practicas` en Mac/Linux.

Paso 2: Abre tu editor de texto (el Bloc de notas sirve, pero te recomiendo IntelliJ IDEA o VS Code) y escribe exactamente esto. Guárdalo como `HolaMundo.java` dentro de tu carpeta:

```
public class HolaMundo {  
    public static void main(String[] args) {  
        System.out.println("Hola, mundo!");  
        System.out.println("Estoy aprendiendo Java en Lucas EGA Academy");  
    }  
}
```

Paso 3: Abre la terminal (CMD o PowerShell en Windows, Terminal en Mac/Linux) y entra a tu carpeta:

```
cd C:\java-practicas
```

Paso 4: Compila:

```
javac HolaMundo.java
```

Si no sale ningún mensaje, todo bien: se creó el archivo `HolaMundo.class`. Si sale un error, léelo con calma: te dice el archivo, la línea y que esperaba encontrar.

Paso 5: Ejecuta:

```
java HolaMundo
```

Verás en pantalla:

```
Hola, mundo!  
Estoy aprendiendo Java en Lucas EGA Academy
```

Felicidades: acabas de compilar y ejecutar tu primer programa Java. Ya hiciste lo que el 90% de la gente que "quiere aprender a programar" nunca hace.

Ejemplo 2: Verificar tu instalación del JDK

Antes de programar, confirma que todo quedó bien instalado. En la terminal escribe:

```
javac --version
```

Deberías ver algo como `javac 17.0.9` o `javac 21.0.1`. Si dice "no se reconoce el comando", el JDK no quedó en el PATH del sistema. Solución: reinstala marcando la opción de agregar al PATH, o agrega manualmente la carpeta `bin` del JDK a las variables de entorno de tu sistema. En Windows esto se hace en: Configuración > Sistema > Acerca de > Configuración avanzada del sistema > Variables de entorno, y agregas algo como `C:\Program Files\Java\jdk-17\bin` al PATH.

También puedes verificar la versión de ejecución con:

```
java --version
```

Ejemplo 3: Varias líneas y comentarios

Los comentarios son notas para humanos que el compilador ignora. Hay dos tipos:

```
public class Comentarios {
    public static void main(String[] args) {
        // Este es un comentario de una línea: el compilador lo ignora

        /*
         * Este es un comentario de varias líneas.
         * Sirve para explicar bloques completos de código.
         */

        System.out.println("Los comentarios no se ejecutan,");
        System.out.println("pero hacen tu código entendible.");
    }
}
```

Buena práctica: comenta el "porqué", no el "qué". `// suma los precios` sobre una línea que claramente suma es ruido; `// se suma el IVA porque la ley lo exige desde 2024` explica algo que el código no dice.

Ejemplo 4: print vs println

`println` imprime y salta de línea. `print` imprime y se queda en la misma línea:

```
public class PrintVsPrintln {  
    public static void main(String[] args) {  
        System.out.print("Hola, ");  
        System.out.print("esto sigue en la misma linea. ");  
        System.out.println("Y esto termina la linea.");  
        System.out.println("Esta ya es otra linea.");  
    }  
}
```

Salida:

```
Hola, esto sigue en la misma linea. Y esto termina la linea.  
Esta ya es otra linea.
```

Ejemplo 5: Caracteres especiales en textos

Dentro de las comillas puedes usar secuencias de escape:

```
public class Escapes {  
    public static void main(String[] args) {  
        System.out.println("Primera linea\nSegunda linea");  
        System.out.println("Con\\ttabulacion");  
        System.out.println("Comillas dentro: \\\"Hola\\\"");  
    }  
}
```

- `\\n` = salto de línea
- `\\t` = tabulación
- `\\\"` = comilla doble dentro del texto

Práctica

Ejercicio 1. Escribe un programa `MisDatos.java` que imprima tu nombre, tu ciudad y tu meta con Java, cada dato en una línea distinta. Compilalo y ejecutalo.

Solución explicada: Es una variación directa del Hola Mundo. Lo importante es practicar el ciclo completo: escribir, guardar con el nombre exacto de la clase, compilar con `javac` y ejecutar con `java`.

```
public class MisDatos {  
    public static void main(String[] args) {  
        System.out.println("Me llamo Esteban");  
        System.out.println("Vivo en Ciudad de Mexico");  
        System.out.println("Mi meta: ser programador Java profesional");  
    }  
}
```

Si el compilador se queja, revisa: nombre del archivo igual al de la clase, punto y coma al final de cada instrucción, llaves balanceadas.

Ejercicio 2. Escribe `Dibujo.java` que imprima esta figura usando `println`:

```
  *  
 ***  
*****  
*****
```

Solución explicada: Cada línea es un `println` con espacios y asteriscos. Cuenta los espacios con cuidado: la punta lleva 3 espacios, luego 2, luego 1, luego 0. Este ejercicio entrena tu precisión, que en programación lo es todo.

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

Ejercicio 3. Crea `Presentacion.java` que use `print` (sin ln) para armar una frase completa en una sola línea de salida, y luego `println` para cerrarla. Por ejemplo: "Java es" + " mi nuevo lenguaje" + " favorito."

Solución explicada: La clave es entender que `print` no salta de línea, así que varias llamadas seguidas se concatenan en pantalla.

```
public class Presentacion {
    public static void main(String[] args) {
        System.out.print("Java es");
        System.out.print(" mi nuevo lenguaje");
        System.out.println(" favorito.");
        System.out.println("Y voy a dominarlo.");
    }
}
```

Ejercicio 4. Escribe `Tabla.java` que imprima una pequena tabla usando `\t` (tabulación):

Nombre	Edad	Ciudad
Ana	25	Lima
Luis	30	Bogota

Solución explicada: El `\t` alinea en columnas aproximadas. No es perfecto (depende del largo de cada palabra), pero para salidas simples de consola funciona muy bien.

```
public class Tabla {
    public static void main(String[] args) {
        System.out.println("Nombre\tEdad\tCiudad");
        System.out.println("Ana\t25\tLima");
        System.out.println("Luis\t30\tBogota");
    }
}
```

Ejercicio 5. A proposito, escribe un programa con TRES errores: un punto y coma faltante, una llave sin cerrar y el método `main` escrito como `Main`. Compilalo, lee los mensajes de error, y luego corrígelos uno por uno hasta que corra. Anota que decía cada mensaje.

Solución explicada: Este es el ejercicio más valioso de la unidad. Los mensajes del compilador al principio asustan, pero son instrucciones precisas. Por ejemplo, `';' expected` significa "aquí falta un punto y coma" (casi siempre en la línea anterior a la que marca). `{ expected` o `reached end of file while parsing` apuntan a llaves sin cerrar. Y si compila pero al correr dice `Main method not found`, ya sabes que el método está mal escrito. Entrenar esto ahora te ahorra horas de frustración después.

Resumen de la unidad

- Java nació en 1995, su lema es "escribe una vez, corre en todas partes" y corre sobre la JVM.
- Para programar necesitas el **JDK** (incluye compilador ``javac``); la JVM ejecuta, el JRE solo sirve para correr programas.
- El ciclo es: escribir ``java`` -> compilar con ``javac`` -> ejecutar con ``java``.
- Todo el código vive dentro de una **clase** cuyo nombre coincide con el archivo.
- El **método main** (``public static void main(String[] args)``) es el punto de arranque.
- ``System.out.println`` imprime con salto de línea; ``print`` sin salto; toda instrucción termina en ``;`.
- Errores típicos: nombre de archivo mal, falta el ``;`, ``main`` mal escrito, confundir ``javac`` con ``java``.

Instalación del JDK paso a paso (según tu sistema)

Windows:

1. Ve a la página oficial de Oracle (oracle.com/java) o a adoptium.net (Temurin, 100% gratis y open source).
2. Descarga el instalador del JDK 17 o 21 para Windows x64 (archivo ``msi`` o ``exe``).
3. Ejecuta el instalador y acepta todo por defecto. Importante: en la pantalla de opciones, verifica que este marcada la opción "Add to PATH" o "Set JAVA_HOME" si aparece.
4. Abre una terminal nueva (CMD o PowerShell) y escribe ``javac --version``. Si muestra la versión, listo.

Mac:

1. Descarga el ``dmg`` del JDK para macOS (versión ARM64 si tu Mac tiene chip M1/M2/M3, x64 si es Intel).
2. Instalalo como cualquier programa.
3. Abre la Terminal y verifica con ``javac --version``. En Mac el PATH suele configurarse solo.

Linux (Ubuntu/Debian):

1. Abre la terminal y escribe: ``sudo apt update && sudo apt install openjdk-17-jdk``
2. Verifica con ``javac --version``.

Si algo falla, el 99% de las veces es el PATH. El PATH es la lista de carpetas donde tu sistema busca programas cuando escribes un comando. Si ``javac`` no está en el PATH, la terminal no lo encuentra aunque este instalado. La solución es agregar la carpeta ``bin`` del JDK al PATH, como explique en el Ejemplo 2.

Versiones de Java: cual elegir

Java saca versiones nuevas cada 6 meses, pero no todas son iguales. Las versiones **LTS (Long Term Support)** tienen soporte por años y son las que usan las empresas: Java 8, 11, 17 y 21. Para aprender, usa **17 o 21**: son modernas, estables y todo lo que aprendas aquí funciona en ellas. Evita tutoriales viejos que usan Java 8 con herramientas obsoletas; la sintaxis base es la misma, pero te conviene aprender con lo actual.

Tu primer IDE: IntelliJ IDEA Community

Puedes programar en el Bloc de notas, pero un **IDE (Entorno de Desarrollo Integrado)** te ayuda con autocompletado, detección de errores en tiempo real y ejecución con un clic. **IntelliJ IDEA Community** es gratis y es el estandar de la industria Java.

Instalación rápida:

1. Descargalo de jetbrains.com/idea (la edición Community).
2. Instalalo con las opciones por defecto.
3. Al abrirlo, crea un "New Project", selecciona Java, verifica que detecte tu JDK instalado, y crea tu primer archivo.
4. Escribe tu ``HolaMundo``, dale clic derecho y "Run". Verás la salida en la consola integrada, sin tocar la terminal.

Alternativa ligera: **Visual Studio Code** con la extensión "Extension Pack for Java" de Microsoft. Es gratis, rápido y muy popular.

Mi recomendación: empieza con la terminal para entender qué pasa por debajo (compilar con ``javac``, correr con ``java``), y en la Unidad 2 pasate a IntelliJ para ir más rápido. Entender la terminal te hace mejor programador; usar el IDE te hace más rápido. Necesitas ambas cosas.

Más allá del Hola Mundo: que sigue

Ya sabes compilar y correr. En la Unidad 2 vas a guardar datos en variables, hacer calculos y pedirle información al usuario. Tu programa va a dejar de ser un cartel estático y va a empezar a interactuar. Pero antes, practica lo de hoy hasta que el ciclo escribir-compilar-ejecutar te salga sin pensar. Esa fluidez es la base de todo lo demás.

Mini glosario de la unidad

- **Lenguaje de programación:** idioma formal para darle instrucciones a una computadora.
- **Compilador:** programa que traduce tu código fuente a un formato ejecutable (en Java: ``javac``, produce bytecode).
- **Bytecode:** código intermedio que genera ``javac``; lo ejecuta la JVM.
- **JVM:** máquina virtual que ejecuta programas Java en cualquier sistema.
- **JDK:** kit de desarrollo (compilador + herramientas). Lo que instalas para programar.
- **Clase:** contenedor principal del código Java; todo vive dentro de una.
- **Método:** bloque de código con nombre que hace una tarea (como ``main`` o ``println``).
- **Consola/terminal:** ventana de texto donde compilas, ejecutas y ves resultados.
- **Sintaxis:** las reglas de escritura del lenguaje (donde van los ``;`, las llaves, etc.).

Práctica extra

Ejercicio 6. Investiga y escribe en un comentario al inicio de tu programa (o en un archivo de texto) las respuestas a: quien creó Java, en que año, y que empresa es dueña hoy. Luego imprime esas respuestas con ``println``.

**Solución:* Java fue creado por James Gosling y su equipo en Sun Microsystems, lanzado en 1995. Sun fue comprada por Oracle en 2010, así que hoy Java es de Oracle. Este dato sale en entrevistas de trabajo más seguido de lo que crees.

Ejercicio 7. Escribe ``Errores.java`` que imprima una lista de los 5 errores más comunes al compilar (los de esta unidad), cada uno con su mensaje típico del compilador.

Ejercicio 8 (reto). Sin mirar el ejemplo, escribe de memoria el programa Hola Mundo completo: clase, main, println, llaves y puntos y coma. Compilalo. Si compila a la primera, ya tienes la estructura base grabada. Si no, repite hasta lograrlo. Los profesionales escriben ``public static void main`` sin pensar; tu también llegarás ahí.

La historia completa: de Oak a Java

Java no nació para hacer páginas web. En 1991, un equipo de Sun Microsystems liderado por James Gosling (el "Green Team") trabajaba en un lenguaje para programar dispositivos electrónicos: televisiones interactivas, controles remotos, decodificadores de cable. El lenguaje se llamó primero **Oak** (roble, por un árbol que Gosling veía desde su ventana). El proyecto de TV interactiva fracasó comercialmente, pero el lenguaje era bueno.

En 1994-1995 explotó la web, y el equipo se dio cuenta de que Oak era perfecto para otro problema: programas pequeños que se descargaban y corrían dentro del navegador, sin importar si el usuario tenía Windows o Mac. Lo renombraron **Java** (se dice que por el café de la isla de Java, Indonesia, que tomaban en la oficina; por eso el logo es una taza de café humeante) y en 1995 lo lanzaron gratis. Netscape, el navegador dominante de la época, lo integró, y el resto es historia.

Dato curioso para entrevistas: los **applets** de Java (programas que corrían en el navegador) fueron la primera gran sensación de Java en los 90. Hoy están muertos, pero Java sobrevivió porque las empresas descubrieron que era perfecto para sistemas grandes y serios: en 1999 salió J2EE (Java empresarial) y los bancos y gobiernos lo adoptaron masivamente.

La JVM por dentro (sin complicarte la vida)

Ya sabes que la JVM ejecuta el bytecode. Pero que hace exactamente cuando corres ``java HolaMundo``? Tres cosas, en orden:

1. **Class Loader (cargador de clases):** busca tu archivo ``HolaMundo.class`` y lo carga en memoria. También carga las clases del sistema que uses (como ``System`` y ``String``).
2. **Verificador de bytecode:** revisa que el bytecode sea válido y seguro: que no intente acceder a memoria prohibida, que los tipos cuadren, que no haga trucos raros. Esta es una de las razones por las que Java es seguro: el código se inspecciona antes de correr.
3. **Ejecución:** la JVM interpreta el bytecode y, para las partes que se repiten mucho, usa el **compilador JIT (Just-In-Time)** para traducirlas a código máquina nativo superveloz en tiempo de ejecución. Por eso Java, aunque no es tan rápido como C++, es muchísimo más rápido de lo que la gente cree: el JIT optimiza mientras el programa corre.

No necesitas memorizar esto para programar, pero entenderlo te da una ventaja: cuando alguien diga "Java es lento", tu sabrás que el JIT y las JVM modernas lo hacen rapidísimo, y que por eso los sistemas que manejan millones de transacciones usan Java.

Las ediciones de Java: SE, EE y ME

- **Java SE (Standard Edition):** la base. Todo lo de este libro es Java SE: el lenguaje, las colecciones, archivos, hilos, JDBC. Es lo que instalas con el JDK.
- **Java EE / Jakarta EE (Enterprise Edition):** construida sobre SE, para aplicaciones empresariales grandes: servidores, transacciones, servicios web. Si algún día trabajas en un banco, usaras esto (Spring Boot, que es lo más pedido en empleos, vive en este mundo).
- **Java ME (Micro Edition):** para dispositivos pequeños y embebidos. Casi no se usa hoy, pero existió.

Tu ruta de aprendizaje: domina SE con este libro, luego aprende Spring Boot, y tendrás el perfil que más contratan las empresas.

El lema en acción: escribe una vez, corre en todas partes

Imagina que escribes `HolaMundo.java` en tu Windows, lo compilas y obtienes `HolaMundo.class`. Ese archivo `.class` lo puedes copiar a una Mac, a un servidor Linux o a una Raspberry Pi, y en todos corre con `java HolaMundo`, sin recompilar. El bytecode es el mismo; lo único que cambia es la JVM de cada sistema, que sabe traducirlo a su máquina.

Comparalo con C++: ahí compilas a código máquina específico de Windows, y para Mac tienes que recompilar todo. Esa es la magia de Java y la razón por la que las empresas lo aman: desarrollan una vez y despliegan donde quieran.

Java vs otros lenguajes (panorama honesto)

- **vs Python:** Python es más fácil de empezar y más rápido de escribir; Java es más rápido al correr y más estricto (atrapa más errores antes). Python domina en ciencia de datos e IA; Java domina en empresas, bancos y Android.
- **vs JavaScript:** JavaScript vive en el navegador y en servidores con Node.js; Java vive en sistemas grandes de backend. Son primos lejanos: el nombre parecido fue marketing de los 90, nada más.
- **vs C#:** son gemelos separados al nacer. C# (de Microsoft) se inspiró muchísimo en Java. Si sabes Java, aprender C# te toma semanas.
- **vs C++:** C++ te da control total de la memoria (y todos sus peligros); Java maneja la memoria por ti con el **recolector de basura (Garbage Collector)**. Menos poder, muchísima menos posibilidad de romper todo.

No hay "mejor lenguaje": hay herramientas para trabajos. Java es la herramienta de los sistemas grandes que no se pueden caer. Aprenderlo te da una base que te sirve para cualquier otro lenguaje orientado a objetos.

Anatomía extendida: que más hay en un programa real

Tu Hola Mundo es minimalista. Un programa Java típico también usa:

- **La palabra ``package``:** organiza clases en paquetes (carpetas logicas). En proyectos reales, la primera línea suele ser algo como ``package com.miempresa.proyecto;``. En este libro no usaremos paquetes hasta los proyectos finales, para no complicar.
- **La palabra ``import``:** trae clases de otros paquetes para usarlas. Por ejemplo, ``import java.util.Scanner;`` te permite usar ``Scanner`` escribiendo solo su nombre corto. La verás desde la Unidad 2.
- **Variables y métodos dentro de la clase:** tu clase crecerá con datos y comportamientos. Eso es la Unidad 6.

Por ahora, tu plantilla mental es: archivo = clase = programa. Todo lo demás se construye encima.

Práctica avanzada de la unidad

Ejercicio 9. Modifica tu ``HolaMundo.java`` para que imprima un marco decorativo alrededor de un mensaje, así:

```
=====
=  Hola, Java!  =
=====
```

Solución explicada: Son 3 ``println``. El truco está en contar caracteres para que el marco cierre parejo. Si una línea queda más corta, agrega espacios antes del ``=``. Este tipo de precisión visual es la misma que usaras al formatear reportes y tablas en programas reales.

```
public class HolaMundo {
    public static void main(String[] args) {
        System.out.println("=====");
        System.out.println("=   Hola, Java!   =");
        System.out.println("=====");
    }
}
```

Ejercicio 10. Escribe `Version.java` que imprima la versión de Java que tienes instalada usando `System.getProperty("java.version")`. Investiga que hace `System.getProperty`.

Solución explicada: `System.getProperty` lee propiedades del sistema donde corre la JVM. Es útil para que tu programa se adapte al entorno.

```
public class Version {
    public static void main(String[] args) {
        System.out.println("Version   de   Java:   "   +
System.getProperty("java.version"));
        System.out.println("Sistema operativo: " + System.getProperty("os.name"));
    }
}
```

El `+` aquí une (concatena) texto con el valor. Lo verás a fondo en la Unidad 2.

Ejercicio 11 (investigación). Abre la carpeta donde compilaste y mira los archivos: verás `HolaMundo.java` y `HolaMundo.class`. Abre el `class` con el Bloc de notas: verás basura ilegible (es bytecode binario). Ahora borra el `class` e intenta correr `java HolaMundo`: que error sale? Vuelve a compilar y corre de nuevo. Este ejercicio te graba la diferencia entre fuente y compilado.

Respuesta esperada: Al borrar el `class`, `java HolaMundo` dice algo como `Error: Could not find or load main class HolaMundo`. La JVM necesita el bytecode compilado; el `java` solo le sirve al compilador.

Catálogo de mensajes del compilador (guardalo, lo usaras a diario)

Cuando `javac` encuentra un problema, te dice el archivo, la línea y una descripción. Aprende a leerlos:

1. `';' expected`

```
HolaMundo.java:4: error: ';' expected
    System.out.println("Hola")
                        ^
```

Significa: falta el punto y coma. Casi siempre el problema está en la línea marcada o en la anterior. Solución: agrega `;`.

2. `cannot find symbol`

```
HolaMundo.java:4: error: cannot find symbol
    System.ou.println("Hola");
           ^
symbol:   method ou
```

Significa: escribiste mal un nombre (`ou` en vez de `out`). Java es sensible a mayúsculas: `system`, `System` y `SYSTEM` son tres cosas distintas. Revisa letra por letra.

3. `class HolaMundo is public, should be declared in a file named HolaMundo.java`

Significa: el archivo se llama distinto a la clase pública. Renombra el archivo (o la clase) para que coincidan exactamente, respetando mayúsculas.

4. `reached end of file while parsing` Significa: te falta cerrar una llave `}`. Cuenta tus llaves: cada `{` necesita su `}`. Los IDEs te ayudan con esto; en terminal, indenta bien para verlo.

5. `illegal start of expression` Suele ser una llave de más, un paréntesis sin cerrar, o código fuera de lugar (por ejemplo, una instrucción suelta fuera de cualquier método). Revisa la zona marcada.

6. Al correr: `Error: Could not find or load main class HolaMundo` El programa compiló bien (o crees que sí), pero la JVM no encuentra la clase. Causas típicas: no compilaste (no existe el `.class`), corres desde la carpeta equivocada, o el método `main` está mal escrito y `javac` ni siquiera lo marco como error (porque sintacticamente tu `Main` es un método válido cualquiera, solo que la JVM no lo reconoce como punto de arranque).

7. Al correr: `Main method not found in class HolaMundo` Variante del anterior: el `.class` existe pero no tiene el `main` exacto. Revisa `public static void main(String[] args)` letra por letra.

Regla de oro para depurar: **lee el primer error, ignora los demás**. Un solo error al inicio (como una llave faltante) puede generar 20 errores en cascada. Corrige el primero, recompila, y mágicamente desaparecen los demás.

Java en el mundo laboral: por qué vale la pena

Datos que te motivan a seguir:

- Java lleva más de 25 años entre los 3 lenguajes más demandados en ofertas de empleo de todo el mundo. No es moda: es infraestructura.
- Los puestos típicos: desarrollador backend Java, desarrollador Android, ingeniero de software empresarial, arquitecto de soluciones.
- Las empresas que más contratan Java: bancos, aseguradoras, gobierno, telecomunicaciones, retail grande, y prácticamente toda empresa con sistemas "que no se pueden caer".
- El ecosistema profesional gira alrededor de **Spring Boot** (framework para crear aplicaciones y APIs rapidísimo), **Maven/Gradle** (manejo de dependencias, lo verás en la Unidad 10) y bases de datos con **JDBC/JPA** (Unidad 10 también).

Tu ruta después de este libro: 1) domina este contenido, 2) aprende Spring Boot, 3) arma 2-3 proyectos para tu portafolio, 4) aplica a empleos junior o freelance. En lucasegaacademy.com encontraras más cursos para seguir esa ruta (el de Android usa Java directamente).

La mentalidad del programador Java

Antes de cerrar la unidad, tres ideas que te van a acompañar todo el libro:

1. **El compilador es tu amigo, no tu enemigo.** Cada error que `javac` te marca es un error que NO llegó al usuario. Los programadores buenos no son los que no cometen errores; son los que leen los mensajes y los corrigen rápido.
2. **Explícito es mejor que mágico.** Java te pide declarar tipos, cerrar llaves, poner puntos y coma. Al principio parece burocracia; con el tiempo descubres que esa claridad hace que programas de millones de líneas sigan siendo entendibles por equipos de cientos de personas.
3. **Todo es un objeto (casi).** Esta frase te va a perseguir amablemente durante todo el libro. Cuando algo no te cuadre, pregúntate: "de qué objeto es este método? de qué clase viene este dato?".

Resumen extendido y checklist de la unidad

Antes de pasar a la Unidad 2, verifica que puedes hacer esto sin mirar:

- [] Explicar con tus palabras que es la JVM, el JDK y el bytecode.

- [] Instalar el JDK y verificarlo con ``javac --version``.
- [] Escribir, compilar y ejecutar un Hola Mundo desde la terminal.
- [] Escribir el método ``main`` de memoria.
- [] Diferenciar ``print`` de ``println`` y usar ``\n`` y ``\t``.
- [] Leer un error de ``javac`` y saber donde buscar el problema.
- [] Explicar el lema "escribe una vez, corre en todas partes".

Si marcaste todo, vas perfecto. Si algo falla, repasa esa sección antes de seguir. La Unidad 2 construye directamente sobre esto: variables, calculos y programas que conversan contigo.

Los argumentos del main: `String[] args`

Te acuerdas del ``(String[] args)`` del método main? Hasta ahora lo hemos ignorado, pero es útil: son los **argumentos de la línea de comandos**, palabras que le pasas a tu programa al ejecutarlo.

```
public class Saludo {
    public static void main(String[] args) {
        System.out.println("Numero de argumentos: " + args.length);
        for (int i = 0; i < args.length; i++) {
            System.out.println("Argumento " + i + ": " + args[i]);
        }
    }
}
```

Compila y corre así:

```
javac Saludo.java
java Saludo Ana Luis Maria
```

Salida:

```
Numero de argumentos: 3
Argumento 0: Ana
Argumento 1: Luis
Argumento 2: Maria
```

Cada palabra separada por espacio llega como un elemento del arreglo ``args``. (No te preocupes por el ``for`` y los arreglos: son las Unidades 4 y 5. Por ahora solo entiende la idea:

tu programa puede recibir datos al arrancar.) Los programas profesionales usan argumentos para cosas como `java MiApp --modo-rapido --archivo=datos.txt`.

El CLASSPATH: donde busca Java tus clases

Cuando corres `java HolaMundo`, la JVM busca `HolaMundo.class` en el **classpath**, que por defecto es la carpeta actual (por eso tienes que estar parado en la carpeta correcta). Si tu clase está en otra carpeta, puedes indicarlo:

```
java -cp C:\java-practicas HolaMundo
```

El `-cp` (classpath) le dice a la JVM donde buscar. En proyectos con decenas de archivos y librerías externas (archivos `.jar`), el classpath se vuelve importante. Los IDEs y Maven lo manejan por ti, pero saber que existe te salva cuando algo "no se encuentra" y si existe.

Java sin IDE vs con IDE: tabla comparativa

Tarea	En terminal	En IntelliJ IDEA
Compilar	<code>javac Archivo.java</code>	Automatico al guardar
Ejecutar	<code>java Archivo</code>	Clic derecho > Run
Ver errores	Mensajes en terminal	Subrayado rojo en vivo
Organizar proyecto	Carpetas manuales	Estructura automatica
Depurar	<code>System.out.println</code> por todos lados	Debugger con puntos de interrupción

Ambos caminos llegan al mismo lugar. La terminal te enseña; el IDE te acelera.

Lo que NO debes hacer en esta etapa

- **No instales 5 versiones del JDK "por si acaso".** Una sola versión LTS (17 o 21) es suficiente. Varias versiones solo generan confusión con el PATH.
- **No programes en Word ni en editores con formato.** Usa texto plano: Bloc de notas, VS Code, IntelliJ. Word mete caracteres invisibles que rompen la compilación.
- **No te saltes la terminal.** Aunque uses IDE desde el día uno, dedica una tarde a compilar y correr desde la terminal. Entender que pasa por debajo te hace depurar mejor toda la vida.
- **No memorices sin practicar.** El `main` se aprende escribiendolo 20 veces, no leyendolo 20 veces.

Práctica final de la unidad

Ejercicio 12. Escribe `Eco.java` que reciba argumentos por línea de comandos y los imprima todos en una sola línea separados por espacio (pista: usa `System.out.print` dentro del ciclo del ejemplo `Saludo`, y un `println` vacío al final).

Solución:

```
public class Eco {  
    public static void main(String[] args) {  
        for (int i = 0; i < args.length; i++) {  
            System.out.print(args[i] + " ");  
        }  
        System.out.println();  
    }  
}
```

Al correr `java Eco Hola mundo Java` imprime `Hola mundo Java`. Es el mismo principio del comando `echo` de la terminal: acabas de reescribir una herramienta del sistema en Java.

Ejercicio 13 (reto de precisión). Escribe el Hola Mundo completo de memoria 5 veces seguidas, compilando cada vez. Si las 5 compilan a la primera, la sintaxis base ya es tuya. Este ejercicio parece tonto; no lo es. La fluidez con la estructura base libera tu mente para pensar en la lógica, que es lo difícil de verdad.

Ejercicio 14 (investigación). Busca en internet "TIOBE Index" y "lenguajes más usados 2026". Anota en que posición esta Java y que lenguajes lo acompañan en el top 5. Luego busca "sueldo desarrollador Java junior" en tu país. Guarda esos números: en los días difíciles, te van a recordar por qué empezaste.

Laboratorio guiado: tu entorno, paso a paso y sin miedo

Vamos a dejar tu entorno a prueba de balas. Sigue estos pasos en orden y verifica cada uno antes de continuar.

Paso 1: verifica Java. Abre la terminal y escribe `java version`. Debes ver algo como `openjdk 17.0.x`. Si dice "comando no encontrado", Java no está instalado o no está en el PATH: instala el JDK (Adoptium Temurin es gratis y fácil) y reinicia la terminal.

Paso 2: verifica el compilador. Escribe `javac version`. Si `java` funciona pero `javac` no, instalaste el JRE en vez del JDK: necesitas el JDK completo.

Paso 3: tu carpeta de trabajo. Crea una carpeta ``java-curso`` y dentro una subcarpeta por unidad (``unidad01``, ``unidad02``...). Organizar desde el día uno te ahorra el caos de 50 archivos sueltos.

Paso 4: el ritual completo. En ``unidad01``, crea ``HolaMundo.java``, compilalo con ``javac HolaMundo.java``, verifica que aparecio ``HolaMundo.class``, y ejecútalo con ``java HolaMundo``. Si ves el saludo, tu entorno está perfecto.

Errores de instalación más comunes:

- `"`javac` no se reconoce"`: JDK no instalado o PATH sin configurar. En Windows, agrega ``C:\Program Files\Java\jdk-17\bin`` al PATH del sistema.
- `"Error: could not find or load main class"`: estas en la carpeta equivocada, escribiste mal el nombre de la clase, o agregaste ``.class`` al comando ``java``.
- Todo compila pero no pasa nada: revisa que tu método sea exactamente ``public static void main(String[] args)``; una letra mal y Java no lo reconoce como punto de entrada.

Tu primer habito profesional: compila después de cada cambio pequeño, no después de escribir 200 líneas. Los errores se encuentran más fácil cuando sabes que las últimas 5 líneas son las sospechosas.

Preguntas frecuentes de la unidad

P: ``java`` y ``javac``: cual es cual? Siempre los confundo. R: ``javac`` = compilador (traduce tu ``.java`` a ``.class``). ``java`` = ejecutor (corre el ``.class`` en la JVM). Orden: primero ``javac``, luego ``java``. Mnemotecnia: la "c" de ``javac`` es de "compilar".

P: Escribi el código bien pero ``java HolaMundo`` dice "could not find or load main class". R: Las causas top: 1) no compilaste (falta el ``.class``), 2) estas en otra carpeta, 3) escribiste ``java HolaMundo.class`` (sin extensión), 4) el nombre de la clase no coincide con el archivo. Revisa en ese orden.

P: Puedo programar Java en el bloc de notas? R: Puedes, y de hecho hacerlo al inicio te ensena a compilar a mano. Pero para proyectos reales usa un IDE (IntelliJ): autocompletado, errores en vivo y depurador te ahorran horas.

P: Que versión de Java aprendo? Veo 8, 11, 17, 21... R: Este libro usa Java 17 (LTS: soporte largo). Lo que aprendas aquí funciona en todas. Evita tutoriales viejos de Java 8 para temas nuevos, pero no te asustes si tu trabajo usa 11: es casi lo mismo en lo esencial.

P: Cuanto tarda en aprenderse Java? R: Con 1 hora diaria: en 3-4 semanas lees y escribes programas basicos; en 3 meses haces proyectos completos; en 1 año eres empleable como junior. La constancia importa más que las horas maratonicas.

UNIDAD 2: Datos, operaciones y conversación

Introducción

En la Unidad 1 tus programas eran carteles: mostraban texto fijo y se acabo. En esta unidad les vas a dar memoria y oídos. **Memoria** con las variables, que guardan datos; **calculadora** con los operadores, que transforman esos datos; y **oídos** con la clase Scanner, que escucha lo que el usuario escribe. Al terminar, tus programas van a pedir datos, hacer cuentas y responder. Eso ya es un programa de verdad.

Piensa en un programa como una cocina: las variables son los recipientes donde guardas ingredientes (datos), los operadores son los utensilios que los transforman (cuchillos, batidora), y el Scanner es el mesero que trae los pedidos del cliente (el usuario). Sin recipientes no hay donde guardar, sin utensilios no hay transformación, y sin mesero la cocina no sabe que cocinar.

Esta unidad es la más importante para tu base: si las variables y los operadores no te quedan clarísimos, todo lo demás (ciclos, objetos, colecciones) se tambalea. Tomate tu tiempo aquí.

Explicación detallada

Variables: cajas con etiqueta y tipo

Una **variable** es un espacio de memoria con un nombre, donde guardas un dato. En Java, toda variable se declara con tres cosas: **tipo**, **nombre** y (opcionalmente) **valor inicial**:

```
int edad = 30;
```

- `int` es el tipo: dice que esta variable solo guarda números enteros.
- `edad` es el nombre: como la vas a llamar en tu código.
- `30` es el valor inicial: lo que guarda al nacer.
- El `;` cierra la instrucción, como siempre.

Analogía: una variable es una caja de zapatos con etiqueta. El tipo dice que cabe en la caja (solo zapatos, no bicicletas), el nombre es la etiqueta escrita con marcador, y el valor es lo que metes dentro. Si intentas meter una bicicleta en una caja de zapatos (guardar texto en un `int`), Java no te deja: el compilador protesta. Esa rigidez es tu protección.

Puedes declarar sin dar valor inicial:

```
int edad;           // declarada, pero vacia (solo vale para variables locales con
cuidado)
edad = 30;          // asignacion posterior
```

Pero ojo: en Java **no puedes usar una variable local sin haberle dado valor antes**. Esto no compila:

```
int edad;
System.out.println(edad); // ERROR: variable edad might not have been initialized
```

El compilador te cuida de usar basura de memoria. Acostumbrate a inicializar siempre.

Los 8 tipos primitivos (memorizalos)

Java tiene 8 tipos **primitivos**: los bloques basicos de datos, rápidos y simples. Todo lo demás se construye con ellos.

Números enteros (4 tamanos):

| Tipo | Tamano | Rango | Uso típico | |---|---|---|---| | `byte` | 8 bits | -128 a 127 | Datos muy pequenos, archivos binarios | | `short` | 16 bits | -32,768 a 32,767 | Casi no se usa | | `int` | 32 bits | -2,147,483,648 a 2,147,483,647 | El de todos los días | | `long` | 64 bits | Enorme (+9 trillones) | Cantidades gigantes, IDs, milisegundos |

`int` es tu caballo de batalla: edades, contadores, cantidades normales. Usa `long` cuando el número puede pasar de 2 mil millones (por ejemplo, la población mundial cabe en `int`, pero los milisegundos desde 1970 ya necesitan `long`). Los literales `long` llevan `L` al final: `long x = 9000000000L;` (sin la `L`, el compilador cree que es `int` y se queja).

Números decimales (2 tamanos):

| Tipo | Tamano | Precision | Uso típico | |---|---|---|---| | `float` | 32 bits | ~7 digitos | Graficos, casi no se usa | | `double` | 64 bits | ~15 digitos | El de todos los días |

`double` es el estandar: `double precio = 99.50;`. Los literales decimales son `double` por defecto; si quieres `float`, agrega `F`: `float f = 3.14F;`.

Advertencia honesta: `float` y `double` son **aproximados** en binario. `0.1 + 0.2` en Java da `0.30000000000000004`, no `0.3`. Para dinero real, los profesionales usan `BigDecimal`, no `double`. En este libro usaremos `double` para practicar, pero ya sabes el secreto.

Otros dos:

- ``char``: un solo carácter, entre comillas simples: ``char` letra = 'A';``. Ojo: comillas simples, no dobles. ``A`` es char, ``"A"`` es String.
- ``boolean``: solo dos valores: ``true`` o ``false``. Es el tipo de las decisiones (Unidad 3) y los ciclos (Unidad 4). Sin ``boolean`` no hay lógica.

String: el tipo para texto (no es primitivo)

``String`` guarda texto entre comillas dobles:

```
String nombre = "Esteban";  
String vacio = "";
```

``String`` no es primitivo: es una **clase** (un objeto). Por eso tiene métodos útiles que los primitivos no tienen:

```
String nombre = "Esteban";  
System.out.println(nombre.length());           // 7: cuantos caracteres  
System.out.println(nombre.toUpperCase());       // "ESTEBAN"  
System.out.println(nombre.charAt(0));           // 'E': caracter en la posicion 0
```

Y se puede **concatenar** (unir) con ``+``:

```
String saludo = "Hola, " + nombre + "!"; // "Hola, Esteban!"
```

Cuando mezclas texto con números, Java convierte el número a texto automáticamente:

```
int edad = 30;  
System.out.println("Tienes " + edad + " años"); // "Tienes 30 años"
```

Nombres de variables: reglas y buenas prácticas

Reglas (obligatorias):

- Empiezan con letra, ``_`` o ``$`` (evita ``$``, es para código generado).
- Después pueden llevar letras, dígitos, ``_``.
- No pueden ser palabras reservadas (``int``, ``class``, ``public``, etc.).
- Java distingue mayúsculas: ``edad``, ``Edad`` y ``EDAD`` son tres variables distintas.

Buenas prácticas (lo que separa profesionales de aficionados):

- Usa **camelCase** empezando en minúscula: ``edadUsuario``, ``precioTotal``, ``nombreCompleto``.
- Nombres descriptivos: ``e`` no dice nada; ``edadDelCliente`` si. El código se lee más veces de las que se escribe.
- Las **constantes** (valores que no cambian) van en MAYUSCULAS_CON_GUIONES y se declaran con ``final``: ``final double IVA = 0.16``; ``final`` es un candado: si intentas cambiarla después, el compilador te lo impide.

```
final double PI = 3.14159;
PI = 3.0; // ERROR: cannot assign a value to final variable PI
```

Operadores: las herramientas de transformación

Aritmeticos (los de la primaria, más dos):

Operador	Nombre	Ejemplo	Resultado
<code>`+`</code>	Suma	<code>`5 + 3`</code>	<code>`8`</code>
<code>`-`</code>	Resta	<code>`5 - 3`</code>	<code>`2`</code>
<code>`*`</code>	Multiplicacion	<code>`5 * 3`</code>	<code>`15`</code>
<code>`/`</code>	Division	<code>`7 / 2`</code>	<code>`3`</code> (si son enteros)
<code>`%`</code>	Módulo (residuo)	<code>`7 % 2`</code>	<code>`1`</code>

La **división entera** es la trampa clasica: ``7 / 2`` da ``3``, no ``3.5``, porque si ambos operandos son ``int``, el resultado es ``int`` y se trunca el decimal. Si quieres decimales, que al menos uno sea ``double``: ``7.0 / 2`` da ``3.5``. Este error aparece en exámenes y en entrevistas. Grabatelo.

El **módulo** ``%`` da el residuo de la división: ``7 % 2 = 1`` (7 entre 2 cabe 3 veces, sobra 1). Sus usos son oro puro:

- Saber si un número es par: ``n % 2 == 0``.
- Obtener el último dígito: ``n % 10``.
- Ciclos que se repiten cada N: ``i % 7`` para días de la semana.

Incremento y decremento (atajos que usaras miles de veces):

```
int i = 5;
i++; // i ahora vale 6 (equivale a i = i + 1)
i--; // i ahora vale 5
```

`++` suma 1, `--` resta 1. Hay versión prefija (`++i`) y postfija (`i++`): la diferencia importa solo cuando las usas dentro de otra expresión, cosa que evitaremos por ahora. En ciclos (Unidad 4) verás `i++` hasta en la sopa.

Asignación compuesta (atajos elegantes):

```
int x = 10;
x += 5;    // x = 15 (equivale a x = x + 5)
x -= 3;    // x = 12
x *= 2;    // x = 24
x /= 4;    // x = 6
x %= 4;    // x = 2
```

Relacionales (comparan y devuelven `boolean`):

Operador	Significado	Ejemplo
`==`	Igual a	`5 == 5` -> `true` `!=` Distinto de
`!=`	Distinto de	`5 != 3` -> `true` `>` Mayor que
`>`	Mayor que	`5 > 3` -> `true` `<` Menor que
`<`	Menor que	`5 < 3` -> `false` `>=` Mayor o igual
`>=`	Mayor o igual	`5 >= 5` -> `true` `<=` Menor o igual
`<=`	Menor o igual	`5 <= 4` -> `false`

La confusión mortal: `=` asigna, `==` compara. `if (x = 5)` asigna 5 a x (y en Java ni siquiera compila en un `if` porque 5 no es boolean, pero en otros lenguajes es un bug silencioso). `if (x == 5)` compara. Cuando compares Strings, **NUNCA uses `==`** (compara referencias, no contenido): usa `.equals()`. Lo repetiremos en la Unidad 3 hasta que duela.

Lógicos (combinan booleanos):

Operador	Nombre	Ejemplo	Resultado
`&&`	AND (y)	`true && false`	`false`
`  `	OR (o)	`true    false`	`true`
`!`	NOT (no)	`!true`	`false`

Tablas de verdad (memoriza la lógica):

- AND (`&&`): solo `true` si AMBOS son `true`. "Para entrar a la fiesta necesitas invitación Y ser mayor de edad."
- OR (`||`): `true` si AL MENOS UNO es `true`. "Puedes pagar con tarjeta O efectivo."
- NOT (`!`): invierte. `!true` es `false`.

Java usa **evaluación de cortocircuito**: en `a && b`, si `a` es `false`, ni siquiera evalúa `b` (ya sabe que el resultado es `false`). Esto no solo es rápido: te protege. Ejemplo: `if (texto != null && texto.length() > 0)` es seguro porque si `texto` es `null`, la segunda parte jamás se ejecuta (y evita el famoso `NullPointerException`).

Precedencia (que se evalúa primero): como en matemáticas, `\*` y `/` antes que `+` y `-`. `2 + 3 \* 4` es `14`, no `20`. Cuando dudes, usa paréntesis: `(2 + 3) \* 4`. Los paréntesis siempre aclaran y nunca están de más.

Scanner: tu programa aprende a escuchar

Hasta ahora los datos estaban escritos en el código. Con `Scanner`, el usuario los escribe al correr el programa:

```
import java.util.Scanner;

public class SaludoUsuario {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.print("Como te llamas? ");
        String nombre = sc.nextLine();

        System.out.println("Hola, " + nombre + ". Bienvenido a Java.");

        sc.close();
    }
}
```

Por partes:

- `import java.util.Scanner;` - Va hasta arriba, antes de la clase. Le dice a Java: "voy a usar la clase Scanner que vive en el paquete `java.util`". Sin el import, tendrías que escribir `java.util.Scanner` completo cada vez.
- `Scanner sc = new Scanner(System.in);` - Creas un objeto Scanner conectado a `System.in` (el teclado). La palabra `new` crea objetos; la verás muchísimo desde la Unidad 6.
- `sc.nextLine()` - Lee una línea completa de texto que el usuario escriba (hasta que presione Enter) y la devuelve como `String`.
- `sc.close()` - Cierra el Scanner al terminar. Buena práctica: libera el recurso.

Métodos de lectura según el tipo de dato:

Método	Lee	Ejemplo de entrada	--- --- ---	`nextLine()`	Línea de texto completa	`Ana Maria`		`next()`	Una palabra (hasta el espacio)	`Ana`		`nextInt()`	Un entero	`25`		`nextDouble()`	Un decimal	`99.50`		`nextBoolean()`	true/false	`true`	
--------	-----	--------------------	-------------	--------------	-------------------------	-------------	--	----------	--------------------------------	-------	--	-------------	-----------	------	--	----------------	------------	---------	--	-----------------	------------	--------	--

La trampa del Scanner (le pasa a TODO el mundo): mezclar `nextInt()` con `nextLine()`:

```
System.out.print("Edad: ");
int edad = sc.nextInt();      // el usuario escribe 25 y presiona Enter
System.out.print("Nombre: ");
String nombre = sc.nextLine(); // AQUI ESTA EL PROBLEMA
```

Que pasa? `nextInt()` lee el `25` pero **deja el Enter (`\n`) en el buffer**. Luego `nextLine()` lee ese Enter pendiente y devuelve un texto vacío, sin dejar al usuario escribir. El programa "se salta" la pregunta del nombre.

Soluciones (elige una):

1. Después de `nextInt()`, agrega un `sc.nextLine();` extra que consuma el Enter pendiente.
2. Lee todo con `nextLine()` y convierte: `int edad = Integer.parseInt(sc.nextLine());`.

La opción 2 es más robusta y es la que usan muchos profesionales. `Integer.parseInt` convierte texto a entero; `Double.parseDouble` convierte texto a decimal.

Conversion de tipos (casting)

A veces necesitas convertir un tipo en otro:

Automatica (implicita): de un tipo pequeño a uno grande, Java lo hace solo:

```
int x = 100;
long y = x;      // automatico, sin peligro
double d = x;    // automatico
```

Manual (explícita, casting): de grande a pequeño, TU asumes el riesgo:

```
double d = 9.99;
int x = (int) d; // x vale 9: se trunca el decimal, se pierde .99
```

El casting con paréntesis `(int)` es como decirle a Java "se lo que hago". Usalo con cuidado: `long` a `int` puede desbordar si el número no cabe.

De String a número (super común con Scanner):

```
String texto = "42";
int n = Integer.parseInt(texto);      // 42
double d = Double.parseDouble("3.14"); // 3.14
```

De número a String: concatena con texto vacío o usa `String.valueOf`:

```
int n = 42;
String s = "" + n;           // "42"
String s2 = String.valueOf(n); // "42", mas elegante
```

Errores comunes de la unidad

1. **Division entera inesperada:** `promedio = (a + b) / 2;` con enteros trunca. Si `promedio` debe tener decimales, declara las variables como `double` o divide entre `2.0`.
2. `=` vs `==`: asignar cuando querias comparar.
3. **Comparar Strings con `==`:** usa `.equals()`. `"hola" == "hola"` a veces da `true` por optimización interna (string pool) y a veces `false`: comportamiento traicionero. Siempre `.equals()`.
4. **La trampa del Scanner** (`nextInt` + `nextLine`): ya la viste arriba.
5. **Variable sin inicializar:** usar una variable local antes de darle valor.
6. **`char` con comillas dobles:** `char c = "A";` no compila. `Char` = comillas simples.
7. **Desbordamiento de `int`:** `int x = 2147483647 + 1;` da `-2147483648` (vuelta completa, sin aviso). Para cantidades grandes usa `long`.

Buenas prácticas

- Inicializa tus variables al declararlas.
- Nombres descriptivos en camelCase; constantes en MAYUSCULAS con `final`.
- Un Scanner por programa, cierralo al final.
- Prefiere `nextLine()` + `parseInt/parseDouble` sobre `nextInt()` directo.
- Cuando una operación mezcla enteros y decimales, se explícito: usa `2.0` en vez de `2` si quieres decimales.

Ejemplos paso a paso

Ejemplo 1: Calculadora de promedio

```
import java.util.Scanner;

public class Promedio {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.print("Calificacion 1: ");
        double c1 = Double.parseDouble(sc.nextLine());
        System.out.print("Calificacion 2: ");
        double c2 = Double.parseDouble(sc.nextLine());
        System.out.print("Calificacion 3: ");
        double c3 = Double.parseDouble(sc.nextLine());

        double promedio = (c1 + c2 + c3) / 3.0;
        System.out.println("Tu promedio es: " + promedio);

        sc.close();
    }
}
```

Nota el `/ 3.0`: como las calificaciones ya son `double`, el resultado seria decimal de todos modos, pero el `3.0` deja clara la intención. Claridad > brevedad.

Ejemplo 2: Conversion de temperatura

```
import java.util.Scanner;

public class Temperatura {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.print("Grados Celsius: ");
        double c = Double.parseDouble(sc.nextLine());

        double f = c * 9.0 / 5.0 + 32;
        System.out.println(c + " C son " + f + " F");

        sc.close();
    }
}
```

Fórmula: $F = C * 9/5 + 32$. Ojo al orden: ``c * 9.0 / 5.0`` se evalúa de izquierda a derecha: primero multiplica, luego divide. Todo en ``double``, sin truncamientos.

Ejemplo 3: Par o impar con módulo

```
import java.util.Scanner;

public class ParImpar {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.print("Dame un numero: ");
        int n = Integer.parseInt(sc.nextLine());

        int residuo = n % 2;
        System.out.println("Residuo de dividir entre 2: " + residuo);
        System.out.println("Es par? " + (residuo == 0));

        sc.close();
    }
}
```

Los paréntesis en ``(residuo == 0)`` son necesarios: sin ellos, Java intentaría concatenar ``"Es par? " + residuo`` primero y luego comparar un String con 0. La precedencia importa.

Ejemplo 4: Ticket de compra

```

import java.util.Scanner;

public class Ticket {
    public static void main(String[] args) {
        final double IVA = 0.16;
        Scanner sc = new Scanner(System.in);

        System.out.print("Precio del producto: ");
        double precio = Double.parseDouble(sc.nextLine());
        System.out.print("Cantidad: ");
        int cantidad = Integer.parseInt(sc.nextLine());

        double subtotal = precio * cantidad;
        double impuesto = subtotal * IVA;
        double total = subtotal + impuesto;

        System.out.println("---- TICKET ----");
        System.out.println("Subtotal: $" + subtotal);
        System.out.println("IVA (16%): $" + impuesto);
        System.out.println("TOTAL: $" + total);

        sc.close();
    }
}

```

Este ejemplo junta todo: constante con `final`, entrada de datos, operaciones y salida formateada. Es un mini programa de negocio real.

Ejemplo 5: Intercambio de valores

```

public class Intercambio {
    public static void main(String[] args) {
        int a = 5;
        int b = 10;
        System.out.println("Antes: a=" + a + ", b=" + b);

        int temp = a;
        a = b;
        b = temp;

        System.out.println("Despues: a=" + a + ", b=" + b);
    }
}

```

Para intercambiar dos variables necesitas una tercera temporal. Si haces `a = b; b = a;` pierdes el valor original de `a`. Este patrón aparece en algoritmos de ordenamiento: grabatelo.

Práctica

Ejercicio 1. Escribe `Rectangulo.java`: pide base y altura, calcula area ($\text{base} * \text{altura}$) y perimetro ($2 * (\text{base} + \text{altura})$), e imprime ambos.

Solución:

```
import java.util.Scanner;

public class Rectangulo {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Base: ");
        double base = Double.parseDouble(sc.nextLine());
        System.out.print("Altura: ");
        double altura = Double.parseDouble(sc.nextLine());

        double area = base * altura;
        double perimetro = 2 * (base + altura);

        System.out.println("Area: " + area);
        System.out.println("Perimetro: " + perimetro);
        sc.close();
    }
}
```

Ejercicio 2. Escribe `EdadEnDias.java`: pide la edad en años y calcula aproximadamente cuantos días ha vivido ($\text{edad} * 365$). Luego calcula cuantas horas son ($\text{días} * 24$).

Solución: Dos multiplicaciones encadenadas. Nota que para edades normales el resultado cabe en `int`, pero si la persona tiene 100 años: 36500 días, sin problema. Este ejercicio practica reutilizar una variable calculada en el siguiente calculo.

```

import java.util.Scanner;

public class EdadEnDias {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Tu edad en años: ");
        int edad = Integer.parseInt(sc.nextLine());

        int dias = edad * 365;
        int horas = dias * 24;

        System.out.println("Has vivido aprox. " + dias + " días (" + horas + " horas).");
        sc.close();
    }
}

```

Ejercicio 3. Escribe `Monedas.java`: pide una cantidad de centavos (entero) y calcula cuantas monedas de 25, 10, 5 y 1 centavo se necesitan (usa división entera y módulo). Ejemplo: 87 centavos = 3 de 25, 1 de 10, 0 de 5, 2 de 1.

Solución explicada: Este es el ejercicio estrella de la unidad porque la división entera y el módulo brillan juntos. Tomas el cociente (cuantas monedas de esa denominación caben) y con el módulo te quedas con el resto para la siguiente denominación.

```

import java.util.Scanner;

public class Monedas {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Centavos: ");
        int centavos = Integer.parseInt(sc.nextLine());

        int de25 = centavos / 25;
        int resto = centavos % 25;
        int de10 = resto / 10;
        resto = resto % 10;
        int de5 = resto / 5;
        int de1 = resto % 5;

        System.out.println("Monedas de 25: " + de25);
        System.out.println("Monedas de 10: " + de10);
        System.out.println("Monedas de 5: " + de5);
        System.out.println("Monedas de 1: " + de1);
        sc.close();
    }
}

```

Verifica con 87: $87/25=3$, resto 12; $12/10=1$, resto 2; $2/5=0$, resto 2; $2\%5=2$. Correcto: 3, 1, 0, 2.

Ejercicio 4. Escribe `Comparador.java`: pide dos números y muestra los resultados de las 6 comparaciones (`>`, `<`, `==`, `!=`, `>=`, `<=`) como verdadero/falso.

Solución:

```

import java.util.Scanner;

public class Comparador {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Numero 1: ");
        double a = Double.parseDouble(sc.nextLine());
        System.out.print("Numero 2: ");
        double b = Double.parseDouble(sc.nextLine());

        System.out.println(a + " > " + b + " ? " + (a > b));
        System.out.println(a + " < " + b + " ? " + (a < b));
        System.out.println(a + " == " + b + " ? " + (a == b));
        System.out.println(a + " != " + b + " ? " + (a != b));
        System.out.println(a + " >= " + b + " ? " + (a >= b));
        System.out.println(a + " <= " + b + " ? " + (a <= b));
        sc.close();
    }
}

```

Ejercicio 5. Escribe `DatosPersonales.java`: pide nombre (texto), edad (entero), estatura en metros (decimal) y si es estudiante (true/false). Imprime una ficha completa. Cuidado con la trampa del Scanner: usa `nextLine()` para todo y convierte.

Solución explicada: Aquí prácticas la técnica robusta: todo se lee con `nextLine()` y se convierte. Para el booleano, `Boolean.parseBoolean` convierte "true" en `true` (todo lo demás lo toma como `false`).

```

import java.util.Scanner;

public class DatosPersonales {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.print("Nombre: ");
        String nombre = sc.nextLine();
        System.out.print("Edad: ");
        int edad = Integer.parseInt(sc.nextLine());
        System.out.print("Estatura (m): ");
        double estatura = Double.parseDouble(sc.nextLine());
        System.out.print("Eres estudiante? (true/false): ");
        boolean estudiante = Boolean.parseBoolean(sc.nextLine());

        System.out.println("---- FICHA ----");
        System.out.println("Nombre: " + nombre);
        System.out.println("Edad: " + edad);
        System.out.println("Estatura: " + estatura + " m");
        System.out.println("Estudiante: " + estudiante);
        sc.close();
    }
}

```

Ejercicio 6 (reto). Escribe `Cajero.java`: pide un monto en pesos (entero) y calcula el mínimo número de billetes de 500, 200, 100, 50 y 20 necesarios (como el ejercicio de monedas pero con billetes). Si sobra un resto menor a 20, indicalo como "resto sin billete".

Solución: Mismo patrón que Monedas, con denominaciones distintas. El reto está en manejar el resto final.

```
import java.util.Scanner;

public class Cajero {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Monto a retirar: $");
        int monto = Integer.parseInt(sc.nextLine());

        int b500 = monto / 500; monto = monto % 500;
        int b200 = monto / 200; monto = monto % 200;
        int b100 = monto / 100; monto = monto % 100;
        int b50 = monto / 50;  monto = monto % 50;
        int b20 = monto / 20;  monto = monto % 20;

        System.out.println("Billetes de 500: " + b500);
        System.out.println("Billetes de 200: " + b200);
        System.out.println("Billetes de 100: " + b100);
        System.out.println("Billetes de 50: " + b50);
        System.out.println("Billetes de 20: " + b20);
        System.out.println("Resto sin billete: $" + monto);
        sc.close();
    }
}
```

Nota la técnica de reutilizar la variable `monto` para el resto: es válida y común, aunque en código profesional a veces se prefiere una variable `resto` separada para no perder el valor original. Ambas están bien si el código queda claro.

Resumen de la unidad

- Las variables guardan datos: `tipo nombre = valor;`. Siempre inicializa antes de usar.
- Los 8 primitivos: `byte`, `short`, `int`, `long` (enteros); `float`, `double` (decimales); `char` (un carácter, comillas simples); `boolean` (`true`/`false`).
- `String` es una clase para texto (comillas dobles), con métodos como `length()` y `toUpperCase()`, y se concatena con `+`.
- Operadores aritméticos (`+ - \* / %`), con la trampa de la **división entera**; `%` da el residuo.
- `++`, `--`, `+=` y familia son atajos de uso diario.

- Relacionales (`== != > < >= <=`) devuelven boolean; **jamás compares Strings con `==`**, usa `.equals()`.
- Lógicos `&&`, `||`, `!` con evaluación de cortocircuito.
- `Scanner` lee del teclado; importa `java.util.Scanner`; cuidado con mezclar `nextInt()` y `nextLine()`: lee todo con `nextLine()` y convierte con `parseInt()`/`parseDouble()`.
- Casting: `(int)` trunca decimales; `Integer.parseInt()` convierte texto a número.
- Constantes con `final` en MAYUSCULAS.

El operador ternario: un if en una línea

Hay un operador especial con tres partes que funciona como un `if-else` compacto (lo verás en acción en la Unidad 3, pero nace aquí):

```
condicion ? valor_si_true : valor_si_false
```

Ejemplo:

```
int edad = 20;
String mensaje = (edad >= 18) ? "Mayor de edad" : "Menor de edad";
System.out.println(mensaje); // "Mayor de edad"
```

Se lee: "si edad `>= 18` es verdadero, mensaje vale 'Mayor de edad'; si no, vale 'Menor de edad'". Es elegante para asignaciones simples. No lo uses para lógica compleja: si necesitas más de una línea por rama, usa `if-else` normal. La legibilidad manda.

Otro ejemplo práctico:

```
int a = 10, b = 7;
int mayor = (a > b) ? a : b;
System.out.println("El mayor es: " + mayor);
```

La clase Math: tu calculadora científica

Java incluye la clase `Math` con funciones matemáticas listas para usar (no necesitas importarla, viene en `java.lang` que siempre está disponible):

Método	Que hace	Ejemplo	--- --- ---	<code>Math.abs(x)</code>	Valor absoluto	<code>Math.abs(-5)</code> -> <code>5</code>
				<code>Math.pow(b, e)</code>	Potencia b^e	<code>Math.pow(2, 10)</code> -> <code>1024.0</code>
				<code>Math.sqrt(x)</code>	Raíz cuadrada	<code>Math.sqrt(16)</code> -> <code>4.0</code>
				<code>Math.max(a, b)</code>	El mayor	<code>Math.max(3, 7)</code> -> <code>7</code>

`Math.min(a, b)` | El menor | `Math.min(3, 7)` -> `3` | | `Math.round(x)` | Redondea al entero más cercano | `Math.round(4.6)` -> `5` | | `Math.ceil(x)` | Redondea hacia arriba | `Math.ceil(4.1)` -> `5.0` | | `Math.floor(x)` | Redondea hacia abajo | `Math.floor(4.9)` -> `4.0` | | `Math.random()` | Aleatorio entre 0.0 y 1.0 | `Math.random()` -> `0.734...` | | `Math.PI` | La constante pi | `3.141592653589793` |

Nota: `Math.pow` devuelve `double` aunque le pases enteros. Si necesitas el resultado entero, haz casting: `int p = (int) Math.pow(2, 10);`.

Números aleatorios enteros (truco clásico): para un aleatorio entre 1 y 10:

```
int aleatorio = (int) (Math.random() * 10) + 1;
```

Cómo funciona: `Math.random()` da entre 0.0 y 0.999..., por 10 da entre 0.0 y 9.999..., el `(int)` trunca a 0-9, y el `+ 1` lo mueve a 1-10. Este patrón lo usaras en juegos y simulaciones.

Precedencia completa de operadores (referencia)

De mayor a menor prioridad:

1. Paréntesis `()`
2. `++`, `--`, `!` (unarios)
3. `*`, `/`, `%`
4. `+`, `-`
5. `<`, `<=`, `>`, `>=`
6. `==`, `!=`
7. `&&`
8. `||`
9. Ternario `? :`
10. Asignación `=`, `+=`, etc.

Ejemplo combinado:

```
boolean r = 5 + 3 * 2 > 10 && !(4 == 2 + 2) || 7 % 3 == 1;
```

Resolviendo: `3\*2=6`, `5+6=11`, `11>10=true`; `2+2=4`, `4==4=true`, `!true=false`; `true && false = false`; `7%3=1`, `1==1=true`; `false || true = true`. Resultado: `true`. Cuando una expresión se ve así de enredada, **usa paréntesis aunque no los necesites**: el código es para humanos.

Más ejemplos

Ejemplo 6: Distancia entre dos puntos

```
import java.util.Scanner;

public class Distancia {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("x1: ");
        double x1 = Double.parseDouble(sc.nextLine());
        System.out.print("y1: ");
        double y1 = Double.parseDouble(sc.nextLine());
        System.out.print("x2: ");
        double x2 = Double.parseDouble(sc.nextLine());
        System.out.print("y2: ");
        double y2 = Double.parseDouble(sc.nextLine());

        double dx = x2 - x1;
        double dy = y2 - y1;
        double distancia = Math.sqrt(dx * dx + dy * dy);

        System.out.println("Distancia: " + distancia);
        sc.close();
    }
}
```

Fórmula de Pitágoras en código. Este ejemplo muestra como las matemáticas de la escuela se traducen directo a Java.

Ejemplo 7: Adivina el número (versión simple)

```
import java.util.Scanner;

public class Adivina {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int secreto = (int) (Math.random() * 10) + 1;

        System.out.print("Adivina el numero (1-10): ");
        int intento = Integer.parseInt(sc.nextLine());

        String resultado = (intento == secreto) ? "Correcto!" : "Fallaste, era " +
secreto;
        System.out.println(resultado);
        sc.close();
    }
}
```

Junta aleatorios, entrada y el ternario. En la Unidad 4 le agregaremos intentos repetidos con ciclos.

Más práctica

Ejercicio 7. Escribe `Circulo.java`: pide el radio y calcula area ($\text{PI} * r^2$) y circunferencia ($2 * \text{PI} * r$). Usa `Math.PI` y `Math.pow`.

Solución:

```
import java.util.Scanner;

public class Circulo {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Radio: ");
        double r = Double.parseDouble(sc.nextLine());

        double area = Math.PI * Math.pow(r, 2);
        double circunferencia = 2 * Math.PI * r;

        System.out.println("Area: " + area);
        System.out.println("Circunferencia: " + circunferencia);
        sc.close();
    }
}
```

Ejercicio 8. Escribe `Redondeo.java`: pide un decimal y muestra como lo redondean `Math.round`, `Math.ceil` y `Math.floor`. Prueba con 4.5, -4.5 y 4.1. Anota las diferencias.

**Solución y análisis:* Con 4.5: round->5, ceil->5.0, floor->4.0. Con -4.5: round->-4 (redondea hacia arriba en negativos, hacia el entero más cercano "hacia positivo"), ceil->-4.0, floor->-5.0. Con 4.1: round->4, ceil->5.0, floor->4.0. Los negativos sorprenden: por eso este ejercicio existe.

Ejercicio 9. Escribe `Dados.java`: simula lanzar dos dados (aleatorios 1-6) y muestra la suma. Usa el patrón de aleatorio entero.

**Solución:*

```
public class Dados {  
    public static void main(String[] args) {  
        int dado1 = (int) (Math.random() * 6) + 1;  
        int dado2 = (int) (Math.random() * 6) + 1;  
        System.out.println("Dado 1: " + dado1);  
        System.out.println("Dado 2: " + dado2);  
        System.out.println("Suma: " + (dado1 + dado2));  
    }  
}
```

Ejercicio 10 (reto). Escribe `CambioOptimo.java`: pide un precio y un pagó (ambos enteros, pagó >= precio), calcula el cambio y desglosa en billetes/monedas de 500, 200, 100, 50, 20, 10, 5, 2, 1 usando el patrón de división y módulo. Es la combinación de los ejercicios 3 y 6 llevada al caso real de una tienda.

Mini glosario de la unidad

- **Variable:** espacio de memoria con nombre que guarda un dato de un tipo fijo.
- **Tipo primitivo:** tipo básico integrado al lenguaje (`int`, `double`, `boolean`, etc.).
- **Constante:** variable con `final` cuyo valor no puede cambiar.
- **Operador:** símbolo que transforma o combina valores (`+`, `==`, `&&`).
- **Módulo (%)**: operador que devuelve el residuo de una división.
- **Casting:** conversión explícita de un tipo a otro, con `(tipo)`.
- **Concatenar:** unir textos con `+`.

- **Cortocircuito:** `&&` y `||` no evalúan la segunda parte si la primera ya decide el resultado.

Caso de estudio: ticket de compra

Vamos a juntar todo lo de la unidad en un programa útil: un ticket de compra que pide datos, calcula y muestra el total con formato.

```
import java.util.Scanner;

public class TicketCompra {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.print("Nombre del producto: ");
        String producto = sc.nextLine();

        System.out.print("Precio unitario: ");
        double precio = Double.parseDouble(sc.nextLine());

        System.out.print("Cantidad: ");
        int cantidad = Integer.parseInt(sc.nextLine());

        System.out.print("Descuento % (0 si no hay): ");
        double descuento = Double.parseDouble(sc.nextLine());

        double subtotal = precio * cantidad;
        double montoDescuento = subtotal * (descuento / 100);
        double total = subtotal - montoDescuento;

        System.out.println("\n===== TICKET =====");
        System.out.println("Producto: " + producto);
        System.out.println("Cantidad: " + cantidad + " x $" + precio);
        System.out.println("Subtotal: $" + subtotal);
        System.out.println("Descuento (" + descuento + "%): -$" + montoDescuento);
        System.out.println("TOTAL: $" + total);
        System.out.println("=====");

        sc.close();
    }
}
```

Que observar:

- Todo se lee con `nextLine()` + `parse`: cero trampas del buffer.

- Los tipos elegidos con intención: `int` para cantidad (no existe media unidad), `double` para dinero.
- La fórmula del descuento: `subtotal * (descuento / 100)`. Ojo: si `descuento` fuera `int`, `descuento / 100` sería división entera y daría 0. Por eso es `double`.
- Las variables intermedias (`subtotal`, `montoDescuento`) hacen el código legible: cada una tiene un nombre que explica qué es.

Experimento: cambia el descuento a `int` y observa como el total sale mal. Luego corrígelo. Ese bug (división entera accidental) aparece en sistemas reales de facturación: entenderlo aquí te ahorra un problema laboral mañana.

Preguntas frecuentes de la unidad

P: Por qué `7 / 2` da 3 y no 3.5? Es un bug de Java? R: No es bug, es diseño. Cuando ambos operandos son enteros, Java hace división entera y trunca. Así funcionan C, C++, C# y casi todos los lenguajes. Si quieres decimales, convierte uno a double: `7.0 / 2` o `(double) 7 / 2`.

P: `nextInt()` y luego `nextLine()` me salta la lectura. Que hago? R: La trampa más famosa del Scanner. `nextInt()` deja el Enter en el buffer. Soluciones: agrega un `sc.nextLine()` extra para limpiarlo, o lee todo con `nextLine()` y convierte con `Integer.parseInt`. La segunda es más robusta.

P: Cuando uso `int` y cuando `double`? R: `int` para cosas contables e indivisibles (edad, cantidad de alumnos, días). `double` para mediciones y dinero (precio, promedio, temperatura). Si haces `int / int` esperando decimales, ya sabes que sale mal.

P: `==` con Strings a veces funciona y a veces no. ¿Por qué? R: Java guarda los literales String en un "pool" y reutiliza referencias iguales, entonces `==` a veces da true por casualidad. Pero con Strings contruidos dinamicamente (ej. leídos del teclado) falla. Nunca confíes en la casualidad: usa `.equals()` siempre.

P: Me marca error en `Double.parseDouble(sc.nextLine())` si escribo "12,5" con coma. R: `parseDouble` espera punto decimal, no coma: escribe `12.5`. Es formato anglosajón en programación, sin excepción. Si tu programa debe aceptar comas, reemplaza antes: `texto.replace(',', '.')`.

UNIDAD 3: Tu programa aprende a decidir

Introducción

Hasta ahora tus programas eran líneas rectas: ejecutaban de arriba a abajo, siempre igual, sin importar nada. En esta unidad les das cerebro: la capacidad de **decidir**. Con ``if-else`` tu programa elige entre caminos según condiciones; con ``switch`` elige entre muchas opciones de forma limpia. Esta es la unidad donde tus programas dejan de ser recetas fijas y se convierten en asistentes que reaccionan.

Piensa en un semáforo: si la luz es roja, te detienes; si es verde, avanzas; si es amarilla, reduces la velocidad. El semáforo no hace siempre lo mismo: decide según lo que ve. Tu programa hará lo mismo: si la calificación es mayor a 70, imprime "aprobado"; si no, "reprobado". Si el día es 1, imprime "lunes"; si es 2, "martes"...

Las decisiones están en todas partes del software real: un banco decide si aprueba tu crédito según tu historial, una tienda decide tu descuento según tu compra, un videojuego decide si tu personaje vive o muere según su energía. Dominar ``if`` y ``switch`` es dominar la lógica de la mitad de los programas que existen.

Explicación detallada

El if: la decisión fundamental

La estructura básica:

```
if (condicion) {  
    // se ejecuta SOLO si la condicion es verdadera  
}
```

La ``condicion`` es cualquier expresión que resulte en ``boolean`` (``true`` o ``false``). Ejemplo:

```
int edad = 20;  
if (edad >= 18) {  
    System.out.println("Puedes votar");  
}  
System.out.println("Fin del programa");
```

Si `edad` es 20, la condición `edad >= 18` es `true`, se imprime "Puedes votar" y luego "Fin del programa". Si `edad` fuera 15, la condición es `false`, se salta el bloque y solo se imprime "Fin del programa". El programa sigue después del `if` como si nada: el `if` no detiene nada, solo decide si ese bloque se ejecuta o se salta.

Analogía: el `if` es como una puerta con guardia. El guardia mira tu condición (tu identificación): si cumples, pasas por la puerta (se ejecuta el bloque); si no, la puerta se queda cerrada y sigues por el pasillo (el programa continua después).

if-else: el plan B

```
if (condicion) {  
    // camino A: si la condicion es verdadera  
} else {  
    // camino B: si la condicion es falsa  
}
```

Ejemplo:

```
int calificacion = 65;  
if (calificacion >= 70) {  
    System.out.println("Aprobado. Felicidades!");  
} else {  
    System.out.println("Reprobado. A estudiar mas.");  
}
```

Uno de los dos caminos se ejecuta, siempre exactamente uno. Es una bifurcación: no hay forma de que se ejecuten ambos, ni de que no se ejecute ninguno.

else if: la escalera de decisiones

Cuando hay más de dos caminos, encadenas:

```
int calificacion = 85;
if (calificacion >= 90) {
    System.out.println("Excelente: A");
} else if (calificacion >= 80) {
    System.out.println("Muy bien: B");
} else if (calificacion >= 70) {
    System.out.println("Aprobado: C");
} else {
    System.out.println("Reprobado");
}
```

Java evalúa de arriba hacia abajo y ejecuta **el primer bloque cuya condición sea verdadera**, saltándose todo lo demás. Con calificación 85: `>= 90`? No. `>= 80`? Sí: imprime "Muy bien: B" y salta hasta el final.

El orden importa muchísimo. Mira este error clásico:

```
// MAL: orden incorrecto
if (calificacion >= 70) {
    System.out.println("Aprobado: C");
} else if (calificacion >= 80) {
    System.out.println("Muy bien: B");    // NUNCA se ejecuta!
} else if (calificacion >= 90) {
    System.out.println("Excelente: A");    // NUNCA se ejecuta!
}
```

Con calificación 95, la primera condición `>= 70` ya es verdadera, así que entra ahí y jamás revisa las demás. Las ramas de B y A son código muerto. **Regla: en una escalera `else if`, ordena de la condición más exigente a la más permisiva** (de mayor a menor, en este ejemplo).

if anidados: decisiones dentro de decisiones

Puedes meter un `if` dentro de otro:

```

int edad = 25;
boolean tieneLicencia = true;

if (edad >= 18) {
    if (tieneLicencia) {
        System.out.println("Puedes conducir");
    } else {
        System.out.println("Eres mayor, pero necesitas licencia");
    }
} else {
    System.out.println("Eres menor de edad");
}

```

Los anidados expresan condiciones compuestas con mensajes distintos por caso. Pero cuidado: más de 2-3 niveles de anidación se vuelven ilegibles (el famoso "código espagueti" o "piramide de la perdición"). Si te ves con 4 niveles, es señal de que debes simplificar: combina condiciones con `&&` y `||`, o extrae a métodos (Unidad 5).

Version simplificada del ejemplo anterior con operadores lógicos:

```

if (edad >= 18 && tieneLicencia) {
    System.out.println("Puedes conducir");
} else if (edad >= 18) {
    System.out.println("Eres mayor, pero necesitas licencia");
} else {
    System.out.println("Eres menor de edad");
}

```

Mismo resultado, un solo nivel. Más limpio.

Llaves: opcionales pero peligrosas

Java permite `if` sin llaves para una sola instrucción:

```

if (edad >= 18)
    System.out.println("Puedes votar");

```

Funciona, pero es una trampa: si después agregas una segunda línea pensando que está dentro del `if`, no lo estará:

```
if (edad >= 18)
    System.out.println("Puedes votar");
    System.out.println("Bienvenido"); // ESTO SIEMPRE SE EJECUTA (fuera del if!)
```

La indentación engaña al humano, pero Java ignora la indentación: sin llaves, solo la primera instrucción pertenece al `if`. **Buena práctica de este libro: SIEMPRE usa llaves**, aunque sea una sola línea. Te ahorra bugs tontos y tu código se ve profesional.

El problema del "dangling else"

Con `if` anidados sin llaves claras, un `else` puede pegarse al `if` equivocado:

```
if (edad >= 18)
    if (tieneLicencia)
        System.out.println("Puedes conducir");
else
    System.out.println("???"); // A que if pertenece este else?
```

En Java, el `else` siempre se pega al `if` más cercano sin pareja (al de `tieneLicencia`). Si tu intención era otra, el programa hace lo contrario de lo que pensabas. Otra razón para usar llaves siempre: con llaves, la ambigüedad desaparece.

switch: elige entre muchas opciones

Cuando tienes muchas ramas según el valor de UNA variable, el `switch` es más limpio que una escalera de `if`:

```

int dia = 3;
switch (dia) {
    case 1:
        System.out.println("Lunes");
        break;
    case 2:
        System.out.println("Martes");
        break;
    case 3:
        System.out.println("Miercoles");
        break;
    case 4:
        System.out.println("Jueves");
        break;
    case 5:
        System.out.println("Viernes");
        break;
    default:
        System.out.println("Fin de semana");
}

```

Cómo funciona: Java evalúa `dia` una vez, salta directamente al `case` que coincida, ejecuta desde ahí, y el `break` lo saca del `switch`. El `default` es el comodín: se ejecuta si ningún `case` coincidió (es opcional, pero ponlo casi siempre).

Con `dia = 3` imprime "Miercoles". Con `dia = 7` imprime "Fin de semana". Con `dia = 1` imprime "Lunes".

El `break` es sagrado. Sin `break`, Java NO se detiene al final del `case`: sigue ejecutando los siguientes (esto se llama **fall-through**, caída). Mira:

```

int dia = 2;
switch (dia) {
    case 1:
        System.out.println("Lunes");
    case 2:
        System.out.println("Martes");
    case 3:
        System.out.println("Miercoles");
}

```

Con `dia = 2` imprime "Martes" Y "Miercoles", porque después del case 2 no hay `break` y cae al 3. A veces el fall-through se usa a proposito (varios cases que hacen lo mismo), pero el 99% de las veces es un bug por olvidar el `break`.

Fall-through intencional (el 1% legítimo): varios cases comparten el mismo código:

```
int mes = 2;
switch (mes) {
    case 12:
    case 1:
    case 2:
        System.out.println("Invierno");
        break;
    case 3:
    case 4:
    case 5:
        System.out.println("Primavera");
        break;
    // ... etc
}
```

Los cases 12, 1 y 2 caen juntos a "Invierno". Es la forma elegante de agrupar.

Que puede evaluar un switch: `int`, `char`, `String` (desde Java 7), y enums. NO puede evaluar `double`, `boolean` ni rangos. `case 1:` compara igualdad exacta: no puedes poner `case > 5:` (eso es trabajo del `if`).

Switch con Strings (muy usado en menus):

```
String opcion = "guardar";
switch (opcion) {
    case "guardar":
        System.out.println("Guardando archivo...");
        break;
    case "abrir":
        System.out.println("Abriendo archivo...");
        break;
    case "salir":
        System.out.println("Adios!");
        break;
    default:
        System.out.println("Opcion desconocida");
}
```

Nota: el `switch` con `String` usa `.equals()` internamente, así que es seguro (no sufre el problema del `==` con `Strings`).

Switch moderno (flecha): Java 14+ permite una sintaxis más corta sin `break`:

```
int dia = 3;
switch (dia) {
    case 1 -> System.out.println("Lunes");
    case 2 -> System.out.println("Martes");
    case 3 -> System.out.println("Miercoles");
    default -> System.out.println("Otro dia");
}
```

La flecha `->` ejecuta solo ese case, sin caída. Es más corta y elimina el bug del `break` olvidado. En este libro usaremos la sintaxis clásica con `break` (funciona en todas las versiones y es la que más verás en código existente), pero ya conoces la moderna.

if vs switch: cuando usar cada uno

| Situación | Usa | |---|---| | Rangos (`>= 70`, `< 18`) | `if-else` | | Comparar una variable contra valores exactos (1, 2, 3...) | `switch` | | Condiciones complejas con `&&` y `||` | `if-else` | | Menús por número o texto | `switch` | | Solo dos caminos | `if-else` | | Más de 4-5 ramas por igualdad | `switch` (más legible) |

Regla práctica: si tu escalera `else if` compara la MISMA variable con `==` una y otra vez, es un `switch` disfrazado. Convierelo.

Comparando Strings: `.equals()` en decisiones

Repito porque es vital: para saber si dos textos son iguales, usa `.equals()`:

```
String password = "secreto123";
if (password.equals("secreto123")) {
    System.out.println("Acceso permitido");
} else {
    System.out.println("Acceso denegado");
}
```

Para ignorar mayúsculas/minúsculas: `password.equalsIgnoreCase("SECRETO123")` también da `true`. Útil cuando el usuario escribe "Lunes", "lunes" o "LUNES" y quieres tratarlos igual.

Y para comparar con seguridad cuando la variable puede ser `null`: pon el literal primero: `"secreto123".equals(password)` nunca lanza `NullPointerException`, mientras que `password.equals(...)` si truena si `password` es `null`. Truco de profesional.

Errores comunes de la unidad

1. **`=` en vez de `==` en la condición:** `if (x = 5)` no compila en Java (porque 5 no es boolean), pero `if (listo = true)` SI compila y siempre es verdadero: asigna `true` a `listo` en vez de compararlo. El compilador no te avisa. Para booleanos, escribe directo `if (listo)` en vez de `if (listo == true)`: más corto y sin riesgo.
2. **Punto y coma después del if:** `if (edad >= 18); { ... }` - ese `;` termina el `if` vacío, y el bloque entre llaves SIEMPRE se ejecuta. Bug silencioso y cruel. Jamás pongas `;` después de la condición del `if`.
3. **Escalera en orden incorrecto:** las condiciones más exigentes primero (ya lo viste).
4. **Olvidar el `break` en switch:** el fall-through imprime de más.
5. **Comparar Strings con `==`:** usa `.equals()`.
6. **Condiciones que siempre son true/false:** `if (edad >= 0)` cuando edad es un `int` que jamás es negativo por como lo validaste... código muerto. Revisa tu lógica.
7. **Anidar demasiado:** más de 3 niveles de `if` = refactoriza.

Buenas prácticas

- Siempre llaves `{ }`, siempre.
- Ordena las ramas de lo más específico a lo más general.
- Un `default` en cada `switch`, aunque creas que cubriste todo.
- Prefiere condiciones positivas: `if (esValido)` se lee mejor que `if (!esInvalido)`.
- Extrae condiciones complejas a variables booleanas con nombre: `boolean puedeVotar = edad >= 18 && tieneINE; if (puedeVotar) { ... }`. El nombre documenta la intención.
- Para rangos usa `if`; para valores exactos múltiples usa `switch`.

Ejemplos paso a paso

Ejemplo 1: Sistema de calificaciones completo

```
import java.util.Scanner;

public class Calificaciones {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Calificacion (0-100): ");
        double cal = Double.parseDouble(sc.nextLine());

        if (cal < 0 || cal > 100) {
            System.out.println("Calificacion invalida");
        } else if (cal >= 90) {
            System.out.println("A: Excelente");
        } else if (cal >= 80) {
            System.out.println("B: Muy bien");
        } else if (cal >= 70) {
            System.out.println("C: Aprobado");
        } else {
            System.out.println("Reprobado");
        }
        sc.close();
    }
}
```

Primero valida el rango (programación defensiva: no confíes en lo que el usuario escribe), luego la escalera ordenada de mayor a menor. Este patrón "validar primero, procesar después" lo usaras toda tu carrera.

Ejemplo 2: Menu con switch

```

import java.util.Scanner;

public class Menu {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("1. Sumar");
        System.out.println("2. Restar");
        System.out.println("3. Salir");
        System.out.print("Elige: ");
        int opcion = Integer.parseInt(sc.nextLine());

        switch (opcion) {
            case 1:
                System.out.println("Elegiste sumar");
                break;
            case 2:
                System.out.println("Elegiste restar");
                break;
            case 3:
                System.out.println("Adios!");
                break;
            default:
                System.out.println("Opcion invalida");
        }
        sc.close();
    }
}

```

Este menu es el esqueleto de los proyectos de la Unidad 10 (calculadora, agenda). Guardalo bien.

Ejemplo 3: Descuento por monto (if anidado simplificado)

Una tienda da 10% de descuento si la compra pasa de \$1000, y 15% si pasa de \$5000:

```

import java.util.Scanner;

public class Descuento {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Monto de la compra: $");
        double monto = Double.parseDouble(sc.nextLine());

        double descuento = 0;
        if (monto > 5000) {
            descuento = 0.15;
        } else if (monto > 1000) {
            descuento = 0.10;
        }

        double total = monto * (1 - descuento);
        System.out.println("Descuento aplicado: " + (descuento * 100) + "%");
        System.out.println("Total a pagar: $" + total);
        sc.close();
    }
}

```

Iniciar `descuento = 0` y solo cambiarlo si aplica es un patrón limpio: evita repetir el calculo del total en cada rama.

Ejemplo 4: Clasificador de triangulos

```

import java.util.Scanner;

public class Triangulo {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Lado a: ");
        double a = Double.parseDouble(sc.nextLine());
        System.out.print("Lado b: ");
        double b = Double.parseDouble(sc.nextLine());
        System.out.print("Lado c: ");
        double c = Double.parseDouble(sc.nextLine());

        if (a + b <= c || a + c <= b || b + c <= a) {
            System.out.println("No es un triangulo valido");
        } else if (a == b && b == c) {
            System.out.println("Equilatero: 3 lados iguales");
        } else if (a == b || b == c || a == c) {
            System.out.println("Isosceles: 2 lados iguales");
        } else {
            System.out.println("Escaleno: todos distintos");
        }
        sc.close();
    }
}

```

Primero valida que los lados formen un triángulo (la suma de dos lados debe ser mayor al tercero), luego clasifica. El orden de las ramas importa: el equilatero se revisa antes que el isosceles, porque un equilatero también cumple "2 lados iguales".

Ejemplo 5: Piedra, papel o tijera (lógica de juego)

```

import java.util.Scanner;

public class Juego {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int pc = (int) (Math.random() * 3); // 0=piedra, 1=papel, 2=tijera

        System.out.print("Elige (piedra/papel/tijera): ");
        String jugador = sc.nextLine().toLowerCase();

        String[] opciones = {"piedra", "papel", "tijera"};
        // (los arreglos se ven a fondo en la Unidad 5; aqui solo usamos uno
simple)
        String eleccionPc = opciones[pc];
        System.out.println("La computadora eligio: " + eleccionPc);

        if (jugador.equals(eleccionPc)) {
            System.out.println("Empate!");
        } else if ((jugador.equals("piedra") && eleccionPc.equals("tijera"))
            || (jugador.equals("papel") && eleccionPc.equals("piedra"))
            || (jugador.equals("tijera") && eleccionPc.equals("papel"))) {
            System.out.println("Ganaste!");
        } else {
            System.out.println("Perdiste!");
        }
        sc.close();
    }
}

```

Las condiciones de victoria se agrupan con `||`: ganas si (piedra vence tijera) O (papel vence piedra) O (tijera vence papel). Los paréntesis agrupan cada caso para que se lea claro.

Práctica

Ejercicio 1. Escribe `MayorEdad.java`: pide la edad y di si la persona es nino (<13), adolescente (13-17), adulto (18-64) o adulto mayor (65+). Usa escalera `else if` ordenada.

Solución:

```

import java.util.Scanner;

public class MayorEdad {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Edad: ");
        int edad = Integer.parseInt(sc.nextLine());

        if (edad < 0) {
            System.out.println("Edad invalida");
        } else if (edad < 13) {
            System.out.println("Nino");
        } else if (edad < 18) {
            System.out.println("Adolescente");
        } else if (edad < 65) {
            System.out.println("Adulto");
        } else {
            System.out.println("Adulto mayor");
        }
        sc.close();
    }
}

```

Nota el orden ascendente aquí: `edad < 13`, luego `< 18`, luego `< 65`. Funciona porque cada condición filtra: si llegó al segundo `else if`, ya se que edad ≥ 13 . Las escaleras pueden ordenarse ascendente o descendente según como escribas las condiciones; lo importante es que cada rama sea alcanzable.

Ejercicio 2. Escribe `Bisiesto.java`: un año es bisiesto si es divisible entre 4, EXCEPTO los divisibles entre 100, A MENOS que también sean divisibles entre 400. (2000 fue bisiesto, 1900 no, 2024 si.)

Solución explicada: La regla completa en una condición:

```
import java.util.Scanner;

public class Bisiesto {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Año: ");
        int ano = Integer.parseInt(sc.nextLine());

        boolean esBisiesto = (ano % 4 == 0 && ano % 100 != 0) || (ano % 400 == 0);

        if (esBisiesto) {
            System.out.println(ano + " es bisiesto");
        } else {
            System.out.println(ano + " no es bisiesto");
        }
        sc.close();
    }
}
```

Extraer la condición a `esBisiesto` hace el `if` legible. Prueba con 2000 (true), 1900 (false), 2024 (true), 2023 (false).

Ejercicio 3. Escribe `DiasMes.java` con switch: dado el número de mes (1-12), imprime cuantos días tiene (febrero: 28, "29 si es bisiesto" lo puedes simplificar a 28). Usa fall-through intencional para agrupar los meses de 31 días.

Solución:

```

import java.util.Scanner;

public class DiasMes {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Mes (1-12): ");
        int mes = Integer.parseInt(sc.nextLine());

        switch (mes) {
            case 1: case 3: case 5: case 7: case 8: case 10: case 12:
                System.out.println("31 dias");
                break;
            case 4: case 6: case 9: case 11:
                System.out.println("30 dias");
                break;
            case 2:
                System.out.println("28 dias");
                break;
            default:
                System.out.println("Mes invalido");
        }
        sc.close();
    }
}

```

Ejercicio 4. Escribe `Login.java`: define usuario "admin" y clave "java123" en el código. Pide ambos al usuario y permite el acceso solo si los dos coinciden (usa `.equals()`). Si falla, di cuál de los dos estuvo mal (o ambos).

Solución:

```

import java.util.Scanner;

public class Login {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Usuario: ");
        String usuario = sc.nextLine();
        System.out.print("Clave: ");
        String clave = sc.nextLine();

        boolean usuarioOk = usuario.equals("admin");
        boolean claveOk = clave.equals("java123");

        if (usuarioOk && claveOk) {
            System.out.println("Bienvenido, admin!");
        } else if (!usuarioOk && !claveOk) {
            System.out.println("Usuario y clave incorrectos");
        } else if (!usuarioOk) {
            System.out.println("Usuario incorrecto");
        } else {
            System.out.println("Clave incorrecta");
        }
        sc.close();
    }
}

```

(En sistemas reales las claves jamás se guardan en texto plano en el código; se guardan cifradas en bases de datos. Esto es solo práctica de lógica.)

Ejercicio 5. Escribe `IMC.java`: pide peso (kg) y estatura (m), calcula $IMC = \text{peso} / \text{estatura}^2$, y clasifica: bajo peso (<18.5), normal (18.5-24.9), sobrepeso (25-29.9), obesidad (>=30).

Ejercicio 6 (reto). Escribe `CajeroMejorado.java`: combina la Unidad 2 con esta. Pide un monto a retirar; si el monto no es múltiplo de 20, rechaza la operación con mensaje; si es mayor a 10000, pide confirmación (el usuario escribe "si"/"no"); solo entonces desglosa los billetes. Usa `if` para las validaciones y el desglose de la Unidad 2.

Resumen de la unidad

- `if (condicion) { }` ejecuta el bloque solo si la condición es `true`.
- `else` es el plan B; `else if` encadena múltiples caminos (ordena de más exigente a más permisivo).

- Siempre usa llaves; jamás pongas `;` después de la condición del `if`.
- `switch` elige entre valores exactos de una variable; cada `case` necesita su `break` (o cae al siguiente); `default` es el comodín.
- `switch` acepta `int`, `char`, `String` y enums; no acepta rangos ni `double`.
- Compara Strings con `.equals()` (o `.equalsIgnoreCase()`), nunca con `==`.
- Extrae condiciones complejas a variables booleanas con nombre para legibilidad.
- Valida primero la entrada del usuario, procesa después.

Tablas de verdad aplicadas a decisiones reales

Los operadores lógicos de la Unidad 2 cobran vida aquí. Práctica leyendolos en voz alta como frases:

```
boolean esFinDeSemana = (dia == 6) || (dia == 7);
// "es fin de semana si el dia es 6 O el dia es 7"

boolean puedeEntrar = tieneBoleto && !estaLleno;
// "puede entrar si tiene boleto Y NO esta lleno"

boolean descuentoEspecial = (edad >= 65 || edad <= 12) && compraMinima;
// "descuento si (es adulto mayor O nino) Y cumple la compra minima"
```

Los paréntesis no solo controlan el orden: documentan tu pensamiento. `(a || b) && c` es distinto de `a || (b && c)`: en el primero, el OR se resuelve primero; en el segundo, el AND. Cuando mezcles `&&` y `||`, pon paréntesis siempre, aunque sepas la precedencia. Tu yo del futuro te lo agradecerá.

Decisiones con múltiples variables: el método de la tabla

Cuando una decisión depende de 2-3 booleanos, dibuja la tabla de casos antes de codificar. Ejemplo: una app de envíos.

Es fragil	Es urgente	Acción	--- --- ---	true	true	"Embalaje especial + envió express"	
true	false	"Embalaje especial + envió normal"	false	true	"Envió express"	false	false
"Envió normal"							

Código:

```
if (esFragil && esUrgente) {
    System.out.println("Embalaje especial + envio express");
} else if (esFragil) {
    System.out.println("Embalaje especial + envio normal");
} else if (esUrgente) {
    System.out.println("Envio express");
} else {
    System.out.println("Envio normal");
}
```

La tabla te asegura no olvidar ningún caso. En sistemas reales (bancos, seguros), las reglas de negocio se documentan así antes de programarlas.

if vs ternario vs switch: guía rápida de decisión

- Una o dos ramas simples con asignación? **Ternario**: `String tipo = (edad >= 18) ? "adulto" : "menor";`
- Dos o más ramas con lógica (varias líneas, validaciones)? **if-else**.
- Muchos valores exactos de la misma variable? **switch**.
- Rangos, combinaciones con `&&`/`||`? **if-else** (switch no puede).

Caso de estudio: validador de contraseñas

Junta todo lo aprendido en un ejemplo de nivel profesional básico:

```

import java.util.Scanner;

public class PasswordCheck {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Crea tu contraseña: ");
        String pass = sc.nextLine();

        boolean largoOk = pass.length() >= 8;
        boolean tieneNumero = false;
        boolean tieneMayuscula = false;

        for (int i = 0; i < pass.length(); i++) {
            char c = pass.charAt(i);
            if (c >= '0' && c <= '9') tieneNumero = true;
            if (c >= 'A' && c <= 'Z') tieneMayuscula = true;
        }
        // (el for se estudia a fondo en la Unidad 4; aquí solo recorre los
        caracteres)

        if (!largoOk) {
            System.out.println("Muy corta: mínimo 8 caracteres");
        } else if (!tieneNumero) {
            System.out.println("Agrega al menos un número");
        } else if (!tieneMayuscula) {
            System.out.println("Agrega al menos una mayuscula");
        } else {
            System.out.println("Contraseña válida. Bien hecho!");
        }
        sc.close();
    }
}

```

El programa revisa tres reglas y reporta LA PRIMERA que falle (orden de prioridad: largo, número, mayúscula). Este patrón de "validar por prioridades y reportar el primer falló" es exactamente cómo funcionan los formularios web reales.

Mini glosario de la unidad

- **Condición:** expresión que se evalúa a `true` o `false` y controla una decisión.
- **Bloque:** código entre `{}` que se ejecuta como una unidad.
- **Escalera else-if:** cadena de `if` / `else if` / `else` para múltiples caminos excluyentes.

- **Fall-through:** en ``switch``, cuando la ejecución cae al siguiente ``case`` por falta de ``break``.
- **Caso (case):** cada valor comparado dentro de un ``switch``.
- **Default:** rama comodín del ``switch`` cuando ningún ``case`` coincide.
- **Código muerto:** código que nunca puede ejecutarse (ramas inalcanzables).
- **Programación defensiva:** validar entradas y casos borde antes de procesar.

Laboratorio: máquina expendedora

Una máquina expendedora es el ejemplo perfecto de ``switch`` + ``if`` anidados: eliges producto (`switch`), y según tu dinero decide si te lo da, te pide más, o te da cambio (`if`).

```

import java.util.Scanner;

public class Expendedora {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.println("=== MAQUINA EXPENDEDORA ===");
        System.out.println("1. Refresco $15");
        System.out.println("2. Papas $12");
        System.out.println("3. Chocolate $10");
        System.out.print("Elige producto: ");
        int opcion = Integer.parseInt(sc.nextLine());

        int precio = 0;
        String producto = "";
        switch (opcion) {
            case 1: producto = "Refresco"; precio = 15; break;
            case 2: producto = "Papas"; precio = 12; break;
            case 3: producto = "Chocolate"; precio = 10; break;
            default:
                System.out.println("Producto inexistente.");
                sc.close();
                return; // termina el programa aqui
        }

        System.out.print("Ingresa tu dinero: $");
        int dinero = Integer.parseInt(sc.nextLine());

        if (dinero < precio) {
            System.out.println("Te faltan $" + (precio - dinero) + ". Vuelve con mas dinero.");
        } else {
            int cambio = dinero - precio;
            System.out.println("Aqui tienes tu " + producto + ".");
            if (cambio > 0) {
                System.out.println("Tu cambio: $" + cambio);
            } else {
                System.out.println("Pago exacto. Gracias!");
            }
        }
        sc.close();
    }
}

```

Que observar:

- El ``switch`` traduce opción -> (producto, precio). El ``default`` con ``return`` corta el programa si la opción es invalida: no tiene sentido pedir dinero por un producto que no existe.
- El ``if`` anidado distingue tres casos: falta dinero, sobra dinero, pagó exacto. Cada rama tiene su mensaje.
- Los mensajes le dicen al usuario exactamente que paso y que hacer: "Te faltan \$3" es mil veces mejor que "Error".

Reto: agrega un cuarto producto y una promoción: si el usuario compra 2 o más productos (pregunta la cantidad), aplica 10% de descuento. Necesitaras multiplicar el precio por la cantidad antes de comparar con el dinero.

Preguntas frecuentes de la unidad

P: ``if`` sin llaves funciona. Las necesito o no? R: Funciona para una sola instrucción, pero es una trampa: si luego agregas otra línea pensando que está dentro del ``if``, no lo estara. Los profesionales usan llaves siempre. Acostumbrate desde ya.

P: Cuando uso ``switch`` y cuando ``if-else``? R: ``switch`` para comparar UNA variable contra valores exactos (menus, días, opciones). ``if-else`` para rangos, condiciones compuestas o cosas que no son igualdad exacta. Si tu ``if-else`` compara la misma variable 5 veces con ``==``, es un ``switch`` disfrazado.

P: Puedo usar Strings en el ``switch``? R: Si, desde Java 7: ``switch (dia)`` con ``case "lunes":`` funciona. Recuerda el ``break`` igual que con números.

P: El ternario es obligatorio aprenderlo? R: No es obligatorio, pero lo verás en código ajeno constantemente. ``int mayor = (a > b) ? a : b;`` es un ``if-else`` de una línea. Usalo para asignaciones simples; para lógica compleja, el ``if`` normal es más legible.

P: Como evito la "piramide de la perdición" (ifs anidados profundos)? R: Tres tecnicas: 1) validaciones al inicio con ``return`` temprano (guard clauses), 2) extraer bloques a métodos con nombre, 3) combinar condiciones con ``&&`` o ``||`` en vez de anidar. Si tu código se recorre a la derecha más de 3 niveles, refactoriza.

UNIDAD 4: Repite sin repetirte - Los ciclos

Introducción

Imagina que debes imprimir los números del 1 al 1000. Con lo que sabes hasta ahora, escribirías 1000 líneas de `println`. Impensable. Los **ciclos** (loops) resuelven esto: repiten un bloque de código mientras una condición se cumpla. Tres líneas de ciclo reemplazan mil líneas manuales. Esta es la unidad donde tu código se vuelve poderoso de verdad.

Piensa en una tortillería: la máquina no hace una tortilla y se detiene a pensar; repite el mismo movimiento mientras haya masa. Un ciclo es igual: repite mientras haya trabajo. Los videojuegos repiten "dibujar pantalla, leer controles" 60 veces por segundo; un banco repite "procesar transacción" millones de veces al día; tu programa repetirá "pedir dato, validar, guardar" hasta que el usuario diga basta.

Java tiene tres ciclos: `while`, `do-while` y `for`. Los tres pueden hacer lo mismo en teoría, pero cada uno brilla en situaciones distintas. Al terminar esta unidad sabrás elegir el correcto sin dudar.

Explicación detallada

while: repite mientras sea verdad

```
while (condicion) {  
    // se repite mientras la condicion sea true  
}
```

El ciclo `while` es el más simple de entender: **primero pregunta, después ejecuta**. Si la condición es falsa desde el inicio, el bloque no se ejecuta ni una sola vez.

```
int i = 1;  
while (i <= 5) {  
    System.out.println("Vuelta " + i);  
    i++;  
}  
System.out.println("Fin");
```

Salida:

```
Vuelta 1
Vuelta 2
Vuelta 3
Vuelta 4
Vuelta 5
Fin
```

Anatomía de todo ciclo `while` bien hecho (memoriza estas 3 partes):

1. **Inicialización** (antes del ciclo): `int i = 1;` - preparas la variable de control.
2. **Condición** (en el paréntesis): `i <= 5` - mientras sea verdadera, se repite.
3. **Avance** (dentro del bloque): `i++;` - acercas la condición a volverse falsa.

Si falta el avance, la condición nunca cambia y el ciclo **jamás termina**: el temido **ciclo infinito**. Tu programa se congela imprimiendo sin parar hasta que lo detengas (Ctrl+C en la terminal). Todo programador ha creado ciclos infinitos; la diferencia es que los buenos los detectan rápido. Cuando un programa "se queda pegado", tu primer sospechoso es un ciclo sin avance.

Analogía: el `while` es como regar el jardín "mientras la tierra este seca". Revisas (condición), riegas un poco (bloque), vuelves a revisar. Si nunca revisas de nuevo (sin avance), riegas hasta inundar todo.

El contador y el acumulador: los dos patrones basicos

Contador: cuenta cuantas veces paso algo.

```
int contador = 0;
while (contador < 10) {
    contador++; // cada vuelta suma 1
}
System.out.println("Se repitio " + contador + " veces");
```

Acumulador: suma valores en cada vuelta.

```
int suma = 0;
int n = 1;
while (n <= 100) {
    suma = suma + n; // o suma += n;
    n++;
}
System.out.println("Suma del 1 al 100: " + suma); // 5050
```

La suma del 1 al 100 da 5050 (Gauss lo calculo de nino con un truco matematico; tu programa lo hace a la fuerza bruta en microsegundos). Contador y acumulador aparecen en casi todos los programas con ciclos: contar aprobados, sumar ventas, acumular puntos.

while con centinela: repetir hasta que el usuario diga basta

Muchas veces no sabes cuantas vueltas daras. Usas un **valor centinela**: un dato especial que significa "ya termine".

```
import java.util.Scanner;

public class PromedioFlexible {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        double suma = 0;
        int cantidad = 0;

        System.out.print("Calificacion (o -1 para terminar): ");
        double cal = Double.parseDouble(sc.nextLine());

        while (cal != -1) {
            suma += cal;
            cantidad++;
            System.out.print("Calificacion (o -1 para terminar): ");
            cal = Double.parseDouble(sc.nextLine());
        }

        if (cantidad > 0) {
            System.out.println("Promedio: " + (suma / cantidad));
        } else {
            System.out.println("No ingresaste calificaciones");
        }
        sc.close();
    }
}
```

El `-1` es el centinela: imposible como calificación real, perfecto como señal de salida. Nota el `if (cantidad > 0)`: si el usuario escribe `-1` de inmediato, evitamos dividir entre cero. Programación defensiva otra vez.

do-while: ejecuta primero, pregunta después

```
do {  
    // se ejecuta AL MENOS UNA VEZ  
} while (condicion);
```

La diferencia con `while` es el orden: `do-while` **ejecuta primero y pregunta después**. Garantiza mínimo una ejecución. Es ideal para menus: quieres mostrar el menu al menos una vez antes de preguntar si continuar.

```
import java.util.Scanner;  
  
public class MenuRepetido {  
    public static void main(String[] args) {  
        Scanner sc = new Scanner(System.in);  
        int opcion;  
  
        do {  
            System.out.println("1. Saludar");  
            System.out.println("2. Despedir");  
            System.out.println("0. Salir");  
            System.out.print("Elige: ");  
            opcion = Integer.parseInt(sc.nextLine());  
  
            if (opcion == 1) {  
                System.out.println("Hola!");  
            } else if (opcion == 2) {  
                System.out.println("Adios!");  
            }  
        } while (opcion != 0);  
  
        System.out.println("Programa terminado");  
        sc.close();  
    }  
}
```

Con `while` normal tendrías que inicializar `opcion` con un valor falso antes del ciclo o duplicar el menu; con `do-while` el código queda natural. **Regla: si el bloque debe correr al menos una vez, usa `do-while`.**

Ojo con el `;` al final del `do-while`: `} while (condicion);` SI lleva punto y coma (es la única estructura de control que lo lleva). Olvidarlo es error de compilación.

for: el ciclo con todo integrado

```
for (inicializacion; condicion; avance) {  
    // bloque  
}
```

El `for` mete las 3 partes del ciclo en una sola línea:

```
for (int i = 1; i <= 5; i++) {  
    System.out.println("Vuelta " + i);  
}
```

Es idéntico al `while` equivalente, pero más compacto y más difícil de arruinar: la inicialización, la condición y el avance están juntos, a la vista. Por eso **el `for` es el rey cuando sabes cuantas vueltas daras** (o puedes calcularlas).

Anatomía:

- `int i = 1` - se ejecuta UNA vez al inicio. La variable `i` solo existe dentro del `for` (no la puedes usar después).
- `i <= 5` - se revisa ANTES de cada vuelta (igual que `while`).
- `i++` - se ejecuta DESPUÉS de cada vuelta.
- El orden real es: inicialización -> [condición -> bloque -> avance] -> [condición -> bloque -> avance]...

El error clásico del for: el off-by-one (error por uno). Quieres 10 vueltas:

```
// 10 vueltas correctas (i de 0 a 9, o de 1 a 10):  
for (int i = 0; i < 10; i++)    // i = 0,1,2,...,9    -> 10 vueltas  
for (int i = 1; i <= 10; i++)   // i = 1,2,...,10     -> 10 vueltas  
  
// ERRORES tipicos:  
for (int i = 0; i <= 10; i++)   // 11 vueltas! (0 a 10)  
for (int i = 1; i < 10; i++)    // 9 vueltas! (1 a 9)
```

En Java la convención es empezar en 0 con ``<``: `for (int i = 0; i < n; i++)` da exactamente `n` vueltas. Los arreglos (Unidad 5) empiezan en 0, así que esta convención te prepara para ellos.

for con paso distinto de 1

El avance puede ser lo que quieras:

```

for (int i = 0; i <= 20; i += 2) {
    System.out.println(i); // pares del 0 al 20
}

for (int i = 10; i >= 1; i--) {
    System.out.println(i); // cuenta regresiva 10..1
}
System.out.println("Despegue!");

for (int i = 1; i <= 1024; i *= 2) {
    System.out.println(i); // potencias de 2: 1,2,4,8,...,1024
}

```

Ciclos anidados: un ciclo dentro de otro

```

for (int fila = 1; fila <= 3; fila++) {
    for (int col = 1; col <= 4; col++) {
        System.out.print "[" + fila + "," + col + " ] ");
    }
    System.out.println();
}

```

Salida:

```

[1,1] [1,2] [1,3] [1,4]
[2,1] [2,2] [2,3] [2,4]
[3,1] [3,2] [3,3] [3,4]

```

El ciclo externo (filas) da 3 vueltas; en CADA vuelta, el interno (columnas) da 4 vueltas completas. Total: $3 \times 4 = 12$ iteraciones. Los anidados sirven para tablas, matrices, dibujos y todo lo bidimensional. **Cuidado con el rendimiento:** dos ciclos de 1000 vueltas = 1 millón de iteraciones; tres = mil millones. Los anidados crecen rápido.

break y continue: control fino

- **`break`**: sale del ciclo inmediatamente (ya lo viste en **`switch`**; aquí rompe el ciclo).
- **`continue`**: salta a la siguiente vuelta, ignorando el resto del bloque actual.

```

for (int i = 1; i <= 10; i++) {
    if (i == 3) {
        continue; // salta el 3
    }
    if (i == 8) {
        break; // se detiene en el 8 (no lo imprime)
    }
    System.out.println(i);
}
// Imprime: 1 2 4 5 6 7

```

Usos legítimos: `break` para "ya encuentre lo que buscaba, no sigas buscando"; `continue` para "este caso no me interesa, pasa al siguiente". No abuses: un ciclo con 5 `break`s es difícil de seguir. Una o dos salidas claras está bien.

while vs do-while vs for: guía de elección

| Situación | Usa | |---|---| | Sabes cuantas vueltas (o es calculable) | `for` | | No sabes cuantas vueltas; depende de una condición | `while` | | El bloque debe ejecutarse al menos una vez (menus) | `do-while` | | Recorrer un rango de números | `for` | | Leer datos hasta un centinela | `while` | | Validar entrada del usuario (repetir hasta que sea valida) | `do-while` o `while` |

Errores comunes de la unidad

1. **Ciclo infinito por falta de avance:** el `i++` olvidado. Sintoma: el programa se congela. Cura: Ctrl+C y revisar el avance.
2. **Off-by-one:** `<` vs `<=`, empezar en 0 vs 1. Cuando dudes, prueba con números pequeños (n=1, n=2) y cuenta a mano.
3. **Punto y coma después del for/while:** `for (int i = 0; i < 5; i++); { ... }` - el `;` crea un ciclo vacío que da 5 vueltas sin hacer nada, y el bloque se ejecuta una sola vez. Bug silencioso idéntico al del `if`.
4. **Usar la variable del for fuera del for:** `for (int i = 0; ...)` y luego `System.out.println(i);` fuera: no compila, `i` no existe ahí. Si la necesitas después, declárala antes del ciclo.
5. **Condición que nunca es falsa con `<=` vs `<` en doubles:** evita contadores `double` en ciclos (errores de precisión); usa `int`.
6. **Modificar la variable de control dentro del bloque de formas raras:** si el `for` ya hace `i++`, no hagas otro `i++` dentro (avanza doble y salta valores).

7. **`break` olvidado en busquedas:** si buscas algo en un ciclo y lo encuentras, sal con ``break``; si no, sigues buscando en vano.

Buenas prácticas

- ``for`` para vueltas contadas, ``while`` para condiciones, ``do-while`` para "al menos una vez".
- Nombres cortos (``i``, ``j``, ``k``) para índices de ciclo; descriptivos para lo demás.
- Inicializa contadores y acumuladores antes del ciclo (cero es tu amigo).
- Un ciclo = una responsabilidad. Si tu ciclo hace 5 cosas distintas, partelo.
- Comenta la intención del ciclo, no la mecánica: ``// busca el primer negativo`` mejor que ``// ciclo for de i``.

Ejemplos paso a paso

Ejemplo 1: Tabla de multiplicar

```
import java.util.Scanner;

public class TablaMultiplicar {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Que tabla quieres? ");
        int n = Integer.parseInt(sc.nextLine());

        for (int i = 1; i <= 10; i++) {
            System.out.println(n + " x " + i + " = " + (n * i));
        }
        sc.close();
    }
}
```

El ciclo clásico de la escuela, ahora en código. Los paréntesis en ``(n * i)`` aseguran que multiplique antes de concatenar.

Ejemplo 2: Adivina el número con intentos (juego completo)

```
import java.util.Scanner;

public class Adivina2 {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int secreto = (int) (Math.random() * 100) + 1;
        int intentos = 0;
        int numero = 0;

        System.out.println("Adivina el numero (1-100). Tienes 7 intentos.");

        while (intentos < 7 && numero != secreto) {
            System.out.print("Intento " + (intentos + 1) + ": ");
            numero = Integer.parseInt(sc.nextLine());
            intentos++;

            if (numero < secreto) {
                System.out.println("Muy bajo");
            } else if (numero > secreto) {
                System.out.println("Muy alto");
            }
        }

        if (numero == secreto) {
            System.out.println("Ganaste en " + intentos + " intentos!");
        } else {
            System.out.println("Perdiste. Era el " + secreto);
        }
        sc.close();
    }
}
```

La condición `intentos < 7 && numero != secreto` combina dos razones para seguir: te quedan intentos Y no has ganado. Es el juego de la Unidad 2, ahora con ciclos, pistas y limite. Así se construye software: por capas.

Ejemplo 3: Validador de entrada (do-while)

```

import java.util.Scanner;

public class Validador {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int edad;

        do {
            System.out.print("Dame tu edad (0-120): ");
            edad = Integer.parseInt(sc.nextLine());
            if (edad < 0 || edad > 120) {
                System.out.println("Edad invalida, intenta de nuevo.");
            }
        } while (edad < 0 || edad > 120);

        System.out.println("Edad registrada: " + edad);
        sc.close();
    }
}

```

El `do-while` es perfecto para validación: pides el dato al menos una vez y repites mientras sea invalido. Este patrón aparece en TODO programa que pide datos.

Ejemplo 4: Dibujo con ciclos anidados (piramide)

```

public class Piramide {
    public static void main(String[] args) {
        int altura = 5;
        for (int fila = 1; fila <= altura; fila++) {
            for (int esp = 1; esp <= altura - fila; esp++) {
                System.out.print(" ");
            }
            for (int ast = 1; ast <= 2 * fila - 1; ast++) {
                System.out.print("*");
            }
            System.out.println();
        }
    }
}

```

Salida:

```

    *
  ***
 *****
*****
*****

```

Cada fila tiene `(altura - fila)` espacios y `(2\*fila - 1)` asteriscos. Los dibujos con ciclos entrenan tu capacidad de encontrar patrones numericos, la habilidad central del pensamiento algoritmico.

Ejemplo 5: Fibonacci

```

public class Fibonacci {
    public static void main(String[] args) {
        int a = 0, b = 1;
        System.out.print(a + " " + b + " ");

        for (int i = 3; i <= 15; i++) {
            int siguiente = a + b;
            System.out.print(siguiente + " ");
            a = b;
            b = siguiente;
        }
        System.out.println();
    }
}

```

La serie Fibonacci (cada número es la suma de los dos anteriores: 0, 1, 1, 2, 3, 5, 8...) se genera con dos variables que "avanzan". El truco `a = b; b = siguiente;` desplaza la ventana. Este patrón de "ventana deslizante" aparece en procesamiento de datos reales.

Ejemplo 6: Tabla de multiplicar completa (anidados)

```

public class TablasTodas {
    public static void main(String[] args) {
        for (int tabla = 1; tabla <= 5; tabla++) {
            System.out.println("--- Tabla del " + tabla + " ---");
            for (int i = 1; i <= 10; i++) {
                System.out.println(tabla + " x " + i + " = " + (tabla * i));
            }
            System.out.println();
        }
    }
}

```

El externo elige la tabla, el interno recorre del 1 al 10. $5 \times 10 = 50$ líneas generadas con 8 líneas de código. Esa es la magia de los ciclos.

Práctica

Ejercicio 1. Escribe `CuentaAtras.java`: pide un número N e imprime la cuenta regresiva de N a 1, luego "Despegue!". Usa `for`.

Ejercicio 2. Escribe `SumaPares.java`: calcula la suma de todos los números pares del 1 al 100 usando `for` con paso 2, e imprime el resultado. (Respuesta: 2550.)

Solución:

```

public class SumaPares {
    public static void main(String[] args) {
        int suma = 0;
        for (int i = 2; i <= 100; i += 2) {
            suma += i;
        }
        System.out.println("Suma de pares 1-100: " + suma);
    }
}

```

Ejercicio 3. Escribe `Factorial.java`: pide N y calcula N! (N factorial = $1 \times 2 \times \dots \times N$). Usa `for`. Prueba con 5 (120) y 10 (3628800). Nota: 20! ya no cabe en `long` (usa `long` para llegar más lejos que `int`).

Solución:

```

import java.util.Scanner;

public class Factorial {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("N: ");
        int n = Integer.parseInt(sc.nextLine());

        long fact = 1;
        for (int i = 1; i <= n; i++) {
            fact *= i;
        }
        System.out.println(n + "! = " + fact);
        sc.close();
    }
}

```

El acumulador empieza en 1 (no en 0, porque multiplicar por 0 arruina todo). Detalle que separa al que entiende del que copia.

Ejercicio 4. Escribe `Primo.java`: pide un número y determina si es primo (solo divisible entre 1 y si mismo). Pista: cuenta sus divisores con `%` en un ciclo; si tiene exactamente 2 divisores, es primo.

Solución explicada: Recorres del 1 al N contando cuantos dividen exacto (`n % i == 0`). Optimización opcional: basta llegar hasta `n/2` (ningun divisor propio pasa de la mitad), o hasta la raíz cuadrada para los avanzados.

```

import java.util.Scanner;

public class Primo {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Numero: ");
        int n = Integer.parseInt(sc.nextLine());

        int divisores = 0;
        for (int i = 1; i <= n; i++) {
            if (n % i == 0) {
                divisores++;
            }
        }

        if (divisores == 2) {
            System.out.println(n + " es primo");
        } else {
            System.out.println(n + " no es primo");
        }
        sc.close();
    }
}

```

Ejercicio 5. Escribe `CajeroCiclos.java`: simula un cajero automatico con menu repetido (`do-while`): 1) consultar saldo, 2) depositar, 3) retirar (no puede quedar en negativo), 0) salir. El saldo empieza en 1000.

Solución: Es el menu de la Unidad 3 con persistencia del saldo entre vueltas y validación en el retiro. El primer "sistema" de verdad del libro.

```

import java.util.Scanner;

public class CajeroCiclos {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        double saldo = 1000;
        int opcion;

        do {
            System.out.println("\n1. Consultar    2. Depositar    3. Retirar    0.
Salir");

            System.out.print("Elige: ");
            opcion = Integer.parseInt(sc.nextLine());

            if (opcion == 1) {
                System.out.println("Saldo: $" + saldo);
            } else if (opcion == 2) {
                System.out.print("Monto a depositar: $");
                double dep = Double.parseDouble(sc.nextLine());
                if (dep > 0) {
                    saldo += dep;
                    System.out.println("Deposito exitoso");
                } else {
                    System.out.println("Monto invalido");
                }
            } else if (opcion == 3) {
                System.out.print("Monto a retirar: $");
                double ret = Double.parseDouble(sc.nextLine());
                if (ret > 0 && ret <= saldo) {
                    saldo -= ret;
                    System.out.println("Retiro exitoso");
                } else {
                    System.out.println("Fondos insuficientes o monto invalido");
                }
            } else if (opcion != 0) {
                System.out.println("Opcion invalida");
            }
        } while (opcion != 0);

        System.out.println("Gracias por usar el cajero. Saldo final: $" + saldo);
        sc.close();
    }
}

```

Ejercicio 6. Escribe `Invertido.java`: pide un número entero positivo e imprime sus dígitos al revés (12345 -> 54321). Pista: en cada vuelta toma `n % 10` (último dígito), imprímelo, y haz `n = n / 10` (quita el último dígito). Repite mientras `n > 0`.

Solución explicada: Es el algoritmo de "pelar dígitos", fundamental en programación. Con 12345: $12345 \% 10 = 5$ (imprime 5), $12345 / 10 = 1234$; $1234 \% 10 = 4$, $1234 / 10 = 123$... hasta que $n = 0$. Módulo y división entera trabajando juntos otra vez.

```
import java.util.Scanner;

public class Invertido {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Numero: ");
        int n = Integer.parseInt(sc.nextLine());

        System.out.print("Al revés: ");
        while (n > 0) {
            int digito = n % 10;
            System.out.print(digito);
            n = n / 10;
        }
        System.out.println();
        sc.close();
    }
}
```

Ejercicio 7 (reto). Escribe `AdivinaPro.java`: mejora el juego del Ejemplo 2. La computadora piensa un número del 1 al 100, el jugador tiene 7 intentos con pistas "muy bajo/muy alto", y al terminar pregunta "Jugar de nuevo? (si/no)": si dice si, todo se repite (ciclo externo). Lleva la cuenta de partidas ganadas.

Resumen de la unidad

- `while` pregunta antes de ejecutar; `do-while` ejecuta al menos una vez y pregunta después (lleva `;` al final).
- `for (init; cond; avance)` integra las 3 partes; ideal cuando sabes cuantas vueltas dar.
- Todo ciclo necesita inicialización, condición y avance; sin avance hay ciclo infinito.
- Contador (`i++`) y acumulador (`suma += x`) son los patrones base.
- Valor centinela para terminar cuando el usuario lo decida.

- ``break`` sale del ciclo; ``continue`` salta a la siguiente vuelta.
- Ciclos anidados: el interno se completa en cada vuelta del externo; cuidado con el crecimiento multiplicativo.
- Off-by-one: verifica con casos pequenos; convención Java: ``for (int i = 0; i < n; i++)``.
- Jamás pongas ``;`` después del ``for`` o ``while``.

Depuración de ciclos: tecnicas de detective

Cuando un ciclo no hace lo esperado, usa estas tecnicas en orden:

1. **Imprime la variable de control.** Agrega ``System.out.println("i=" + i);`` dentro del ciclo. Ver los valores reales revela si el avance funciona, si la condición se evalua como esperas, o si el ciclo termina antes/despues.
2. **Prueba con números pequenos.** Si tu ciclo procesa 1000 elementos y falla, pruebalo con 3. Con 3 puedes seguir cada vuelta a mano.
3. **Revisa los bordes.** La primera vuelta y la última son donde viven los bugs: el off-by-one, la división entre cero en el primer dato, el arreglo que se sale en el último índice.
4. **Cuenta las vueltas esperadas.** Antes de correr, predice: "este for debe dar 10 vueltas". Si da 9 u 11, ya sabes donde mirar.
5. **Sospecha del ``;`` fantasma.** Si el ciclo "no hace nada" pero termina rápido, busca ``for (...);`` o ``while (...);``.

Ejemplo de depuración en acción: este ciclo debería sumar del 1 al 10 pero da 55... espera, 55 es correcto ($1+2+...+10=55$). Cambiemos: debería sumar solo los impares:

```
// Version con bug: suma todo
int suma = 0;
for (int i = 1; i <= 10; i++) {
    suma += i;
}
// El programador queria solo impares. Imprimiendo i en cada vuelta
// veria que i toma valores pares tambien. Correccion:
int suma2 = 0;
for (int i = 1; i <= 10; i += 2) {
    suma2 += i;
}
// suma2 = 1+3+5+7+9 = 25. Correcto.
```

Ciclos y validación: el guardián de la entrada

Combina lo aprendido: ningún programa profesional acepta el primer dato que el usuario escribe sin revisarlo.

```
import java.util.Scanner;

public class EntradaSegura {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int dia = 0;
        boolean valido = false;

        while (!valido) {
            System.out.print("Dia del mes (1-31): ");
            String texto = sc.nextLine();
            try {
                dia = Integer.parseInt(texto);
                if (dia >= 1 && dia <= 31) {
                    valido = true;
                } else {
                    System.out.println("Fuera de rango.");
                }
            } catch (NumberFormatException e) {
                System.out.println("Eso no es un numero.");
            }
        }
        System.out.println("Dia valido: " + dia);
        sc.close();
    }
}
```

Usa `try-catch` (lo verás a fondo en la Unidad 9) para atrapar el error si el usuario escribe letras. El ciclo repite hasta tener un dato válido. Este es el estándar de oro para pedir datos: **repite hasta que sea válido, valida rango y tipo.**

Mini glosario de la unidad

- **Ciclo/loop/bucle:** estructura que repite un bloque mientras una condición se cumpla.
- **Iteración:** cada vuelta completa del ciclo.
- **Variable de control:** variable que decide cuando termina el ciclo (como `i`).
- **Ciclo infinito:** ciclo cuya condición nunca se vuelve falsa; congela el programa.

- **Centinela:** valor especial que senala el fin de la entrada de datos.
- **Acumulador:** variable que suma valores en cada iteración.
- **Off-by-one:** error de una unidad en los limites del ciclo (9 u 11 vueltas en vez de 10).
- **Anidado:** ciclo dentro de otro ciclo.

Caso de estudio: juego de adivinanza

El juego clásico: la computadora piensa un número del 1 al 100 y tu lo adivinas con pistas de "más alto / más bajo". Es el ``while`` en su habitat natural: no sabes cuantas vueltas dara.

```

import java.util.Scanner;
import java.util.Random;

public class Adivinanza {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        Random random = new Random();
        int secreto = random.nextInt(100) + 1; // 1 a 100
        int intento = 0;
        int contador = 0;

        System.out.println("Pense un numero del 1 al 100. Adivinalo!");

        while (intento != secreto) {
            System.out.print("Tu intento: ");
            intento = Integer.parseInt(sc.nextLine());
            contador++;

            if (intento < secreto) {
                System.out.println("Mas alto...");
            } else if (intento > secreto) {
                System.out.println("Mas bajo...");
            }
        }

        System.out.println("Correcto! Lo lograste en " + contador + " intentos.");
        if (contador <= 5) {
            System.out.println("Increible, eres un crack.");
        } else if (contador <= 10) {
            System.out.println("Muy bien jugado.");
        } else {
            System.out.println("A la proxima usa estrategia: empieza en 50.");
        }
        sc.close();
    }
}

```

Que observar:

- `random.nextInt(100)` da 0-99; el `+ 1` lo mueve a 1-100. Este ajuste de rangos aparece en dados, cartas, sorteos: dominalo.
- El `while (intento != secreto)` es la condición de salida natural: el ciclo vive mientras no adivines.

- El contador de intentos convierte un juego simple en un reto con niveles ("crack" si lo haces en 5 o menos).
- La estrategia optima es la búsqueda binaria (empezar en 50, luego 25 o 75...): con ella siempre ganas en 7 intentos o menos. Mencionarlo en una entrevista suma puntos.

Reto: limita a 10 intentos; si se acaban, revela el número y pregunta si quiere jugar otra vez (ciclo exterior con ``do-while``). Lleva la cuenta de partidas ganadas.

Preguntas frecuentes de la unidad

P: ``while`` o ``for``? Como elijo? R: ``for`` cuando sabes cuantas vueltas (o recorres un rango/arreglo). ``while`` cuando depende de una condición que cambia dentro (leer hasta "fin", validar entrada). ``do-while`` cuando el cuerpo debe correr al menos una vez (menus). Si usas ``while`` para algo contado, un ``for`` seria más claro.

P: Mi ciclo no termina nunca. Como lo debuggeo? R: Imprime la variable de control dentro del ciclo: ``System.out.println("i=" + i);``. Verás si no avanza, si avanza mal, o si la condición nunca se vuelve falsa. El 99% de los ciclos infinitos es variable que no se actualiza o condición al revés.

P: ``break`` y ``continue``: no son "trampa"? R: Son herramientas legítimas con usos claros: ``break`` para salir al encontrar lo buscado, ``continue`` para saltar casos invalidos. El abuso (múltiples ``break`` anidados) si es mala práctica: señal de que el ciclo necesita reestructurarse o extraerse a un método.

P: El ``for`` de dos variables (``for (i=0, j=10; ...)``) se usa? R: Existe y compila, pero rara vez es la mejor opción: confunde. Dos variables que avanzan juntas suelen indicar que el problema se modela mejor de otra forma (o con un ``while`` explícito). Conocelo para leer código ajeno; evitalo en el tuyo.

P: Como repito algo exactamente N veces sin arreglo? R: ``for (int i = 0; i < N; i++)``. Es el idioma estandar. Si N viene del usuario, valida que sea positivo antes: un N negativo simplemente no ejecuta el cuerpo (no truena), pero un N enorme puede tardar o agotar memoria.

UNIDAD 5: Guarda muchos datos y organiza tu código

Introducción

Hasta ahora cada variable guardaba UN dato: una edad, un nombre, un precio. Pero la vida real viene en grupos: las 30 calificaciones de un salón, los 12 meses del año, los 500 productos de una tienda. Para eso están los **arreglos**: una sola variable que guarda muchos valores del mismo tipo, ordenados y numerados. Y cuando tu programa crece, necesitas los **métodos**: bloques de código con nombre que hacen una tarea y se pueden reutilizar. Arreglos + métodos = el salto de "programitas" a "programas".

Analogía: si las variables son cajas de zapatos individuales, un arreglo es un **archivero** con cajones numerados: un solo mueble, muchos espacios, cada uno con su número. Y un método es como un **empleado especializado**: en vez de hacer tú mismo la misma tarea 20 veces, le enseñas una vez a tu empleado ("calcula el promedio") y luego solo le dices "hazlo" cada vez que lo necesitas.

Al terminar esta unidad sabrás guardar colecciones de datos, recorrerlas con ciclos, y partir tus programas en piezas reutilizables. Es la unidad que más se parece al trabajo real: ningún programa profesional es un solo bloque gigante.

Explicación detallada

Arreglos: muchas cajas, un solo nombre

Un **arreglo** (array) guarda varios valores del mismo tipo bajo un solo nombre. Se declara así:

```
int[] numeros = {10, 20, 30, 40};
```

- `int[]` dice: "esto es un arreglo de enteros" (los corchetes `[]` son la marca del arreglo).
- `numeros` es el nombre.
- `{10, 20, 30, 40}` son los valores iniciales, entre llaves y separados por comas.

Cada valor vive en una **posición** (índice), y aquí viene la regla de oro de Java: **los índices empiezan en 0**. En un arreglo de 4 elementos, las posiciones son 0, 1, 2, 3. No hay posición 4.

```
int[] numeros = {10, 20, 30, 40};
System.out.println(numeros[0]); // 10 (el PRIMERO)
System.out.println(numeros[3]); // 40 (el ULTIMO)
System.out.println(numeros.length); // 4 (cuantos elementos hay)
```

`numeros[2]` se lee "el elemento 2 del arreglo números". Los corchetes con un número seleccionan la posición. Y `length` (sin paréntesis: es una propiedad, no un método) te dice el tamaño.

Por qué empiezan en 0? Herencia de C y de cómo funciona la memoria: el índice es el "desplazamiento" desde el inicio. El primer elemento está a desplazamiento 0. Todos los lenguajes importantes (Java, C, C++, Python, JavaScript) usan base 0. Acostumbrate: **el último índice válido siempre es `length - 1`**.

Crear arreglos vacíos y llenarlos después

```
int[] edades = new int[5]; // arreglo de 5 enteros, vacío por ahora
edades[0] = 25;
edades[1] = 30;
edades[2] = 22;
// edades[3] y edades[4] quedan en 0 (valor por defecto)
```

`new int[5]` crea un arreglo de 5 espacios. Java los llena con valores por defecto: `0` para números, `false` para boolean, `null` para objetos (como Strings). Luego asignas con `arreglo[posición] = valor;`.

El tamaño es fijo. Un arreglo de 5 jamás crecerá a 6. Si necesitas tamaño flexible, usa `ArrayList` (Unidad 8). Elegir entre arreglo fijo y lista dinámica es una decisión de diseño que harás seguido.

Recorrer arreglos con for: el dúo inseparable

El 90% del trabajo con arreglos es recorrerlos, y el `for` es la herramienta:

```
String[] nombres = {"Ana", "Luis", "Maria", "Pedro"};

for (int i = 0; i < nombres.length; i++) {
    System.out.println("Posicion " + i + ": " + nombres[i]);
}
```

El patrón `for (int i = 0; i < arr.length; i++)` es el más escrito en Java después del `main`. Grabátele en los dedos: empieza en 0, sigue mientras `i < length` (estrictamente menor: así `i` jamás llega a `length`, que sería posición inválida), y avanza de 1 en 1.

for-each: el recorrido elegante

Cuando solo necesitas los valores (no las posiciones), el `for-each` es más limpio:

```
int[] numeros = {10, 20, 30, 40};
for (int n : numeros) {
    System.out.println(n);
}
```

Se lee: "para cada `n` en `numeros`, imprime `n`". Sin índices, sin riesgo de salirte del rango. **Usa for-each cuando no necesites la posición; usa for clásico cuando si** (por ejemplo, para modificar elementos o comparar vecinos).

Operaciones típicas con arreglos

Buscar el mayor:

```
int[] ventas = {120, 340, 210, 500, 150};
int mayor = ventas[0];
for (int i = 1; i < ventas.length; i++) {
    if (ventas[i] > mayor) {
        mayor = ventas[i];
    }
}
System.out.println("La mayor venta: " + mayor);
```

El patrón: supone que el primero es el mayor, luego compara con cada uno y actualiza. Funciona para el menor cambiando `>` por `<`.

Sumar y promediar:

```
double[] califs = {8.5, 9.0, 7.5, 10.0};
double suma = 0;
for (double c : califs) {
    suma += c;
}
double promedio = suma / califs.length;
System.out.println("Promedio: " + promedio);
```

Contar coincidencias:

```
int[] numeros = {5, 8, 5, 3, 5, 9};
int conteo = 0;
for (int n : numeros) {
    if (n == 5) {
        conteo++;
    }
}
System.out.println("El 5 aparece " + conteo + " veces");
```

Copiar un arreglo (no uses `=`):

```
int[] original = {1, 2, 3};
int[] copia = new int[original.length];
for (int i = 0; i < original.length; i++) {
    copia[i] = original[i];
}
```

Por qué no `int[] copia = original;`? Porque eso NO copia los datos: hace que `copia` apunte al MISMO arreglo. Si cambias `copia[0]`, también cambia `original[0]`. Los arreglos son objetos (referencias), y la asignación copia la referencia, no el contenido. Este es uno de los conceptos más importantes de Java: lo verás a fondo en la Unidad 6 con los objetos.

Errores comunes con arreglos

1. `ArrayIndexOutOfBoundsException`: el error más famoso de Java. Ocurre al usar un índice inválido:

```
int[] a = {1, 2, 3};
System.out.println(a[3]); // TRUENA: indices validos 0,1,2
```

Causas típicas: usar `<=` en vez de `<` en el for (`i <= a.length`), o asumir que el último índice es `length`.

2. **Arreglo de tamaño 0 o negativo:** `new int[0]` es válido pero inútil; `new int[-5]` truena.
3. **Olvidar inicializar elementos:** `String[] nombres = new String[3]; System.out.println(nombres[0].length());` truena con `NullPointerException` porque `nombres[0]` es `null` (valor por defecto de objetos).
4. **Confundir `length` (arreglos) con `length()` (Strings):** arreglos usan `arr.length` sin paréntesis; Strings usan `texto.length()` con paréntesis. Java es así; memorízalo.

Métodos: tus empleados especializados

Un **método** es un bloque de código con nombre que hace una tarea específica. Lo defines una vez y lo llamas (usas) cuantas veces quieras.

```
public static int sumar(int a, int b) {  
    int resultado = a + b;  
    return resultado;  
}
```

Anatomía:

- `public` - visible para todos (por ahora siempre `public`).
- `static` - se puede llamar sin crear objetos (en la Unidad 6 entenderás el `static` a fondo; por ahora, todos tus métodos serán `static` porque los llamas desde el `main` que también es `static`).
- `int` - el **tipo de retorno**: qué tipo de dato devuelve el método. Si no devuelve nada, se usa `void`.
- `sumar` - el nombre (camelCase empezando en minúscula, como las variables).
- `(int a, int b)` - los **parámetros**: datos que el método recibe para trabajar. Pueden ser cero, uno o varios.
- `return resultado;` - devuelve el valor al que llamó al método. El `return` termina el método inmediatamente.

Para usarlo (llamarlo) desde el `main`:

```

public class Calculadora {
    public static int sumar(int a, int b) {
        return a + b;
    }

    public static void main(String[] args) {
        int r1 = sumar(5, 3);    // r1 = 8
        int r2 = sumar(10, 20);  // r2 = 30
        System.out.println("5+3=" + r1);
        System.out.println("10+20=" + r2);
    }
}

```

Escribiste la suma UNA vez y la usaste DOS veces con datos distintos. Si mañana la suma necesita cambiar (impuestos, redondeo), la cambias en UN solo lugar. Eso es **reutilización**, y es la razón de ser de los métodos.

Analogía: el método es una receta de cocina escrita en una tarjeta. Los parámetros son los ingredientes que varían ("2 huevos" vs "3 huevos"), el cuerpo es el procedimiento, y el `return` es el platillo terminado que entregas. Escribes la receta una vez; cocinas mil veces.

Métodos void: hacen, no devuelven

```

public static void saludar(String nombre) {
    System.out.println("Hola, " + nombre + "!");
    System.out.println("Bienvenido al sistema.");
}

```

`void` significa "no devuelve nada": el método hace su trabajo (imprimir, guardar, etc.) y termina. Se llama solo, sin asignar a variable:

```

saludar("Ana"); // imprime las dos líneas

```

Parámetros vs argumentos (terminología)

- **Parámetro:** la variable en la DEFINICIÓN del método (`int a` en `sumar(int a, int b)`).
- **Argumento:** el valor en la LLAMADA (`5` y `3` en `sumar(5, 3)`).

Los argumentos se copian a los parámetros **por valor**: el método recibe una copia, no la variable original. Si el método modifica el parámetro, la variable original NO cambia (para primitivos):

```

public static void duplicar(int x) {
    x = x * 2;
    System.out.println("Dentro: " + x); // 20
}

int n = 10;
duplicar(n);
System.out.println("Fuera: " + n); // 10 (no cambio!)

```

Esto confunde al principio, pero es una protección: los métodos no pueden romper tus variables por accidente. (Con objetos la historia es distinta: lo verás en la Unidad 6.)

Return: las reglas

1. El tipo del `return` debe coincidir con el tipo declarado: si el método dice `int`, el `return` debe devolver un `int` (o algo convertible).
2. Un método con retorno NO `void` debe devolver un valor en TODOS los caminos posibles:

```

public static String clasificar(int edad) {
    if (edad >= 18) {
        return "adulto";
    } else {
        return "menor";
    }
    // Si faltara el else, el compilador se quejaria:
    // "missing return statement"
}

```

3. `return` termina el método al instante: lo que este después no se ejecuta.
4. En un método `void` puedes usar `return;` solo (sin valor) para salir antes de tiempo:

```

public static void procesar(int edad) {
    if (edad < 0) {
        System.out.println("Edad invalida");
        return; // sale del metodo aqui
    }
    System.out.println("Procesando edad " + edad);
}

```

Alcance (scope): donde vive cada variable

- Las variables declaradas DENTRO de un método (incluyendo parámetros) solo existen ahí: son **locales**. Nacen al entrar al método, mueren al salir.
- Dos métodos pueden tener variables con el mismo nombre sin conflicto: cada una vive en su método.

```
public static int sumar(int a, int b) {  
    int resultado = a + b; // 'resultado' solo existe aqui  
    return resultado;  
}  
  
public static int multiplicar(int a, int b) { // 'a' y 'b' propios  
    int resultado = a * b; // otro 'resultado', sin conflicto  
    return resultado;  
}
```

Esto es liberador: no tienes que inventar nombres únicos para todo el programa. Cada método es su propio mundo.

Sobrecarga: mismo nombre, distintas versiones

Java permite varios métodos con el mismo nombre si sus parámetros difieren (tipo o cantidad). Se llama **sobrecarga** (overloading):

```
public static int sumar(int a, int b) {  
    return a + b;  
}  
  
public static double sumar(double a, double b) {  
    return a + b;  
}  
  
public static int sumar(int a, int b, int c) {  
    return a + b + c;  
}
```

Java elige automáticamente según los argumentos: `sumar(2, 3)` usa la primera, `sumar(2.5, 3.1)` la segunda, `sumar(1, 2, 3)` la tercera. Ya usaste sobrecarga sin saberlo: `System.out.println` tiene versiones para `int`, `double`, `String`... por eso puedes imprimir lo que sea.

Por qué partir en métodos: el principio de responsabilidad

Un ``main`` de 200 líneas es una pesadilla: difícil de leer, de probar y de arreglar. La regla profesional: **cada método hace UNA cosa y la hace bien**. Señales de que necesitas partir:

- Tu método tiene más de 30-40 líneas.
- Puedes describir partes con "primero hace X, luego Y, luego Z" (X, Y, Z son métodos).
- Copiaste y pegaste código (si lo pegaste dos veces, debe ser un método).
- No puedes explicar que hace el método en una frase.

Ejemplo de refactor:

```
// ANTES: todo en el main (difícil de leer)
public static void main(String[] args) {
    // 15 líneas para pedir datos...
    // 20 líneas para calcular...
    // 10 líneas para imprimir reporte...
}

// DESPUES: main como índice
public static void main(String[] args) {
    double[] datos = pedirDatos();
    double promedio = calcularPromedio(datos);
    imprimirReporte(datos, promedio);
}
```

El ``main`` se vuelve un índice legible: "pide datos, calcula, imprime". Cada método se entiende y se prueba por separado. Así se escribe código profesional.

Errores comunes con métodos

1. **Llamar un método no estático desde ``main``**: por ahora todos tus métodos deben ser ``static``. Si olvidas ``static``, el error es ``non-static method cannot be referenced from a static context``. (En la Unidad 6 entenderás por qué.)
2. **Tipo de retorno incorrecto**: declarar ``int`` y devolver ``String``, o olvidar el ``return``.
3. **``missing return statement``**: algún camino del método no devuelve valor. Revisa todos los ``if/else``.
4. **Parámetros en orden incorrecto**: ``restar(10, 3)`` vs ``restar(3, 10)`` dan distinto. El orden importa.

5. **Confundir definir con llamar:** escribir el método no lo ejecuta; solo la llamada lo activa. Un método jamás llamado es código muerto.

Ejemplos paso a paso

Ejemplo 1: Métodos para estadísticas de un arreglo

```

public class Estadisticas {
    public static double promedio(int[] datos) {
        int suma = 0;
        for (int n : datos) {
            suma += n;
        }
        return (double) suma / datos.length;
    }

    public static int mayor(int[] datos) {
        int m = datos[0];
        for (int i = 1; i < datos.length; i++) {
            if (datos[i] > m) {
                m = datos[i];
            }
        }
        return m;
    }

    public static int menor(int[] datos) {
        int m = datos[0];
        for (int n : datos) {
            if (n < m) {
                m = n;
            }
        }
        return m;
    }

    public static void main(String[] args) {
        int[] ventas = {120, 340, 210, 500, 150};
        System.out.println("Promedio: " + promedio(ventas));
        System.out.println("Mayor: " + mayor(ventas));
        System.out.println("Menor: " + menor(ventas));
    }
}

```

Tres métodos reutilizables que trabajan sobre cualquier arreglo de enteros. Nota el casting `(double) suma / datos.length`: sin el, la división sería entera y el promedio saldría truncado.

Ejemplo 2: Buscar en un arreglo (método que devuelve posición)

```
public class Buscador {  
    // Devuelve la posicion del valor, o -1 si no existe  
    public static int buscar(int[] arr, int valor) {  
        for (int i = 0; i < arr.length; i++) {  
            if (arr[i] == valor) {  
                return i; // encontrado: salimos de inmediato  
            }  
        }  
        return -1; // terminamos el ciclo sin encontrarlo  
    }  
  
    public static void main(String[] args) {  
        int[] codigos = {101, 205, 308, 412, 520};  
        int pos = buscar(codigos, 308);  
        if (pos != -1) {  
            System.out.println("Encontrado en la posicion " + pos);  
        } else {  
            System.out.println("No existe");  
        }  
    }  
}
```

Devolver `-1` para "no encontrado" es una convención clásica (los índices válidos empiezan en 0, así que -1 nunca es una posición real). El `return i` dentro del ciclo es un `break` elegante: encuentra y sale.

Ejemplo 3: Método con validación

```

import java.util.Scanner;

public class ValidadorMetodos {
    public static int pedirEnteroEnRango(Scanner sc, String mensaje, int min, int
max) {
        int valor;
        do {
            System.out.print(mensaje + " (" + min + "-" + max + "): ");
            valor = Integer.parseInt(sc.nextLine());
            if (valor < min || valor > max) {
                System.out.println("Fuera de rango, intenta de nuevo.");
            }
        } while (valor < min || valor > max);
        return valor;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int edad = pedirEnteroEnRango(sc, "Tu edad", 0, 120);
        int dia = pedirEnteroEnRango(sc, "Dia del mes", 1, 31);
        System.out.println("Edad: " + edad + ", Dia: " + dia);
        sc.close();
    }
}

```

El validador de la Unidad 4, convertido en método reutilizable con parámetros. Ahora validar cualquier entero en cualquier rango es una sola línea. Así se construyen librerías personales.

Ejemplo 4: Sobrecarga en acción

```
public class Formateador {  
    public static String formato(double valor) {  
        return "$" + valor;  
    }  
  
    public static String formato(double valor, int decimales) {  
        double factor = Math.pow(10, decimales);  
        double redondeado = Math.round(valor * factor) / factor;  
        return "$" + redondeado;  
    }  
  
    public static void main(String[] args) {  
        System.out.println(formato(99.567));           // $99.567  
        System.out.println(formato(99.567, 2));        // $99.57  
    }  
}
```

Ejemplo 5: Arreglo de Strings con métodos

```

public class Agenda {
    public static void imprimir(String[] nombres, String[] telefonos) {
        System.out.println("--- AGENDA ---");
        for (int i = 0; i < nombres.length; i++) {
            System.out.println((i + 1) + ". " + nombres[i] + " - " + telefonos[i]);
        }
    }

    public static int contarConPrefijo(String[] telefonos, String prefijo) {
        int c = 0;
        for (String t : telefonos) {
            if (t.startsWith(prefijo)) {
                c++;
            }
        }
        return c;
    }

    public static void main(String[] args) {
        String[] nombres = {"Ana", "Luis", "Maria"};
        String[] telefonos = {"555-1234", "555-5678", "444-9999"};
        imprimir(nombres, telefonos);
        System.out.println("Con prefijo 555: " + contarConPrefijo(telefonos,
"555"));
    }
}

```

Dos arreglos paralelos (el índice `i` conecta nombre con teléfono) y métodos que los procesan. Es la antesala de los objetos: en la Unidad 6 verás como un objeto `Contacto` reemplaza elegantemente a los dos arreglos paralelos.

Práctica

Ejercicio 1. Escribe `Utilidades.java` con estos métodos y pruebalos en el `main`: `esPar(int n)` (devuelve boolean), `doble(int n)` (devuelve int), `repetir(String s, int veces)` (imprime `s` `N` veces, void).

Ejercicio 2. Escribe `InvertirArreglo.java` con un método `invertir(int[] arr)` que devuelva un NUEVO arreglo con los elementos al revés (no modifiques el original). Pruébalo con {1,2,3,4,5} -> {5,4,3,2,1}.

Solución explicada: Creas un arreglo nuevo del mismo tamaño y copias desde los extremos opuestos: `nuevo[i] = arr[arr.length - 1 - i]`. Cuando `i=0`, toma el último; cuando `i` es el último, toma el primero.

```
public class InvertirArreglo {
    public static int[] invertir(int[] arr) {
        int[] nuevo = new int[arr.length];
        for (int i = 0; i < arr.length; i++) {
            nuevo[i] = arr[arr.length - 1 - i];
        }
        return nuevo;
    }

    public static void main(String[] args) {
        int[] original = {1, 2, 3, 4, 5};
        int[] alReves = invertir(original);
        for (int n : alReves) {
            System.out.print(n + " ");
        }
        System.out.println();
    }
}
```

Ejercicio 3. Escribe `FibonacciMetodo.java`: un método `fib(int n)` que devuelva el n-esimo número de Fibonacci (`fib(1)=0`, `fib(2)=1`, `fib(3)=1`, `fib(4)=2...`). Luego en el `main` imprime los primeros 12 usando el método en un ciclo.

Solución: El método calcula hasta la posición pedida; el `main` lo llama 12 veces. Separar "calcular uno" de "mostrar varios" es diseño limpio.

```

public class FibonacciMetodo {
    public static long fib(int n) {
        if (n == 1) return 0;
        if (n == 2) return 1;
        long a = 0, b = 1;
        for (int i = 3; i <= n; i++) {
            long s = a + b;
            a = b;
            b = s;
        }
        return b;
    }

    public static void main(String[] args) {
        for (int i = 1; i <= 12; i++) {
            System.out.print(fib(i) + " ");
        }
        System.out.println();
    }
}

```

Ejercicio 4. Escribe `Palindromo.java`: un método `esPalindromo(String texto)` que devuelva `true` si el texto se lee igual al revés ("ana", "reconocer", "12321"). Pista: compara el carácter `i` con el carácter `length-1-i`.

Ejercicio 5. Escribe `Notas.java`: pide las calificaciones de 5 alumnos en un arreglo, y con métodos calcula e imprime: promedio, la más alta, la más baja, y cuantos aprobaron (≥ 70). El `main` debe ser un índice de 4 llamadas.

Ejercicio 6 (reto). Escribe `Ordenar.java`: implementa el **ordenamiento burbuja** (bubble sort) en un método `ordenar(int[] arr)` que ordene de menor a mayor. Pista: con dos ciclos anidados, compara vecinos `arr[j]` y `arr[j+1]`; si están en desorden, intercámbialos (usa variable temporal como en la Unidad 2). Repite N veces. Prueba con {5, 2, 9, 1, 7}.

**Solución explicada:* La burbuja hace "flotar" el mayor al final en cada pasada. En la pasada 1, el mayor queda al final; en la 2, el segundo mayor queda en su lugar... Con N pasadas queda ordenado.

```

public class Ordenar {
    public static void ordenar(int[] arr) {
        for (int pasada = 0; pasada < arr.length - 1; pasada++) {
            for (int j = 0; j < arr.length - 1 - pasada; j++) {
                if (arr[j] > arr[j + 1]) {
                    int temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
                }
            }
        }
    }

    public static void main(String[] args) {
        int[] datos = {5, 2, 9, 1, 7};
        ordenar(datos);
        for (int n : datos) {
            System.out.print(n + " ");
        }
        System.out.println();
    }
}

```

El `pasada` en el ciclo interno es la optimización: los últimos ya están ordenados, no los revisas de nuevo. No es el algoritmo más rápido (existen mejores), pero entender la burbuja te prepara para todos los demás.

Resumen de la unidad

- Los arreglos guardan muchos valores del mismo tipo: `int[] a = {1,2,3};` o `new int[5]`.
- Los índices empiezan en **0**; el último válido es `length - 1`; el tamaño es fijo.
- Recorre con `for (int i = 0; i < a.length; i++)` o con for-each si no necesitas la posición.
- `ArrayIndexOutOfBoundsException` = índice inválido; la causa típica es `<=` en vez de `<`.
- Asignar arreglos copia la referencia, no los datos: para copiar de verdad, ciclo elemento por elemento.
- Los métodos encapsulan tareas reutilizables: `public static Tipo nombre(params) { ... return valor; }`.
- `void` no devuelve nada; `return` entrega el valor y termina el método.

- Parámetros se copian por valor (primitivos): el método no afecta la variable original.
- Las variables locales viven solo dentro de su método (alcance).
- Sobrecarga: mismo nombre, distintos parámetros; Java elige según los argumentos.
- Parte tu código: un método = una responsabilidad; el `main` como índice legible.

Arreglos multidimensionales: tablas de verdad

Un arreglo puede guardar... arreglos. Así nacen las **matrices** (arreglos 2D), perfectas para tablas, tableros y cuadrículas:

```
int[][] tabla = {
    {1, 2, 3},
    {4, 5, 6}
};
System.out.println(tabla[0][1]); // 2 (fila 0, columna 1)
System.out.println(tabla.length); // 2 (filas)
System.out.println(tabla[0].length); // 3 (columnas de la fila 0)
```

Se recorre con ciclos anidados (los de la Unidad 4):

```
int[][] m = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
for (int fila = 0; fila < m.length; fila++) {
    for (int col = 0; col < m[fila].length; col++) {
        System.out.print(m[fila][col] + " ");
    }
    System.out.println();
}
```

Usos reales: tableros de ajedrez, mapas de videojuegos, hojas de calculo, imagenes (cada pixel es una celda). En este libro los usaremos poco, pero ya sabes que existen.

Paso de parámetros: por valor, siempre (con matiz)

Java **siempre** pasa parámetros por valor: copia el contenido de la variable al parámetro. Para primitivos, la copia es el dato (el método no afecta el original, ya lo viste). Para arreglos y objetos, la copia es **la referencia** (la dirección del objeto): el método recibe una copia de la dirección, pero apunta al MISMO objeto. Resultado: si el método modifica `arr[0]`, el original SI cambia.

```
public static void ponerCero(int[] arr) {
    arr[0] = 0; // modifica el arreglo ORIGINAL
}

int[] datos = {5, 5, 5};
ponerCero(datos);
System.out.println(datos[0]); // 0! El original cambio.
```

No es una contradicción: se copio el valor del parámetro, pero ese valor ERA una referencia al mismo arreglo. Regla mental: **con primitivos, los métodos no tocan tus variables; con arreglos/objetos, pueden tocar el contenido**. Por eso el ejercicio de invertir creaba un arreglo NUEVO en vez de modificar el recibido: era una decisión de diseño deliberada.

Recursividad: un adelanto (opcional pero poderoso)

Un método puede llamarse a si mismo. Se llama **recursividad** y es otra forma de repetir:

```
public static long factorialRec(int n) {
    if (n <= 1) {
        return 1; // caso base: aqui se detiene
    }
    return n * factorialRec(n - 1); // caso recursivo
}
```

`factorialRec(5)` = 5 * factorialRec(4) = 5 * 4 * factorialRec(3)... hasta llegar a 1. Toda recursión necesita un **caso base** que la detenga; sin el, es un ciclo infinito con esteroides (termina en `StackOverflowError`). La recursión brilla en problemas que se parten en subproblemas iguales (recorrer carpetas, árboles, algoritmos de ordenamiento avanzados). Para ciclos simples, el `for` es más claro. Ya conoces la idea; la dominaras con práctica.

Mini glosario de la unidad

- **Arreglo:** colección de tamaño fijo de valores del mismo tipo, con índices desde 0.
- **Índice:** número de posición dentro del arreglo.
- **For-each:** ciclo simplificado para recorrer valores sin usar índices.
- **Método:** bloque de código con nombre, parámetros opcionales y retorno opcional.
- **Parámetro/argumento:** variable en la definición / valor en la llamada.
- **Tipo de retorno:** tipo de dato que devuelve el método (`void` si no devuelve).

- **Alcance (scope):** region del código donde una variable existe y es visible.
- **Sobrecarga:** varios métodos con el mismo nombre y distintos parámetros.
- **Referencia:** "dirección" de un objeto en memoria; asignarla no copia el objeto.
- **Matriz:** arreglo de dos dimensiones (arreglo de arreglos).
- **Recursividad:** método que se llama a si mismo.

Laboratorio: análisis de calificaciones

Un maestro tiene las calificaciones de su grupo y quiere: promedio, la más alta, la más baja, y cuantos aprobaron (≥ 6). Todo con arreglos y métodos separados: cada calculo es un método.

```
public class AnalisisCalificaciones {

    static double promedio(double[] califs) {
        double suma = 0;
        for (double c : califs) {
            suma += c;
        }
        return suma / califs.length;
    }

    static double maxima(double[] califs) {
        double max = califs[0];
        for (int i = 1; i < califs.length; i++) {
            if (califs[i] > max) {
                max = califs[i];
            }
        }
        return max;
    }

    static double minima(double[] califs) {
        double min = califs[0];
        for (int i = 1; i < califs.length; i++) {
            if (califs[i] < min) {
                min = califs[i];
            }
        }
        return min;
    }

    static int contarAprobados(double[] califs) {
```

```

        int contador = 0;
        for (double c : califs) {
            if (c >= 6) {
                contador++;
            }
        }
        return contador;
    }

    public static void main(String[] args) {
        double[] grupo = {8.5, 6.0, 9.2, 4.5, 7.8, 5.9, 10.0, 6.3};

        System.out.println("Alumnos: " + grupo.length);
        System.out.println("Promedio: " + promedio(grupo));
        System.out.println("Maxima: " + maxima(grupo));
        System.out.println("Minima: " + minima(grupo));
        System.out.println("Aprobados: " + contarAprobados(grupo));
        System.out.println("Reprobados: " + (grupo.length - contarAprobados(grupo)));
    }
}

```

Que observar:

- El truco de inicializar `max`/`min` con el primer elemento (no con 0): si todas las calificaciones fueran menores que 0, inicializar en 0 daría un máximo falso. Con el primer elemento, el algoritmo es correcto para cualquier contenido.
- Cada método hace UNA cosa y su nombre lo dice: `promedio`, `maxima`, `minima`, `contarAprobados`. El `main` se lee como un resumen en español.
- El patrón "recorrer y acumular/comparar/contar" aparece en el 80% del trabajo con arreglos: si lo dominas aquí, lo aplicarás en todas partes.

Reto: agrega el método `mediana(double[] califs)` (ordena una copia con `Arrays.sort` y toma el valor central) y `porcentajeAprobados` que devuelva el porcentaje con un decimal.

Preguntas frecuentes de la unidad

P: Puedo cambiar el tamaño de un arreglo después de crearlo? R: No. El tamaño es fijo para siempre. Si necesitas crecer, crea un arreglo nuevo más grande y copia los elementos (o mejor: usa `ArrayList`, Unidad 8).

P: Qué pasa si no inicializo los elementos del arreglo? R: Java los inicializa solo: `0` para numéricos, `false` para boolean, `null` para objetos. Pero no confíes en eso por flojera: inicializa explícitamente lo que uses.

P: Por qué mi método no cambia la variable que le pase? R: Java pasa los primitivos por valor (una copia). El método trabaja con la copia. Para "devolver" un resultado usa ``return``. Los arreglos y objetos pasan copia de la referencia: sus CONTENIDOS si se pueden modificar desde el método.

P: ``static`` en los métodos: cuando si y cuando no? R: ``static`` = pertenece a la clase, no necesita objeto (ej. ``Math.sqrt``, tus métodos ayudantes llamados desde ``main``). Sin ``static`` = pertenece al objeto y necesita ``new``. En este libro usamos ``static`` por simplicidad; en POO real (Unidad 6+) los métodos viven en objetos.

P: Me confundo entre parámetro y argumento. Algun truco? R: Parámetro = el "lugar" en la definición (``void saludar(String nombre)``); argumento = el "valor" en la llamada (``saludar("Ana")``). El parámetro recibe, el argumento entrega. Con el tiempo sale solo.

UNIDAD 6: Piensa en objetos - POO parte 1

Introducción

Todo lo que has hecho hasta ahora es **programación estructurada**: datos sueltos (variables, arreglos) y funciones sueltas (métodos) que los manipulan. Funciona para programas pequeños, pero cuando el programa crece a miles de líneas, se vuelve un caos: ¿qué método puede tocar qué variable? ¿Dónde se guarda cada cosa? ¿Quién es responsable de qué?

La **Programación Orientada a Objetos (POO)** resuelve esto con una idea simple y poderosa: junta los datos y los comportamientos que les corresponden en una sola pieza llamada **objeto**. Un objeto `CuentaBancaria` guarda su saldo y sabe depositar, retirar y mostrar su estado. Nadie más toca su saldo directamente: si quieres cambiarlo, le pides al objeto que lo haga con sus métodos. Cada objeto es responsable de sí mismo.

Esta es LA unidad más importante del libro para tu carrera. Java es un lenguaje orientado a objetos hasta la medula: `String`, `Scanner`, `ArrayList`... todo lo que has usado SON objetos. Las empresas contratan programadores Java porque la POO permite construir sistemas enormes con equipos grandes sin que todo se rompa. Entender POO a fondo es lo que separa a un "alguien que escribe código Java" de un "programador Java".

En esta primera parte: clases y objetos, constructores, encapsulamiento, `this` y `static`. En la Unidad 7: herencia, polimorfismo, clases abstractas e interfaces (los superpoderes).

Explicación detallada

El cambio de mentalidad: de verbos a sustantivos

En programación estructurada piensas en **verbos** (acciones): calcular, imprimir, guardar. En POO piensas en **sustantivos** (cosas): `Cuenta`, `Producto`, `Estudiante`, `Perro`. Cada sustantivo se convierte en una **clase**, y de cada clase creas **objetos** concretos.

Analogía definitiva: la clase es el **molde para hacer galletas**; los objetos son **las galletas**. El molde define la forma (atributos: redonda, con chispas) y nadie se come el molde. Las galletas son las instancias reales que comes (cada una con sus propios valores: esta tiene más chispas, aquella está más dorada). Puedes hacer mil galletas del mismo molde, cada una independiente.

Otro ejemplo: la clase `Perro` define que todo perro tiene nombre, edad y puede ladrar. Los objetos son Firulais (3 años) y Rex (5 años): cada uno tiene SUS datos, pero ambos saben ladrar porque comparten el molde.

Anatomía de una clase

```
public class Perro {  
    // ATRIBUTOS (datos): lo que el objeto SABE  
    String nombre;  
    int edad;  
  
    // METODOS (comportamientos): lo que el objeto HACE  
    void ladrar() {  
        System.out.println(nombre + " dice: Guau!");  
    }  
  
    void cumplirAnos() {  
        edad++;  
        System.out.println(nombre + " ahora tiene " + edad + " anos");  
    }  
}
```

- **Atributos** (variables de instancia): cada objeto guarda sus propios valores. `nombre` y `edad` existen por objeto.
- **Métodos**: comportamientos que operan sobre los atributos del objeto. Nota que `ladrar()` usa `nombre` directamente: los métodos ven los atributos de su propio objeto sin necesidad de parámetros.

Hasta ahora tus métodos eran `static` y vivían sueltos. Los métodos de una clase (no estáticos) pertenecen a cada objeto: `firulais.ladrar()` usa el nombre de Firulais; `rex.ladrar()` usa el de Rex.

Crear y usar objetos: new y el punto

```

public class Veterinaria {
    public static void main(String[] args) {
        Perro firulais = new Perro();    // crea un objeto Perro
        firulais.nombre = "Firulais";    // asigna SUS datos
        firulais.edad = 3;

        Perro rex = new Perro();          // otro objeto, independiente
        rex.nombre = "Rex";
        rex.edad = 5;

        firulais.ladrar(); // "Firulais dice: Guau!"
        rex.ladrar();      // "Rex dice: Guau!"
        firulais.cumplirAnos(); // Firulais ahora tiene 4 (Rex sigue en 5)
    }
}

```

- `new Perro()` **crea** el objeto en memoria y devuelve su referencia (su "dirección").
- `Perro firulais` declara una variable que guarda esa referencia (no el objeto en si: la variable es como una etiqueta pegada al objeto).
- `firulais.nombre` usa el **operador punto** para acceder a lo del objeto: sus atributos y métodos.
- `firulais` y `rex` son independientes: cambiar la edad de uno no afecta al otro. Cada `new` crea un objeto nuevo.

La variable NO es el objeto. `Perro firulais = new Perro();` crea UN objeto y UNA variable que lo referencia. Si haces `Perro otro = firulais;`, NO creas otro perro: creas otra etiqueta pegada al MISMO perro. `otro.nombre = "Cambio"` también cambia `firulais.nombre`, porque son el mismo objeto. Este concepto (referencias vs objetos) es la causa del 50% de los bugs de principiantes en POO. Grabátele.

El valor null: la referencia vacía

```

Perro p;           // declarada pero sin objeto
p.ladrar();        // TRUENA: NullPointerException

```

Si declaras una variable de objeto sin asignarle un `new`, vale `null` ("no apunta a nada"). Usarla truena con `NullPointerException` (NPE), el error más famoso de Java. Los atributos de objeto también inician en `null` si no los inicializas. Regla: **crea el objeto con `new` antes de usarlo**, y cuando recibas objetos de otros lados, verifica que no sean `null` antes de usarlos (o usa el truco del literal primero en `.equals()`).

Constructores: el nacimiento controlado

En el ejemplo anterior, creamos el perro y LUEGO le pusimos nombre y edad en dos pasos. Es incomodo y peligroso: ¿qué pasa si olvidas ponerle nombre? El perro nace "incompleto". El **constructor** resuelve esto: es un método especial que se ejecuta **automáticamente al crear el objeto con `new`**, y sirve para inicializarlo desde el primer segundo.

```
public class Perro {
    String nombre;
    int edad;

    // CONSTRUCTOR: mismo nombre que la clase, SIN tipo de retorno
    public Perro(String nombreInicial, int edadInicial) {
        nombre = nombreInicial;
        edad = edadInicial;
    }

    void ladrar() {
        System.out.println(nombre + " dice: Guau!");
    }
}
```

Uso:

```
Perro firulais = new Perro("Firulais", 3); // nace completo
Perro rex = new Perro("Rex", 5);
firulais.ladrar();
```

Reglas del constructor:

1. Se llama **igual que la clase** (`Perro`), con mayúscula inicial.
2. **No tiene tipo de retorno** (ni siquiera `void`). Si le pones `void`, Java cree que es un método normal y no lo trata como constructor.
3. Se ejecuta solo con `new`. No lo puedes llamar como método normal.
4. Si NO escribes ningún constructor, Java te regala uno **por defecto** sin parámetros (`new Perro()` funciona). Pero en cuanto escribes UNO con parámetros, el por defecto **desaparece**: `new Perro()` ya no compila a menos que tu también escribas un constructor sin parámetros.

Puedes tener **varios constructores** (sobrecarga, como los métodos):

```

public Perro(String nombreInicial, int edadInicial) {
    nombre = nombreInicial;
    edad = edadInicial;
}

public Perro(String nombreInicial) {
    nombre = nombreInicial;
    edad = 0; // edad desconocida
}

```

Así `new Perro("Firulais", 3)` y `new Perro("Callejero")` ambos funcionan. Cada forma de nacer es un constructor.

this: "yo mismo"

Dentro de un método, `this` significa "este objeto, yo mismo". Su uso principal: distinguir entre el atributo y un parámetro con el mismo nombre.

```

public class Perro {
    String nombre;
    int edad;

    public Perro(String nombre, int edad) {
        this.nombre = nombre; // this.nombre = el ATRIBUTO; nombre = el PARAMETRO
        this.edad = edad;
    }
}

```

Sin `this`, `nombre = nombre;` asignaría el parámetro a si mismo (no hace nada útil). Con `this`, queda clarísimo: el atributo del objeto recibe el valor del parámetro. Esta es la forma estándar de escribir constructores en Java profesional: parámetros con el mismo nombre que los atributos + `this`. Verás este patrón en TODO código Java del mundo.

Otros usos de `this`: pasarte a ti mismo como argumento (`registro.agregar(this)`), o llamar a otro constructor de tu misma clase (`this("nombre", 0)` debe ser la primera línea). Por ahora, quédate con el uso principal.

Encapsulamiento: protege tus datos

Mira este problema:

```
public class CuentaBancaria {  
    double saldo; // atributo PUBLICO (por defecto es de paquete, casi publico)  
}  
  
CuentaBancaria c = new CuentaBancaria();  
c.saldo = -999999; // alguien pone un saldo absurdo...  
c.saldo = 10000000000; // ...o se regala dinero. Sin control!
```

Si cualquiera puede tocar `saldo` directamente, no hay forma de validar, registrar o proteger. La solución es el **encapsulamiento**: esconde los atributos (`private`) y ofrece métodos controlados (**getters y setters**) como única puerta de acceso.

```

public class CuentaBancaria {
    private double saldo; // PRIVATE: nadie de fuera lo toca directo

    public CuentaBancaria(double saldoInicial) {
        if (saldoInicial >= 0) {
            saldo = saldoInicial;
        } else {
            saldo = 0;
        }
    }

    // GETTER: lee el valor (puerta de salida)
    public double getSaldo() {
        return saldo;
    }

    // Metodos de negocio: la forma CORRECTA de modificar
    public void depositar(double monto) {
        if (monto > 0) {
            saldo += monto;
            System.out.println("Deposito exitoso. Nuevo saldo: " + saldo);
        } else {
            System.out.println("Monto invalido");
        }
    }

    public boolean retirar(double monto) {
        if (monto > 0 && monto <= saldo) {
            saldo -= monto;
            System.out.println("Retiro exitoso. Nuevo saldo: " + saldo);
            return true;
        }
        System.out.println("Fondos insuficientes o monto invalido");
        return false;
    }
}

```

Uso:

```

CuentaBancaria cuenta = new CuentaBancaria(1000);
cuenta.depositar(500); // OK, con validacion
cuenta.retirar(2000); // rechazado: no hay fondos
// cuenta.saldo = 99999; // ERROR DE COMPILACION: saldo es private
System.out.println("Saldo: " + cuenta.getSaldo());

```

Los beneficios del encapsulamiento:

1. **Validación:** ningún saldo negativo, ningún depósito absurdo. Las reglas del negocio viven en UN lugar.
2. **Flexibilidad interna:** mañana puedes cambiar cómo se guarda el saldo (por ejemplo, en centavos como ``long``) sin romper el código que usa la cuenta: los métodos ``depositar``/``retirar``/``getSaldo`` siguen igual por fuera.
3. **Control de lectura/escritura:** puedes tener solo getter (solo lectura, como el saldo: se lee pero no se asigna directo), solo setter, o ambos. Tú decides por atributo.
4. **Depuración:** si el saldo sale mal, sabes que SOLO los métodos de la clase lo tocaron. El sospechoso está acorralado.

Convención JavaBeans (estándar de la industria): atributo ``private`` + ``getNombre()`` + ``setNombre(valor)``. Los frameworks (Spring, etc.) esperan esta convención. Los IDEs generan getters/setters automáticamente (en IntelliJ: Alt+Insert o clic derecho > Generate).

```

public class Persona {
    private String nombre;
    private int edad;

    public Persona(String nombre, int edad) {
        this.nombre = nombre;
        setEdad(edad); // reutiliza la validacion del setter!
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public int getEdad() {
        return edad;
    }

    public void setEdad(int edad) {
        if (edad >= 0 && edad <= 150) {
            this.edad = edad;
        } else {
            System.out.println("Edad invalida, se mantiene: " + this.edad);
        }
    }
}

```

Nota como el constructor llama a `setEdad(edad)` en vez de asignar directo: reutiliza la validación. No dupliques reglas.

Niveles de acceso (modificadores)

Modificador	Quien lo ve	--- ---	`private`	Solo dentro de la misma clase	(sin modificador, "paquete")
			`protected`	Mismo paquete + subclases (Unidad 7)	`public`
					Todos

Regla práctica: **atributos `private` siempre; métodos `public` casi siempre.** Empieza todo privado y abre solo lo necesario. Es más fácil abrir después que cerrar un diseno expuesto.

static: lo que pertenece a la clase, no al objeto

Hasta ahora cada objeto tiene SUS atributos. Pero a veces un dato pertenece a la CLASE completa, compartido por todos los objetos. Para eso es `static`:

```
public class ContadorVisitas {
    private static int totalVisitas = 0; // UNA sola copia para todos

    private String nombreSitio;

    public ContadorVisitas(String nombreSitio) {
        this.nombreSitio = nombreSitio;
        totalVisitas++; // cada objeto nuevo suma al total compartido
    }

    public static int getTotalVisitas() {
        return totalVisitas;
    }
}
```

```
ContadorVisitas a = new ContadorVisitas("tienda.com");
ContadorVisitas b = new ContadorVisitas("blog.com");
System.out.println(ContadorVisitas.getTotalVisitas()); // 2
```

- `static` = una sola copia compartida por todos los objetos de la clase. Sin `static`, cada objeto tendría su propio `totalVisitas` (siempre sería 1).
- Se accede con `NombreClase.miembro` (no necesitas un objeto).
- Los métodos `static` solo pueden tocar miembros `static` (no tienen `this`, porque no pertenecen a ningún objeto en particular).
- Por eso el `main` es `static`: la JVM lo llama sin crear objetos.

Usos típicos de `static`: constantes (`Math.PI` es `public static final`), contadores globales, métodos utilitarios (`Math.max`, tus métodos de la Unidad 5), y el método `main`.

`final` con objetos: `final` en una variable de objeto significa que la REFERENCIA no cambia (siempre apunta al mismo objeto), pero el contenido del objeto si puede cambiar. `final Perro p = new Perro(...); p = otroPerro;` no compila; `p.setEdad(4);` si funciona.

Diseño de clases: como pensar en objetos

Cuando te den un problema, haz esto:

1. **Subraya los sustantivos:** son tus clases candidatas. "Una *tienda* vende *productos* a *clientes*, genera *pedidos*." -> clases: Tienda, Producto, Cliente, Pedido.
2. **Subraya las características:** son atributos. "El producto tiene *nombre*, *precio* y *stock*."
3. **Subraya las acciones:** son métodos. "El cliente *compra*, el pedido *calcula el total*."
4. **Busca relaciones:** un Pedido TIENE productos (composición); un Cliente HACE pedidos. (La herencia viene en la Unidad 7.)
5. **Encapsula:** atributos privados, constructores que dejan el objeto valido, métodos que representan acciones reales del dominio.

Pregunta de oro para cada método: "es responsabilidad de ESTA clase hacerlo?" El total del pedido lo calcula `Pedido`, no `Producto`. El saldo lo maneja `CuentaBancaria`, no el `main`. Cada objeto cuida lo suyo.

Errores comunes de la unidad

1. **Olvidar `new`:** declarar `Perro p;` y usar `p.ladRAR()` -> `NullPointerException`.
2. **Constructor con `void`:** `public void Perro(...)` no es constructor, es un método normal que jamás se llama solo.
3. **Poner el tipo de retorno al constructor:** `public Perro Perro(...)` no compila como constructor.
4. **Atributos publicos:** funciona, pero es mala práctica y en código profesional no pasa revisión.
5. **No inicializar en el constructor:** objetos a medio construir con `null`s y ceros que truenan después.
6. **Confundir clase con objeto:** la clase es el molde (no guarda datos de nadie); el objeto es la instancia (guarda SUS datos).
7. **`static` donde no va:** un método `static` no puede usar atributos no estaticos (no tiene `this`).
8. **Dos etiquetas, un objeto:** `Perro b = a;` no crea otro perro; cambiar por `b` cambia `a`.

Ejemplos paso a paso

Ejemplo 1: Clase Estudiante completa

```
// Archivo: Estudiante.java
public class Estudiante {
    private String nombre;
    private int edad;
    private double promedio;

    public Estudiante(String nombre, int edad) {
        this.nombre = nombre;
        setEdad(edad);
        this.promedio = 0.0;
    }

    public String getNombre() {
        return nombre;
    }

    public int getEdad() {
        return edad;
    }

    public void setEdad(int edad) {
        if (edad >= 5 && edad <= 120) {
            this.edad = edad;
        } else {
            System.out.println("Edad invalida");
        }
    }

    public double getPromedio() {
        return promedio;
    }

    public void registrarCalificacion(double cal) {
        if (cal >= 0 && cal <= 100) {
            // promedio simple con la nueva calificacion (ejemplo didactico)
            promedio = (promedio + cal) / 2;
        }
    }

    public String situacion() {
        return (promedio >= 70) ? "Aprobado" : "En riesgo";
    }

    public void mostrarFicha() {
```

```

        System.out.println("--- " + nombre + " ---");
        System.out.println("Edad: " + edad);
        System.out.println("Promedio: " + promedio + " (" + situacion() + ")");
    }
}

```

```

// Archivo: Escuela.java (clase con el main)
public class Escuela {
    public static void main(String[] args) {
        Estudiante e1 = new Estudiante("Ana", 20);
        e1.registrarCalificacion(85);
        e1.registrarCalificacion(95);
        e1.mostrarFicha();

        Estudiante e2 = new Estudiante("Luis", 22);
        e2.registrarCalificacion(60);
        e2.mostrarFicha();
    }
}

```

Cada clase en su archivo (convención Java). La clase `Escuela` solo orquesta; la lógica vive en `Estudiante`. Nota como `situacion()` usa el ternario de la Unidad 2: todo se conecta.

Ejemplo 2: Producto con stock (encapsulamiento real)

```

public class Producto {
    private String nombre;
    private double precio;
    private int stock;

    public Producto(String nombre, double precio, int stock) {
        this.nombre = nombre;
        setPrecio(precio);
        setStock(stock);
    }

    public String getNombre() {
        return nombre;
    }

    public double getPrecio() {
        return precio;
    }
}

```

```

    public void setPrecio(double precio) {
        if (precio > 0) {
            this.precio = precio;
        } else {
            System.out.println("Precio invalido");
        }
    }

    public int getStock() {
        return stock;
    }

    public void setStock(int stock) {
        if (stock >= 0) {
            this.stock = stock;
        }
    }

    public boolean vender(int cantidad) {
        if (cantidad > 0 && cantidad <= stock) {
            stock -= cantidad;
            return true;
        }
        return false;
    }

    public void reabastecer(int cantidad) {
        if (cantidad > 0) {
            stock += cantidad;
        }
    }

    public double valorInventario() {
        return precio * stock;
    }
}

```

`vender` devuelve `boolean`: el que llama sabe si la venta fue posible. El stock jamás es negativo porque el único camino para bajarlo valida. Este es encapsulamiento haciendo su trabajo.

Ejemplo 3: Contador static en acción

```
public class Sesion {
    private static int sesionesActivas = 0;
    private String usuario;

    public Sesion(String usuario) {
        this.usuario = usuario;
        sesionesActivas++;
        System.out.println(usuario + " inicio sesion. Activas: " +
sesionesActivas);
    }

    public void cerrar() {
        sesionesActivas--;
        System.out.println(usuario + " cerro sesion. Activas: " + sesionesActivas);
    }

    public static int getSesionesActivas() {
        return sesionesActivas;
    }

    public static void main(String[] args) {
        Sesion s1 = new Sesion("ana");
        Sesion s2 = new Sesion("luis");
        s1.cerrar();
        System.out.println("Total: " + Sesion.getSesionesActivas());
    }
}
```

Ejemplo 4: Punto geometrico (this y constructores múltiples)

```

public class Punto {
    private double x;
    private double y;

    public Punto() {
        this(0, 0); // llama al otro constructor: reutilizacion
    }

    public Punto(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public double distanciaAlOrigen() {
        return Math.sqrt(x * x + y * y);
    }

    public double distanciaA(Punto otro) {
        double dx = this.x - otro.x;
        double dy = this.y - otro.y;
        return Math.sqrt(dx * dx + dy * dy);
    }

    public void mostrar() {
        System.out.println("(" + x + ", " + y + ")");
    }

    public static void main(String[] args) {
        Punto p1 = new Punto();
        Punto p2 = new Punto(3, 4);
        p1.mostrar();
        p2.mostrar();
        System.out.println("Distancia al origen de p2: " + p2.distanciaAlOrigen());
        System.out.println("Distancia p1-p2: " + p1.distanciaA(p2));
    }
}

```

`this(0, 0)` en el primer constructor llama al segundo: no duplicas la inicialización. Y `distanciaA(Punto otro)` muestra objetos interactuando: un punto le pide datos a otro punto (a traves de sus atributos privados, permitido porque es la misma clase).

Práctica

Ejercicio 1. Crea la clase `Libro` con atributos privados `titulo`, `autor`, `paginas`, constructor completo, getters/setters con validación (páginas > 0), y un método `descripcion()` que devuelva un String con la ficha. Pruebala en un `main`.

Ejercicio 2. Crea la clase `Rectangulo` con `base` y `altura` privados, constructor, getters/setters validados (> 0), y métodos `area()` y `perimetro()`. En el `main`, crea dos rectangulos e imprime cuál tiene mayor area.

Solución (clase):

```
public class Rectangulo {
    private double base;
    private double altura;

    public Rectangulo(double base, double altura) {
        setBase(base);
        setAltura(altura);
    }

    public double getBase() { return base; }
    public void setBase(double base) {
        if (base > 0) this.base = base;
    }
    public double getAltura() { return altura; }
    public void setAltura(double altura) {
        if (altura > 0) this.altura = altura;
    }
    public double area() { return base * altura; }
    public double perimetro() { return 2 * (base + altura); }
}
```

Ejercicio 3. Crea la clase `Termometro` con atributo privado `celsius`, constructor, getter/setter (rango -273.15 a 1000), y métodos `aFahrenheit()` y `aKelvin()` que devuelvan la conversión. El setter debe rechazar temperaturas bajo el cero absoluto.

Ejercicio 4. Modifica `CuentaBancaria` del ejemplo de encapsulamiento: agrega atributo `titular` (String, solo lectura tras el constructor: sin setter), un contador `static` de cuentas creadas, y un método `transferir(CuentaBancaria destino, double monto)` que retire de una y deposite en otra solo si hay fondos.

Solución (método transferir):

```

public boolean transferir(CuentaBancaria destino, double monto) {
    if (this.retirar(monto)) {
        destino.depositar(monto);
        return true;
    }
    return false;
}

```

Elegante: reutiliza `retirar` (que ya valida fondos) y `depositar` (que ya valida el monto). Si el retiro falla, no se deposita nada. Objetos colaborando con sus métodos públicos, sin tocar atributos ajenos directamente.

Ejercicio 5. Crea la clase `Dado` con atributo `caras` (por defecto 6 via constructor sin parámetros, o personalizado), y método `lanzar()` que devuelva un aleatorio entre 1 y caras. Agrega un contador `static` de lanzamientos totales. En el `main`, lanza dos dados 10 veces e imprime cada tiro.

Ejercicio 6 (reto). Diseña las clases para una **biblioteca**: `Libro` (título, autor, disponible boolean), `Usuario` (nombre, lista de libros prestados como arreglo simple de máximo 3), y `Biblioteca` con métodos `prestar(Libro, Usuario)` y `devolver(Libro, Usuario)` que actualicen la disponibilidad y la lista del usuario. Piensa que validaciones necesita cada método (libro disponible? usuario con espacio?).

Resumen de la unidad

- POO = datos + comportamientos juntos en objetos; la clase es el molde, el objeto la instancia.
- `new Clase()` crea el objeto; el operador `.` accede a sus miembros; cada objeto tiene sus propios atributos.
- Las variables guardan referencias (etiquetas), no objetos: `b = a` no duplica.
- `null` = sin objeto; usarlo truena con `NullPointerException`.
- El constructor (mismo nombre que la clase, sin retorno) inicializa el objeto al crearlo; puede haber varios (sobrecarga); `this(...)` llama a otro constructor.
- `this.atributo` distingue el atributo del parámetro con igual nombre.
- Encapsulamiento: atributos `private` + getters/setters públicos con validación; las reglas del negocio viven en la clase.

- ``static`` = de la clase, compartido por todos los objetos (una sola copia); ``main`` es ``static`` por eso.
- Diseña por sustantivos (clases), características (atributos) y acciones (métodos); cada objeto cuida lo suyo.

UML básico: dibuja tus clases antes de codificar

Los profesionales dibujan sus clases en **UML** (diagramas simples) antes de escribir código. Un diagrama de clase es una cajita con 3 secciones:

```
+-----+
|      Perro      | <- nombre de la clase
+-----+
| - nombre: String | <- atributos (- = private, + = public)
| - edad: int      |
+-----+
| + Perro(n, e)    | <- constructor y metodos
| + ladrar(): void |
+-----+
```

Dibujar 5 cajitas y sus relaciones en papel te ahorra horas de reescribir código. Para este libro no necesitas UML formal, pero el hábito de "diseñar en papel primero" es de profesional.

Composicion: objetos dentro de objetos

Un objeto puede tener como atributo OTRO objeto. Esto se llama **composición** ("tiene un"):

```

public class Motor {
    private int caballos;
    public Motor(int caballos) { this.caballos = caballos; }
    public int getCaballos() { return caballos; }
}

public class Auto {
    private String marca;
    private Motor motor; // un Auto TIENE UN Motor

    public Auto(String marca, int caballos) {
        this.marca = marca;
        this.motor = new Motor(caballos); // el auto crea su motor
    }

    public void describir() {
        System.out.println(marca + " con motor de " + motor.getCaballos() + " HP");
    }
}

```

El `Auto` no sabe de caballos: le delega al `Motor`. Cada objeto se especializa. Los sistemas reales son redes de objetos componiéndose: una `Tienda` tiene `Producto`s, un `Pedido` tiene `Cliente` y lista de `Producto`s. La composición es la relación más usada en POO (más que la herencia, que verás en la Unidad 7).

Inmutabilidad: objetos que no cambian

Un objeto **immutable** no permite modificar sus atributos tras crearlo (sin setters, atributos `final`):

```

public class Coordenada {
    private final double x;
    private final double y;

    public Coordenada(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public double getX() { return x; }
    public double getY() { return y; }
    // sin setters: nadie la modifica jamas
}

```

`String` es immutable en Java (por eso es segura para compartir). Los inmutables son ideales para datos que no deben cambiar: configuraciones, coordenadas, fechas. Si necesitas "cambiar", creas un objeto nuevo. Menos bugs, código más predecible.

Mini glosario de la unidad

- **POO:** paradigma que organiza el software en objetos con datos y comportamientos.
- **Clase:** molde que define atributos y métodos; no guarda datos propios.
- **Objeto/instancia:** ejemplar concreto creado con `new` a partir de una clase.
- **Atributo:** variable de instancia; cada objeto tiene su propia copia.
- **Constructor:** método especial (nombre = clase, sin retorno) que inicializa al crear.
- **Encapsulamiento:** ocultar atributos (`private`) y exponer acceso controlado.
- **Getter/setter:** métodos `getX`/`setX` para leer/escribir atributos privados.
- **this:** referencia al objeto actual dentro de sus métodos.
- **static:** miembro de la clase (una copia compartida), no del objeto.
- **Referencia:** variable que apunta a un objeto (no lo contiene).
- **Composicion:** un objeto contiene a otro como atributo ("tiene un").
- **Immutable:** objeto cuyos atributos no pueden cambiar tras su creación.

Caso de estudio: cuenta bancaria

Una cuenta bancaria es el ejemplo de POO que nunca falla: tiene reglas de negocio reales (no sobregirar, no depositar negativo, llevar historial) que el encapsulamiento protege.

```

import java.util.ArrayList;

public class CuentaBancaria {
    private String titular;
    private String numeroCuenta;
    private double saldo;
    private ArrayList<String> movimientos;

    public CuentaBancaria(String titular, String numeroCuenta, double saldoInicial) {
        this.titular = titular;
        this.numeroCuenta = numeroCuenta;
        this.saldo = 0;
        this.movimientos = new ArrayList<>();
        depositar(saldoInicial); // reutiliza la validacion
    }

    public double getSaldo() { return saldo; }
    public String getTitular() { return titular; }
    public String getNumeroCuenta() { return numeroCuenta; }

    public boolean depositar(double cantidad) {
        if (cantidad <= 0) {
            System.out.println("El deposito debe ser positivo.");
            return false;
        }
        saldo += cantidad;
        movimientos.add("Deposito: +" + cantidad);
        return true;
    }

    public boolean retirar(double cantidad) {
        if (cantidad <= 0) {
            System.out.println("El retiro debe ser positivo.");
            return false;
        }
        if (cantidad > saldo) {
            System.out.println("Fondos insuficientes. Saldo: $" + saldo);
            return false;
        }
        saldo -= cantidad;
        movimientos.add("Retiro: -$" + cantidad);
        return true;
    }

    public void mostrarEstado() {

```

```

        System.out.println("\n--- Cuenta " + numeroCuenta + " ---");
        System.out.println("Titular: " + titular);
        System.out.println("Saldo: $" + saldo);
        System.out.println("Movimientos:");
        for (String m : movimientos) {
            System.out.println("  " + m);
        }
    }

    public static void main(String[] args) {
        CuentaBancaria cuenta = new CuentaBancaria("Ana Lopez", "001-123", 1000);
        cuenta.depositar(500);
        cuenta.retirar(200);
        cuenta.retirar(5000); // debe fallar: sin fondos
        cuenta.depositar(-100); // debe fallar: negativo
        cuenta.mostrarEstado();
    }
}

```

Que observar:

- El constructor llama a `depositar(saldoInicial)` en vez de asignar directo: así el saldo inicial pasa por la misma validación. Cero código duplicado.
- Los métodos devuelven `boolean`: el que llama sabe si la operación funcionó. En una app real, la interfaz mostraria el mensaje según ese valor.
- El `saldo` jamás se toca desde fuera: no existe `setSaldo`. Las unicas formas de cambiarlo son `depositar` y `retirar`, que aplican las reglas. Así es imposible dejar la cuenta en estado invalido... a menos que el propio método tenga un bug, que es mucho más fácil de encontrar en un solo lugar.
- El historial (`movimientos`) se construye solo: cada operación valida se registra. Auditoria gratis.

Reto: agrega `transferir(CuentaBancaria destino, double cantidad)`: retira de esta cuenta y deposita en la destino, registrando el movimiento en ambas. Quépasa si el retiro falla? (Pista: no debes depositar nada.)

Preguntas frecuentes de la unidad

P: Si todo es `private`, como pruebo mi clase? No puedo ver nada desde fuera. R: Para eso son los getters: te dejan VER sin dejar MODIFICAR libremente. Y los setters validan. Probar = crear objetos y llamar sus métodos publicos, no espiar sus atributos.

P: Mi constructor con validación deja el objeto en estado raro si los datos son invalidos.

Está bien? R: Funciona, pero lo profesional es avisar: lanza `IllegalArgumentException` con un mensaje claro en vez de dejar valores por defecto silenciosos. Así el error se detecta donde nace, no 200 líneas después.

P: Cuantos constructores puede tener una clase? R: Los que quieras, mientras difieran en parámetros (sobrecarga). Truco: el constructor con menos parámetros puede llamar al más completo con `this(...)` y valores por defecto, evitando duplicar código.

P: `this` es obligatorio? R: Solo cuando hay ambigüedad (parámetro con el mismo nombre que el atributo). Muchos programadores lo usan siempre por claridad; otros solo cuando hace falta. Elige un estilo y se consistente.

P: POO se me hace mucho código para cosas simples. No es más fácil todo en el `main`? R: Para 50 líneas, sí. Para 5,000, el `main` gigante es un pantano imposible de mantener. La POO paga su costó cuando el programa crece: cada clase es una pieza que entiendes, pruebas y cambias por separado. Los programas reales crecen; entrena el músculo ahora.

UNIDAD 7: Hereda, adapta y contrata - POO

parte 2

Introducción

En la Unidad 6 aprendiste a crear objetos autosuficientes. Ahora viene lo que hace a la POO realmente poderosa: las relaciones entre clases. Con la **herencia**, una clase hija reutiliza todo lo de su clase padre y lo extiende. Con el **polimorfismo**, un mismo método se comporta distinto según el objeto. Con las **clases abstractas** e **interfaces**, defines contratos que otros deben cumplir.

Estos cuatro conceptos son el corazón de Java profesional. Todo framework que uses (Spring, Android, JavaFX) está construido sobre herencia y polimorfismo. Las entrevistas de trabajo para programador Java SIEMPRE preguntan estos temas. Y lo mejor: ya tienes la base de la Unidad 6, así que estos conceptos van a encajar como piezas de rompecabezas.

Analogía general: piensa en una empresa. El **padre** es el "manual del empleado" con reglas generales (herencia: todos los empleados heredan horario, vacaciones, código de vestimenta). Cada **departamento** adapta esas reglas a su realidad (polimorfismo: "horario flexible" significa algo distinto en ventas que en fábrica). Los **puestos** definen requisitos mínimos sin decir cómo cumplirlos (clases abstractas: "todo gerente debe saber liderar", cada gerente lidera a su manera). Y las **certificaciones** son contratos externos que cualquiera puede firmar (interfaces: "certificado en primeros auxilios", lo puede tener un gerente, un vendedor o un almacenista, sin importar su jerarquía).

Explicación detallada

Herencia: no reinventes la rueda

La **herencia** permite que una clase (**hija** o subclase) obtenga automáticamente los atributos y métodos de otra (**padre** o superclase), y luego agregue o modifique lo suyo. En Java se declara con `extends`:

```

public class Animal {
    protected String nombre;
    protected int edad;

    public Animal(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }

    public void comer() {
        System.out.println(nombre + " esta comiendo");
    }

    public void hacerSonido() {
        System.out.println(nombre + " hace un sonido generico");
    }
}

public class Perro extends Animal {
    public Perro(String nombre, int edad) {
        super(nombre, edad); // llama al constructor del padre
    }

    public void moverCola() {
        System.out.println(nombre + " mueve la cola feliz");
    }
}

public class Gato extends Animal {
    public Gato(String nombre, int edad) {
        super(nombre, edad);
    }

    public void ronronear() {
        System.out.println(nombre + " ronronea: prrrr");
    }
}

```

Uso:

```

Perro firulais = new Perro("Firulais", 3);
firulais.comer();           // heredado de Animal: "Firulais esta comiendo"
firulais.hacerSonido();     // heredado (version generica por ahora)
firulais.moverCola();       // propio de Perro

Gato misifu = new Gato("Misifu", 2);
misifu.comer();             // heredado tambien
misifu.ronronear();         // propio de Gato
// misifu.moverCola();      // ERROR: los gatos no mueven la cola (no lo heredan de
                             // Perro)

```

Que ganó con esto? Escribi ``nombre``, ``edad``, ``comer()`` UNA vez en ``Animal``, y ``Perro`` y ``Gato`` los reutilizan. Si mañana agregé ``dormir()`` a ``Animal``, automáticamente perros y gatos saben dormir. Sin herencia, copiaría ese código en cada clase: duplicación, el enemigo número uno.

``super``: dentro de la hija, ``super`` significa "mi padre". ``super(nombre, edad)`` llama al constructor del padre (debe ser la primera línea del constructor hijo). También puedes usar ``super.metodo()`` para llamar explícitamente a la versión del padre.

``protected``: nuevo nivel de acceso. ``private`` = solo mi clase; ``protected`` = mi clase + mis hijas (+ mismo paquete). Los atributos que las hijas necesitan usar directamente van en ``protected``; si solo el padre los toca, ``private`` con getters. En el ejemplo use ``protected`` para que ``Perro`` pueda usar ``nombre`` directamente.

La regla de oro: "es un"

La herencia modela relaciones **"es un"**: un Perro ES UN Animal, un Gato ES UN Animal. Si la frase no tiene sentido ("un Perro es un Motor"), la herencia está mal usada: probablemente necesitas composición ("un Auto TIENE UN Motor", Unidad 6).

Jerarquias reales: ``Empleado`` -> ``Gerente``, ``Vendedor``, ``Programador``. ``Vehiculo`` -> ``Auto``, ``Moto``, ``Camion``. ``CuentaBancaria`` -> ``CuentaAhorros``, ``CuentaCorriente``. Cuando veas un grupo de clases que comparten datos y comportamientos, busca al padre comun.

Java solo permite herencia simple: una clase tiene UN solo padre directo (``extends`` una sola clase). No puedes hacer ``class X extends A, B``. (Para "tener varias habilidades", están las interfaces, que si permiten múltiples.) Esta limitación evita el "problema del diamante" de otros lenguajes y mantiene las jerarquias entendibles.

Todas las clases heredan de ``Object``: si no pones ``extends``, tu clase hereda implícitamente de ``java.lang.Object``, la madre de todas. Por eso todo objeto tiene métodos como ``toString()``,

`equals()` y `hashCode()` (aunque sean versiones genericas). Cuando haces `System.out.println(miObjeto)`, Java llama a su `toString()`. Si no lo sobrescribes, imprime algo feo como `Perro@1a2b3c`. Consejo profesional: sobrescribe `toString()` en tus clases para que impriman algo útil:

```
@Override
public String toString() {
    return "Perro{nombre='" + nombre + "', edad=" + edad + "}";
}
```

Sobrescritura: la hija mejora lo del padre

La hija puede **redefinir** un método del padre para darle su propio comportamiento. Se llama **sobrescritura** (overriding) y se marca con `@Override`:

```
public class Perro extends Animal {
    public Perro(String nombre, int edad) {
        super(nombre, edad);
    }

    @Override
    public void hacerSonido() {
        System.out.println(nombre + " dice: Guau guau!");
    }
}

public class Gato extends Animal {
    public Gato(String nombre, int edad) {
        super(nombre, edad);
    }

    @Override
    public void hacerSonido() {
        System.out.println(nombre + " dice: Miau!");
    }
}
```

Ahora `firula.is.hacerSonido()` ladra y `misifu.hacerSonido()` maulla. Mismo nombre de método, comportamiento adaptado a cada hija.

`@Override` es una **anotación** que le dice al compilador "estoy sobrescribiendo a proposito". Si escribes mal el nombre o los parámetros, el compilador te avisa en vez de crear un método

nuevo por accidente. **Usa `@Override` siempre** al sobrescribir: es gratis y te salva de bugs silenciosos.

Reglas de la sobrescritura:

- Mismo nombre, mismos parámetros, mismo (o compatible) tipo de retorno.
- No puedes reducir la visibilidad: si el padre es `public`, la hija no puede hacerlo `protected` o `private`.
- Puedes llamar a la versión del padre con `super.hacerSonido()` si quieres extender en vez de reemplazar:

```
@Override
public void comer() {
    super.comer(); // primero lo del padre
    System.out.println(nombre + " mueve la cola mientras come"); // luego lo
    extra
}
```

Sobrecarga vs sobrescritura (pregunta clásica de entrevista):

| | Sobrecarga (overload) | Sobrescritura (override) | |---|---|---| | Donde | Misma clase | Hija redefine método del padre | | Firma | Mismo nombre, DISTINTOS parámetros | MISMO nombre y parámetros | | Propósito | Varias formas de llamar | Adaptar comportamiento heredado | | Se decide | En compilación | En ejecución (polimorfismo) |

Polimorfismo: muchas formas, un solo nombre

Polimorfismo significa "muchas formas". En Java: una variable del tipo padre puede apuntar a objetos de cualquier hija, y al llamar un método sobrescrito, se ejecuta la versión de la hija (no la del padre). La decisión se toma **en tiempo de ejecución** según el objeto real.

```
Animal miMascota = new Perro("Firulais", 3);
miMascota.hacerSonido(); // "Firulais dice: Guau guau!" (version PERRO)

miMascota = new Gato("Misifu", 2);
miMascota.hacerSonido(); // "Misifu dice: Miau!" (version GATO)
```

La variable es `Animal`, pero el comportamiento es el del objeto real. Java mira el objeto, no la etiqueta. Esto es revolucionario porque permite escribir código que trabaja con el padre y funciona con CUALQUIER hija, incluso hijas que se inventen en el futuro:

```

public class Veterinaria {
    // Atiende a CUALQUIER animal: perros, gatos, y los que vengan
    public static void atender(Animal a) {
        System.out.println("Atendiendo...");
        a.comer();
        a.hacerSonido(); // cada animal hace SU sonido
    }

    public static void main(String[] args) {
        atender(new Perro("Firulais", 3));
        atender(new Gato("Misifu", 2));
        // Si mañana creas clase Pajaro extends Animal,
        // atender() funciona sin cambiar ni una línea.
    }
}

```

Este es el superpoder del polimorfismo: **código abierto a extensión, cerrado a modificación**. Escribes `atender(Animal)` una vez y sirve para siempre. Los frameworks viven de esto: Spring te pide objetos de ciertos tipos padre y tu les das tus hijas personalizadas.

Arreglos polimorficos: puedes guardar hijas distintas en un arreglo del padre:

```

Animal[] animales = new Animal[3];
animales[0] = new Perro("Firulais", 3);
animales[1] = new Gato("Misifu", 2);
animales[2] = new Perro("Rex", 5);

for (Animal a : animales) {
    a.hacerSonido(); // cada uno suena según su clase real
}

```

Limitación: con una variable `Animal` solo puedes llamar métodos que existan en `Animal`. `miMascota.moverCola()` no compila aunque el objeto sea un Perro, porque `Animal` no tiene ese método. Si necesitas lo específico de la hija, haz **casting**:

```

Animal a = new Perro("Firulais", 3);
if (a instanceof Perro) {
    Perro p = (Perro) a; // casting: "se que eres un Perro"
    p.moverCola(); // ahora sí
}

```

``instanceof`` pregunta "eres de esta clase?" antes del casting, evitando ``ClassCastException``. El casting excesivo es señal de mal diseño (si necesitas preguntar el tipo a cada rato, probablemente el método debería estar en el padre, sobrescrito por cada hija).

Clases abstractas: moldes incompletos a proposito

A veces el padre es una **idea**, no algo concreto. "Animal" como tal no existe en la realidad: existen perros y gatos. Una **clase abstracta** es un padre que NO se puede instanciar (``new Animal()`` prohibido), diseñado para ser extendido. Puede tener métodos normales (con código) y métodos **abstractos** (solo la firma, sin código: las hijas DEBEN implementarlos).

```
public abstract class Figura {
    protected String color;

    public Figura(String color) {
        this.color = color;
    }

    // Metodo normal: todas las hijas lo heredan tal cual
    public String getColor() {
        return color;
    }

    // Metodos abstractos: cada hija LOS DEBE implementar
    public abstract double area();
    public abstract double perimetro();
}

public class Circulo extends Figura {
    private double radio;

    public Circulo(String color, double radio) {
        super(color);
        this.radio = radio;
    }

    @Override
    public double area() {
        return Math.PI * radio * radio;
    }

    @Override
    public double perimetro() {
        return 2 * Math.PI * radio;
    }
}
```

```

    }
}

public class RectanguloF extends Figura {
    private double base;
    private double altura;

    public RectanguloF(String color, double base, double altura) {
        super(color);
        this.base = base;
        this.altura = altura;
    }

    @Override
    public double area() {
        return base * altura;
    }

    @Override
    public double perimetro() {
        return 2 * (base + altura);
    }
}

```

Uso polimorfico:

```

Figura[] figuras = new Figura[2];
figuras[0] = new Circulo("rojo", 5);
figuras[1] = new RectanguloF("azul", 4, 6);

for (Figura f : figuras) {
    System.out.println("Area: " + f.area() + ", Perimetro: " + f.perimetro());
}
// Figura f = new Figura("verde"); // ERROR: no se puede instanciar una abstracta

```

La clase abstracta dice: "toda figura TIENE area y perimetro, pero cada una los calcula a su manera; yo no se como". **Obliga a las hijas a completar el diseno.** Si una hija no implementa todos los abstractos, ella también debe declararse abstracta.

Cuando usar clase abstracta: cuando las hijas comparten código real (atributos, métodos concretos, constructor) ADEMÁS de un contrato. `Figura` comparte `color` y `getColor()`, y exige `area()`.

Interfaces: contratos puros de comportamiento

Una **interfaz** es un contrato 100% abstracto: declara QUE métodos debe tener quien la firme, sin decir COMO (todos sus métodos son implícitamente abstractos y públicos). Una clase la implementa con `implements`, y **una clase puede implementar MÚLTIPLES interfaces** (aquí esta la "herencia múltiple" de Java, pero solo de comportamiento, no de código).

```
public interface Volador {
    void volar(); // implícitamente public abstract
    int ALTURA_MAXIMA = 10000; // las "variables" en interfaces son constantes
}

public interface Nadador {
    void nadar();
}

public class Pato implements Volador, Nadador {
    private String nombre;

    public Pato(String nombre) {
        this.nombre = nombre;
    }

    @Override
    public void volar() {
        System.out.println(nombre + " vuela aleteando");
    }

    @Override
    public void nadar() {
        System.out.println(nombre + " nada en el lago");
    }
}

public class Avion implements Volador {
    @Override
    public void volar() {
        System.out.println("El avion vuela con motores a reaccion");
    }
}
```

El `Pato` firma DOS contratos: vuela Y nada. El `Avion` solo vuela. Y nota lo hermoso: `Pato` podría además extender de `Ave` (una clase). **Una clase extiende UNA clase y implementa MUCHAS interfaces**: `class Pato extends Ave implements Volador, Nadador`.

Polimorfismo con interfaces:

```
Volador[] voladores = new Volador[2];
voladores[0] = new Pato("Lucas");
voladores[1] = new Avion();
for (Volador v : voladores) {
    v.volar(); // cada uno vuela a su manera
}
```

Interface vs clase abstracta: la decisión

		Clase abstracta		Interfaz		--- --- ---		Instanciar		No		No			Métodos con código		Si		Solo
<code>`default`/`static`</code> (Java 8+)																			
		Atributos		Si (cualquiera)		Solo constantes			Herencia										
<code>`extends`</code> (una sola) <code>`implements`</code> (múltiples)																			
		Relacion		"es un" fuerte (familia)		"puede hacer" (habilidad)			Constructor		Si		No						

Regla práctica:

- Usa **clase abstracta** cuando las clases forman una familia con código compartido: ``Animal`` -> ``Perro``, ``Gato`` (comparten nombre, edad, comer()).
- Usa **interfaz** cuando defines una habilidad que cruza familias: ``Volador`` lo firman ``Pato`` (animal) y ``Avion`` (máquina), que no comparten padre.

En el mundo real verás interfaces por todos lados: ``List``, ``Map`` (Unidad 8) son interfaces; ``ArrayList`` y ``HashMap`` son las implementaciones. Programar contra interfaces (declarar ``List<String> lista`` en vez de ``ArrayList<String> lista``) te permite cambiar la implementación sin romper nada.

Métodos ``default`` en interfaces (Java 8+): una interfaz puede dar una implementación por defecto que las clases heredan tal cual (o sobrescriben):

```
public interface Saludador {
    default void saludar() {
        System.out.println("Hola!");
    }
}
```

Esto permitio evolucionar interfaces existentes sin romper a quienes ya las implementaban. Detalle avanzado; ya lo conoces.

Errores comunes de la unidad

1. **Instanciar una clase abstracta o interfaz:** ``new Figura()``` no compila. Solo se instancian las hijas concretas.
2. **Olvidar implementar un método abstracto:** la hija debe implementarlos TODOS o declararse abstracta también.
3. **Sobrescritura con firma distinta:** si cambias un parámetro por accidente, no sobreescribes: sobrecargas. El ``@Override``` te protege.
4. **Reducir visibilidad al sobreescribir:** ``public``` en el padre exige ``public``` en la hija.
5. **Herencia donde va composición:** "Auto extiende Motor" es absurdo; "Auto tiene un Motor" es correcto. Prueba del "es un".
6. **Jerarquias profundas:** Abuelo -> Padre -> Hijo -> Nieto -> Bisnieto es una pesadilla de entender. 2-3 niveles bastan en la práctica.
7. **Casting sin ``instanceof```:** truena con ``ClassCastException``` si te equivocas de tipo.
8. **Llamar métodos de la hija con variable del padre:** no compila; el tipo de la variable manda en compilación.

Ejemplos paso a paso

Ejemplo 1: Jerarquía de empleados con polimorfismo

```
public abstract class Empleado {
    protected String nombre;
    protected double salarioBase;

    public Empleado(String nombre, double salarioBase) {
        this.nombre = nombre;
        this.salarioBase = salarioBase;
    }

    public String getNombre() {
        return nombre;
    }

    // Cada tipo de empleado calcula su pago distinto
    public abstract double calcularPago();

    public void mostrarInfo() {
        System.out.println(nombre + " - Pago: $" + calcularPago());
    }
}
```

```

    }
}

public class EmpleadoTiempoCompleto extends Empleado {
    public EmpleadoTiempoCompleto(String nombre, double salarioBase) {
        super(nombre, salarioBase);
    }

    @Override
    public double calcularPago() {
        return salarioBase; // cobra completo
    }
}

public class EmpleadoPorHoras extends Empleado {
    private int horasTrabajadas;
    private double pagoPorHora;

    public EmpleadoPorHoras(String nombre, double pagoPorHora, int horas) {
        super(nombre, 0);
        this.pagoPorHora = pagoPorHora;
        this.horasTrabajadas = horas;
    }

    @Override
    public double calcularPago() {
        return pagoPorHora * horasTrabajadas;
    }
}

public class Nomina {
    public static void main(String[] args) {
        Empleado[] personal = new Empleado[3];
        personal[0] = new EmpleadoTiempoCompleto("Ana", 15000);
        personal[1] = new EmpleadoPorHoras("Luis", 120, 160);
        personal[2] = new EmpleadoTiempoCompleto("Maria", 18000);

        double total = 0;
        for (Empleado e : personal) {
            e.mostrarInfo(); // polimorfismo: cada uno calcula a su manera
            total += e.calcularPago();
        }
        System.out.println("Total nomina: $" + total);
    }
}

```

La nomina no sabe ni le importa que tipo de empleado es cada uno: confía en el contrato ``calcularPago()``. Si mañana agregas ``EmpleadoComision``, la nomina sigue funcionando intacta.

Ejemplo 2: Interfaces en un sistema de notificaciones

```
public interface Notificable {
    void enviar(String mensaje);
}

public class EmailNotificador implements Notificable {
    private String direccion;

    public EmailNotificador(String direccion) {
        this.direccion = direccion;
    }

    @Override
    public void enviar(String mensaje) {
        System.out.println("Email a " + direccion + ": " + mensaje);
    }
}

public class SMSNotificador implements Notificable {
    private String telefono;

    public SMSNotificador(String telefono) {
        this.telefono = telefono;
    }

    @Override
    public void enviar(String mensaje) {
        System.out.println("SMS al " + telefono + ": " + mensaje);
    }
}

public class Sistema {
    public static void notificarATodos(Notificable[] canales, String mensaje) {
        for (Notificable n : canales) {
            n.enviar(mensaje);
        }
    }

    public static void main(String[] args) {
        Notificable[] canales = new Notificable[2];
    }
}
```

```
    canales[0] = new EmailNotificador("ana@empresa.com");  
    canales[1] = new SMSNotificador("555-1234");  
    notificarATodos(canales, "Tu pedido fue enviado");  
    // Manana agregas WhatsAppNotificador sin tocar notificarATodos  
  }  
}
```

Ejemplo 3: Herencia + interfaz combinadas

```

public abstract class Vehiculo {
    protected String marca;
    protected int velocidadMax;

    public Vehiculo(String marca, int velocidadMax) {
        this.marca = marca;
        this.velocidadMax = velocidadMax;
    }

    public abstract void acelerar();

    public void mostrar() {
        System.out.println(marca + " (max " + velocidadMax + " km/h)");
    }
}

public interface Electrico {
    void cargarBateria();
}

public class AutoElectrico extends Vehiculo implements Electrico {
    private int nivelBateria;

    public AutoElectrico(String marca, int velocidadMax) {
        super(marca, velocidadMax);
        this.nivelBateria = 100;
    }

    @Override
    public void acelerar() {
        System.out.println(marca + " acelera en silencio electrico");
    }

    @Override
    public void cargarBateria() {
        nivelBateria = 100;
        System.out.println(marca + " bateria al 100%");
    }
}

```

`AutoElectrico` ES UN `Vehiculo` (herencia) y PUEDE `Electrico` (habilidad por interfaz). Las dos relaciones conviviendo, cada una en su rol.

Práctica

Ejercicio 1. Crea la jerarquía ``Instrumento`` (abstracta, método abstracto ``tocar()``) con hijas ``Guitarra``, ``Piano`` y ``Trompeta``. Cada una implementa ``tocar()`` con su sonido. En el ``main``, crea un arreglo polimorfo y hazlos tocar a todos.

Ejercicio 2. Crea la interfaz ``Dibujable`` con método ``dibujar()``. Implementala en ``CirculoD`` y ``CuadradoD`` (clases simples con atributos). Luego un método ``dibujarTodo(Dibujable[] elementos)`` que los dibuje a todos sin saber sus tipos.

Ejercicio 3. Extiende la jerarquía ``Animal``/``Perro``/``Gato``: agrega la clase ``Pajaro`` extends `Animal` implements `Volador`` (usa la interfaz ``Volador`` de esta unidad). ``Pajaro`` debe hacer sonido propio ("Pio pio!") y volar. Prueba el polimorfismo con un arreglo de ``Animal`` y otro de ``Volador``.

Ejercicio 4. Crea la clase abstracta ``CuentaBancaria`` con atributo ``saldo`` protegido, métodos concretos ``depositar`` y ``getSaldo``, y método abstracto ``retirar(double monto)``. Hijas: ``CuentaAhorros`` (no permite sobregiro) y ``CuentaCorriente`` (permite sobregiro hasta -5000). Prueba retiros que pasen y que fallen en cada tipo.

Solución (esqueleto):

```
public abstract class CuentaBancaria {
    protected double saldo;

    public CuentaBancaria(double saldoInicial) {
        this.saldo = saldoInicial;
    }

    public void depositar(double monto) {
        if (monto > 0) saldo += monto;
    }

    public double getSaldo() {
        return saldo;
    }

    public abstract boolean retirar(double monto);
}

public class CuentaAhorros extends CuentaBancaria {
    public CuentaAhorros(double saldoInicial) {
        super(saldoInicial);
    }
}
```

```

        @Override
        public boolean retirar(double monto) {
            if (monto > 0 && monto <= saldo) {
                saldo -= monto;
                return true;
            }
            return false;
        }
    }

    public class CuentaCorriente extends CuentaBancaria {
        private static final double SOBREGIRO_MAX = 5000;

        public CuentaCorriente(double saldoInicial) {
            super(saldoInicial);
        }

        @Override
        public boolean retirar(double monto) {
            if (monto > 0 && (saldo - monto) >= -SOBREGIRO_MAX) {
                saldo -= monto;
                return true;
            }
            return false;
        }
    }
}

```

Ejercicio 5. Define la interfaz `Pagable` con método `double montoAPagar()`. Implementala en `Factura` (subtotal + IVA 16%) y en `NominaEmpleado` (horas * tarifa). Luego un método que reciba `Pagable[]` y sume el total a pagar. Observa como dos clases sin relación familiar comparten un contrato.

Ejercicio 6 (reto). Disena un mini sistema de figuras geometricas profesional: interfaz `Medible` (métodos `area()`, `perimetro()`), clase abstracta `Figura` (color + `getColor()`), implementa `Medible` dejando area/perimetro abstractos), e hijas `Circulo`, `RectanguloG` y `TrianguloRectangulo`. Agrega un método `figuraMayorArea(Medible[] figuras)` que devuelva la de mayor area. Este ejercicio combina TODO lo de la unidad.

Resumen de la unidad

- Herencia (`extends`): la hija reutiliza y extiende al padre; modela "es un"; Java solo permite un padre directo.

- ``super(...)`` llama al constructor del padre (primera línea); ``super.metodo()`` invoca su versión.
- ``protected`` = visible para la clase, el paquete y las hijas.
- Sobrescritura (``@Override``): la hija redefine un método del padre con la misma firma; usala siempre con la anotación.
- Sobrecarga = mismo nombre, distintos parámetros (misma clase). Sobrescritura = misma firma, otra implementación (hija).
- Polimorfismo: variable del tipo padre + objeto de la hija = se ejecuta la versión de la hija (decisión en ejecución).
- El polimorfismo permite código generico que funciona con hijas futuras sin cambios.
- ``instanceof`` + casting para acceder a lo específico de la hija (usalo poco).
- Clase abstracta: no instanciable, mezcla métodos concretos y abstractos; las hijas completan el diseno.
- Interfaz (``implements``, múltiples): contrato puro de comportamiento; modela "puede hacer".
- Abstracta = familia con código compartido; interfaz = habilidad que cruza familias.
- Toda clase hereda de ``Object``: sobreescribe ``toString()`` para depurar mejor.

El principio de sustitución de Liskov (idea avanzada, explicada simple)

Hay un principio famoso (la "L" de los principios SOLID) que dice: **una hija debe poder usarse en cualquier lugar donde se espera al padre, sin romper nada**. En cristiano: si tu ``Perro`` truena cuando alguien lo trata como ``Animal``, tu herencia está mal diseñada.

Ejemplo de violación: si ``CuentaAhorros.retirar()`` lanzara una excepción inesperada donde el padre promete devolver ``boolean``, el código polimorfico se rompe. La sobrescritura puede ESPECIALIZAR el comportamiento, pero no puede ROMPER el contrato del padre. Cuando diseñes jerarquias, preguntate: "si alguien usa mi hija como si fuera el padre, todo sigue funcionando?" Si la respuesta es sí, tu diseño es sólido.

Clases selladas (sealed): herencia con lista de invitados (Java 17+)

Java moderno permite restringir QUIENES pueden heredar de tu clase:

```
public sealed class Figura2 permits Circulo2, Rectangulo2 {  
    // solo Circulo2 y Rectangulo2 pueden extender Figura2  
}
```

Es avanzado y no lo necesitaras pronto, pero ya sabes que existe: sirve cuando tu diseño exige un conjunto cerrado de hijas (como los tipos de un sistema fiscal o un protocolo).

Mini glosario de la unidad

- **Herencia:** mecanismo por el que una clase obtiene miembros de otra (`extends`).
- **Superclase/subclase:** la clase padre / la clase hija.
- **Sobrescritura (override):** redefinir en la hija un método del padre.
- **Polimorfismo:** un mismo método con comportamientos distintos según el objeto real.
- **Clase abstracta:** clase no instanciable con métodos abstractos por implementar.
- **Método abstracto:** firma sin cuerpo; obliga a las hijas a implementarlo.
- **Interfaz:** contrato de métodos que una clase se compromete a implementar.
- **instanceof:** operador que pregunta si un objeto es de cierto tipo.
- **Casting de objetos:** convertir una referencia del padre al tipo de la hija.
- **Contrato:** acuerdo de comportamiento que una clase promete cumplir.

Laboratorio: zoológico polimorfo

Vamos a ver el polimorfismo trabajando en un programa completo: un zoológico donde cada animal hace su sonido, se alimenta distinto, y el cuidador los atiende a todos sin saber de que especie es cada uno.

```
abstract class Animal {  
    protected String nombre;  
  
    public Animal(String nombre) {  
        this.nombre = nombre;  
    }  
  
    public abstract void hacerSonido();  
    public abstract void alimentar();  
  
    public void dormir() {  
        System.out.println(nombre + " esta durmiendo...");  
    }  
}
```

```

    }
}

class Leon extends Animal {
    public Leon(String nombre) { super(nombre); }

    @Override
    public void hacerSonido() {
        System.out.println(nombre + " ruge: RRRROAR!");
    }

    @Override
    public void alimentar() {
        System.out.println(nombre + " come 5 kg de carne.");
    }
}

class Elefante extends Animal {
    public Elefante(String nombre) { super(nombre); }

    @Override
    public void hacerSonido() {
        System.out.println(nombre + " barrita: PRUUU!");
    }

    @Override
    public void alimentar() {
        System.out.println(nombre + " come 50 kg de vegetales.");
    }
}

class Pinguino extends Animal {
    public Pinguino(String nombre) { super(nombre); }

    @Override
    public void hacerSonido() {
        System.out.println(nombre + " grazna: CUAC CUAC!");
    }

    @Override
    public void alimentar() {
        System.out.println(nombre + " come 2 kg de pescado.");
    }
}

```

```

public class Zoologico {
    public static void main(String[] args) {
        Animal[] animales = {
            new Leon("Simba"),
            new Elefante("Dumbo"),
            new Pinguino("Pingu")
        };

        System.out.println("=== RUTINA DEL CUIDADOR ===");
        for (Animal a : animales) {
            a.hacerSonido(); // cada uno suena distinto
            a.alimentar();   // cada uno come distinto
            a.dormir();      // todos duermen igual (heredado)
            System.out.println();
        }
    }
}

```

Que observar:

- El cuidador (el `for`) no sabe ni le importa que animal atiende: trabaja contra la abstracción `Animal`. Mañana agregas `Jirafa` y el cuidador sigue funcionando sin cambios. Esa es la extensibilidad que las empresas pagan.
- `dormir()` es concreto y heredado: comportamiento compartido en el padre, específico en las hijas. La mezcla correcta de ambos es el diseño orientado a objetos.
- Si olvidas implementar `hacerSonido` en una hija, el compilador te lo exige: la clase abstracta es un contrato que no se puede romper por accidente.

Reto: agrega la interfaz `Volador` con método `volar()`, implementala solo en `Pinguino`... no, espera: los pinguinos no vuelan. Implementala en una nueva clase `Aguila`. Luego, en el `main`, haz que solo los animales que sean `instanceof Volador` vuelen. Piensa por qué el pinguino demuestra que la herencia debe modelar la realidad, no solo el código.

Preguntas frecuentes de la unidad

P: Herencia o composición? Como decido? R: Pregunta "es un?" vs "tiene un?". Un `Perro` ES UN `Animal` (herencia). Un `Pedido` TIENE `Producto`s (composición). Si dudas, prefiere composición: es más flexible y evita jerarquías frágiles. Los expertos dicen "favorece la composición sobre la herencia".

P: Por qué Java no permite herencia múltiple de clases? R: Por el problema del diamante: si `C` hereda de `A` y `B`, y ambas tienen método `m()`, ¿cual usa `C`? C++, que si lo permite, sufre con esto. Java lo evito; para múltiples comportamientos usa interfaces.

P: `@Override` es obligatorio? Mi código compila sin el. R: Compila, pero pierdes la red de seguridad: si escribes mal la firma, creas un método nuevo en vez de sobrescribir, y el bug es silencioso. Usala siempre: cuesta 9 caracteres y te salva horas.

P: Puedo sobrescribir un método `static`? R: No realmente: se "oculta", no se sobrescribe, y no hay polimorfismo (se decide por el tipo de la variable, no del objeto). Los métodos `static` no participan en la herencia dinámica. Si necesitas polimorfismo, el método debe ser de instancia.

P: Una interfaz con un solo método: para que sirve? R: Mucho: son las interfaces funcionales (ej. `Runnable` con `run()`), base de las lambdas y la programación funcional en Java. Un solo método = un solo proposito = maxima flexibilidad para quien la implementa.

UNIDAD 8: Colecciones que crecen solas

Introducción

Los arreglos de la Unidad 5 tienen un defecto fatal: su tamaño es fijo. Si creas un arreglo de 10 y llegan 11 datos, tienes que crear otro más grande y copiar todo a mano. En programas reales (una tienda con productos que entran y salen, un chat con mensajes que llegan sin parar), el tamaño fijo es una camisa de fuerza.

Las **colecciones** de Java resuelven esto: son estructuras de datos que crecen y se encogen solas. Las dos que usaras todos los días son `ArrayList` (lista ordenada que crece sola) y `HashMap` (diccionario de llave -> valor con búsqueda instantánea). Viven en el paquete `java.util` y son parte del **Java Collections Framework**, una de las librerías mejor diseñadas del lenguaje.

Al terminar esta unidad manejarás listas y mapas como profesional, sabrás elegir la colección correcta para cada problema, y entenderás conceptos que aparecen en TODAS las entrevistas de Java (autoboxing, `equals`/`hashCode`, complejidad de búsqueda).

Explicación detallada

ArrayList: el arreglo que crece solo

```

import java.util.ArrayList;

public class ListaDemo {
    public static void main(String[] args) {
        ArrayList<String> nombres = new ArrayList<>();

        nombres.add("Ana");    // agrega al final
        nombres.add("Luis");
        nombres.add("Maria");

        System.out.println(nombres);    // [Ana, Luis, Maria]
        System.out.println(nombres.size());    // 3 (como length, pero metodo)
        System.out.println(nombres.get(0));    // "Ana" (como [0])
        nombres.set(1, "Pedro");    // cambia la posicion 1
        nombres.remove(0);    // quita "Ana"
        System.out.println(nombres);    // [Pedro, Maria]
    }
}

```

Operaciones esenciales:

| Método | Que hace | |---|---| | `add(valor)` | Agrega al final | | `add(pos, valor)` | Inserta en la posición dada | | `get(i)` | Obtiene el elemento i | | `set(i, valor)` | Reemplaza el elemento i | | `remove(i)` | Quita el elemento i (los demás se recorren) | | `remove(valor)` | Quita la primera ocurrencia del valor | | `size()` | Cuantos elementos hay | | `isEmpty()` | `true` si está vacía | | `contains(valor)` | `true` si el valor existe | | `clear()` | Vacía la lista | | `indexOf(valor)` | Posición del valor, o -1 |

Diferencias clave con arreglos: `size()` en vez de `length`, `get(i)`/`set(i)` en vez de `[i]`, y métodos como `add`/`remove`/`contains` que los arreglos no tienen.

Los genericos ``<>``: `ArrayList<String>` dice "lista DE Strings". Los ``<>`` se llaman **genericos** y le dicen al compilador que tipo guarda la colección. Sin genericos (`ArrayList lista = new ArrayList();`) puedes meter de todo, pero al sacar tienes que hacer casting y el compilador no te protege. **Usa genericos siempre:** `ArrayList<String>`, `ArrayList<Integer>`, `ArrayList<Producto>`.

El diamante ``<>``: en `new ArrayList<>()` puedes dejar los ``<>`` vacios: Java infiere el tipo del lado izquierdo. Menos repetición, mismo chequeo.

Autoboxing: primitivos en colecciones (el truco)

Las colecciones NO guardan primitivos (`int`, `double`...): guardan OBJETOS. Para enteros usas la clase envoltorio `Integer`:

```
ArrayList<Integer> numeros = new ArrayList<>();
numeros.add(10);           // parece un int... pero Java lo convierte solo a Integer
int x = numeros.get(0);    // ...y de regreso a int, tambien solo
```

Esta conversión automática se llama **autoboxing** (int -> Integer) y **unboxing** (Integer -> int). Java lo hace por ti, así que el código se ve natural. Los envoltorios existen para los 8 primitivos: `Integer`, `Double`, `Character`, `Boolean`, `Long`, `Float`, `Short`, `Byte`.

Cuidado con `null`: un `Integer` puede ser `null`, y al hacer unboxing de un `null` truena con `NullPointerException`:

```
Integer n = null;
int x = n;    // TRUENA: NullPointerException en el unboxing
```

Si tu lista puede tener `null`s, verifica antes de operar.

Recorrer un ArrayList

Igual que los arreglos, con for clásico o for-each:

```
ArrayList<String> nombres = new ArrayList<>();
nombres.add("Ana");
nombres.add("Luis");
nombres.add("Maria");

// Con índice (si necesitas la posición)
for (int i = 0; i < nombres.size(); i++) {
    System.out.println(i + ": " + nombres.get(i));
}

// For-each (más limpio si no necesitas el índice)
for (String nombre : nombres) {
    System.out.println(nombre);
}
```

Filtrar, buscar y transformar (patrones de todos los días)

Filtrar (quedarse con los que cumplen):

```
ArrayList<Integer> numeros = new ArrayList<>();
// ... llenar con 1..10
ArrayList<Integer> pares = new ArrayList<>();
for (int n : numeros) {
    if (n % 2 == 0) {
        pares.add(n);
    }
}
```

Buscar (encontrar el primero que cumple):

```
public static Producto buscarPorNombre(ArrayList<Producto> productos, String
nombre) {
    for (Producto p : productos) {
        if (p.getNombre().equalsIgnoreCase(nombre)) {
            return p;
        }
    }
    return null; // no encontrado
}
```

Transformar (convertir cada elemento):

```
ArrayList<String> nombres = new ArrayList<>();
// ... llenar
ArrayList<String> mayusculas = new ArrayList<>();
for (String n : nombres) {
    mayusculas.add(n.toUpperCase());
}
```

Estos tres patrones (filtrar, buscar, transformar) son el 80% del trabajo con colecciones en el mundo real. (Java 8+ los hace más elegantes con Streams; ya los conocerás, pero estos ciclos son la base que debes dominar primero.)

Ordenar un ArrayList

```
import java.util.ArrayList;
import java.util.Collections;

ArrayList<Integer> numeros = new ArrayList<>();
numeros.add(5);
numeros.add(2);
numeros.add(9);
numeros.add(1);

Collections.sort(numeros); // [1, 2, 5, 9] (menor a mayor)
Collections.reverse(numeros); // [9, 5, 2, 1]
```

`Collections.sort` ordena de menor a mayor (para Strings, orden alfabético). Para ordenar TUS objetos (como `Producto` por precio), la clase debe implementar `Comparable` o le pasas un `Comparator`:

```
// Ordenar productos por precio (de barato a caro)
productos.sort((p1, p2) -> Double.compare(p1.getPrecio(), p2.getPrecio()));
```

Esa flecha `->` es una **lambda** (función anónima corta): le dice al sort COMO comparar dos productos. Las lambdas son de Java 8+ y las verás en código profesional constantemente; por ahora basta con reconocer el patrón.

Errores comunes con ArrayList

1. **Usar `[i]` en vez de `get(i)`**: los corchetes son solo para arreglos. `lista[0]` no compila con ArrayList.
2. **`IndexOutOfBoundsException`**: `get(5)` en una lista de 3 elementos. Igual que con arreglos: el último válido es `size() - 1`.
3. **Modificar la lista mientras la recorres con for-each**: quitar elementos dentro de un for-each lanza `ConcurrentModificationException`. Solución: usa un `Iterator` explícito o recorre por índice de atrás hacia adelante.
4. **Comparar con `==` en `contains`/`remove`**: estos métodos usan `.equals()`. Para TUS clases, si no sobrescribes `equals`, comparan por referencia (dos objetos con los mismos datos se consideran distintos). Regla: si tu clase va a vivir en colecciones y la vas a buscar/comparar, sobrescribe `equals` (y `hashCode`, van juntos).
5. **Olvidar el import**: `import java.util.ArrayList;` es obligatorio.

HashMap: el diccionario superveloz

Un `HashMap` guarda pares **llave -> valor**: buscas por la llave y obtienes el valor al instante, sin importar cuantos elementos haya.

```
import java.util.HashMap;

public class MapaDemo {
    public static void main(String[] args) {
        HashMap<String, Integer> edades = new HashMap<>();

        edades.put("Ana", 25);        // guarda el par
        edades.put("Luis", 30);
        edades.put("Maria", 22);

        System.out.println(edades.get("Ana")); // 25 (busqueda por llave)
        System.out.println(edades.get("Pedro")); // null (no existe)

        edades.put("Ana", 26);        // si la llave existe, ACTUALIZA el valor
        System.out.println(edades.get("Ana")); // 26

        System.out.println(edades.containsKey("Luis")); // true
        System.out.println(edades.size()); // 3
        edades.remove("Maria");        // quita el par
    }
}
```

Operaciones esenciales:

| Método | Que hace | |---|---| | `put(llave, valor)` | Guarda o actualiza el par | | `get(llave)` | Obtiene el valor, o `null` si no existe | | `getOrDefault(llave, defecto)` | El valor, o un valor por defecto si no existe | | `containsKey(llave)` | `true` si la llave existe | | `containsValue(valor)` | `true` si el valor existe (lento: revisa todos) | | `remove(llave)` | Quita el par | | `size()`, `isEmpty()`, `clear()` | Como en las listas | | `keySet()` | Todas las llaves (para recorrer) | | `values()` | Todos los valores | | `entrySet()` | Todos los pares llave-valor |

Las llaves son únicas: no puede haber dos pares con la misma llave. `put` con una llave existente SOBREScribe el valor anterior. Los valores si pueden repetirse.

Analogía: el `HashMap` es un diccionario de verdad. La llave es la palabra que buscas ("Ana"), el valor es la definición ("25"). Buscar en el diccionario no requiere leerlo todo: vas directo a la palabra. Así de rápido es el `HashMap`: búsqueda, inserción y eliminación en tiempo casi constante, no importa si hay 10 o 10 millones de pares. (Por dentro usa una tabla hash:

convierte la llave en un número con `hashCode()` y salta directo a su casilla. Por eso `equals` y `hashCode` van juntos.)

Recorrer un HashMap:

```
// Por llaves
for (String nombre : edades.keySet()) {
    System.out.println(nombre + " tiene " + edades.get(nombre) + " anos");
}

// Por pares (mas eficiente: no busca dos veces)
for (HashMap.Entry<String, Integer> par : edades.entrySet()) {
    System.out.println(par.getKey() + " -> " + par.getValue());
}
```

El patrón contador con HashMap (vale oro)

Contar cuantas veces aparece cada cosa:

```
String[] palabras = {"hola", "mundo", "hola", "java", "mundo", "hola"};
HashMap<String, Integer> conteo = new HashMap<>();

for (String p : palabras) {
    conteo.put(p, conteo.getOrDefault(p, 0) + 1);
}
System.out.println(conteo); // {hola=3, mundo=2, java=1}
```

`getOrDefault(p, 0)` devuelve el conteo actual o 0 si es la primera vez; le sumas 1 y lo guardas. Este patrón aparece en análisis de texto, encuestas, estadísticas y entrevistas de trabajo. Grabatelo.

HashMap no tiene orden (y sus primos ordenados)

Importante: `HashMap` **no garantiza ningun orden** al recorrerlo. Si necesitas orden:

- `LinkedHashMap`: mantiene el orden de INSERCIÓN (como lo metiste).
- `TreeMap`: mantiene las llaves ORDENADAS (alfabetico/numerico).

```
import java.util.LinkedHashMap;
import java.util.TreeMap;

LinkedHashMap<String, Integer> a = new LinkedHashMap<>(); // orden de insercion
TreeMap<String, Integer> b = new TreeMap<>();           // llaves ordenadas
```

HashSet: la colección sin duplicados

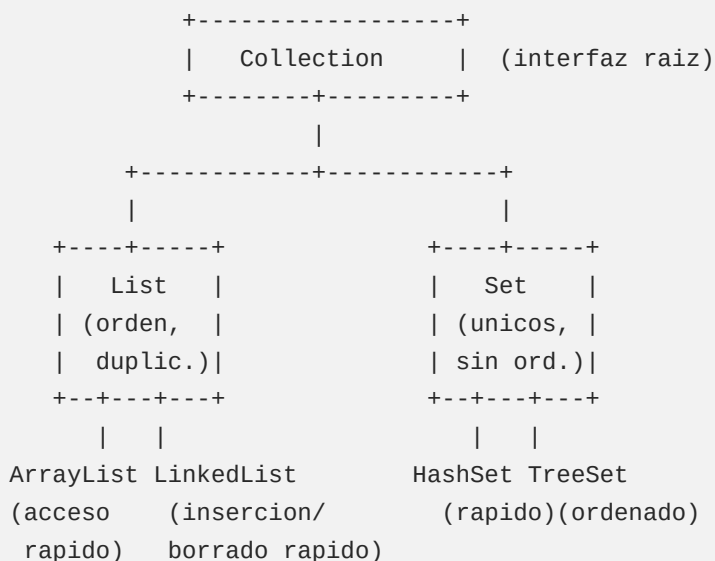
Un `HashSet` guarda valores únicos, sin orden, con búsqueda rapidísima. Perfecto para "ya vi esto antes?":

```
import java.util.HashSet;

HashSet<String> visitados = new HashSet<>();
visitados.add("ana.com");
visitados.add("blog.com");
visitados.add("ana.com"); // duplicado: se ignora
System.out.println(visitados.size()); // 2
System.out.println(visitados.contains("blog.com")); // true
```

Usos: eliminar duplicados de una lista (mete la lista en un Set), recordar que usuarios ya votaron, palabras únicas de un texto.

La familia completa: guía de elección



Y aparte: `Map` (llave-valor, no es Collection): `HashMap`, `LinkedHashMap`, `TreeMap`.

Guía rápida de elección:

| Necesito... | Uso | |---|---| | Lista ordenada, acceso por posición | `ArrayList` | | Muchas inserciones/eliminaciones en medio | `LinkedList` | | Buscar por llave rapidísimo | `HashMap` | | Llaves ordenadas | `TreeMap` | | Valores únicos sin duplicados | `HashSet` | | Contar frecuencias | `HashMap<K, Integer>` | | Tamaño fijo, máximo rendimiento | arreglo `[]` |

Programa contra interfaces: declara tus variables con la interfaz, no con la clase concreta:

```
// Bien: flexible
List<String> nombres = new ArrayList<>();
Map<String, Integer> edades = new HashMap<>();

// Funcional pero rígido:
ArrayList<String> nombres2 = new ArrayList<>();
```

Así, si mañana cambias `ArrayList` por `LinkedList`, solo cambias el `new`: todo lo demás sigue igual porque el código usa la interfaz `List`. Necesitas `import java.util.List;` y `import java.util.Map;`. Este es uno de esos hábitos que distinguen código junior de código profesional.

equals y hashCode: el contrato sagrado

Si tus objetos van a vivir en `HashSet` o ser llaves de `HashMap`, DEBES sobrescribir `equals` y `hashCode` con reglas coherentes:

- `equals`: dos objetos son iguales si sus datos importantes son iguales.
- `hashCode`: objetos iguales (`equals` = true) DEBEN tener el mismo `hashCode`.
- Si rompes esto (equals sin hashCode), el `HashMap` "pierde" tus objetos: los guarda en una casilla y los busca en otra.

```

public class Producto {
    private String codigo;
    private String nombre;
    private double precio;

    // constructor, getters...

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Producto p = (Producto) o;
        return codigo.equals(p.codigo);
    }

    @Override
    public int hashCode() {
        return codigo.hashCode();
    }
}

```

Dos productos con el mismo código son "el mismo producto" aunque difieran en precio (el precio puede cambiar; el código no). Los IDEs generan estos métodos automáticamente (igual que getters/setters). En entrevistas preguntan este contrato SIEMPRE.

Errores comunes de la unidad

1. **`get` de llave inexistente devuelve `null`**: verifica con ``containsKey`` o usa ``getOrDefault`` antes de operar con el resultado.
2. **Esperar orden en un `HashMap`**: no lo tiene. Usa ``LinkedHashMap`` o ``TreeMap`` si importa.
3. **Llaves mutables**: si usas un objeto como llave y luego lo modificas, el mapa lo "pierde". Las llaves deben ser efectivamente inmutables (Strings, Integers, o tus clases inmutables).
4. **``ConcurrentModificationException``**: quitar elementos de una lista/mapa mientras la recorres con `for-each`. Usa ``Iterator`` con su método ``remove()``, o ``removeIf``:

```

lista.removeIf(n -> n < 0); // quita todos los negativos, sin excepcion

```

5. **Usar `==` para buscar en colecciones:** `contains` y `remove(valor)` usan `equals`. Sin `equals` sobrescrito en tu clase, no encontrarán nada.
6. **Autoboxing con null:** `map.get(llave)` puede dar `null`; asignarlo a `int` truena.

Ejemplos paso a paso

Ejemplo 1: Agenda con ArrayList y objetos

```
import java.util.ArrayList;

class Contacto {
    private String nombre;
    private String telefono;

    public Contacto(String nombre, String telefono) {
        this.nombre = nombre;
        this.telefono = telefono;
    }

    public String getNombre() { return nombre; }
    public String getTelefono() { return telefono; }

    @Override
    public String toString() {
        return nombre + " - " + telefono;
    }
}

public class AgendaLista {
    public static void agregar(ArrayList<Contacto> agenda, String nombre, String tel)
    {
        agenda.add(new Contacto(nombre, tel));
    }

    public static Contacto buscar(ArrayList<Contacto> agenda, String nombre) {
        for (Contacto c : agenda) {
            if (c.getNombre().equalsIgnoreCase(nombre)) {
                return c;
            }
        }
        return null;
    }
}
```

```

public static boolean eliminar(ArrayList<Contacto> agenda, String nombre) {
    Contacto c = buscar(agenda, nombre);
    if (c != null) {
        agenda.remove(c);
        return true;
    }
    return false;
}

public static void main(String[] args) {
    ArrayList<Contacto> agenda = new ArrayList<>();
    agregar(agenda, "Ana", "555-1234");
    agregar(agenda, "Luis", "555-5678");

    Contacto e = buscar(agenda, "ana");
    System.out.println("Encontrado: " + e);

    eliminar(agenda, "Luis");
    System.out.println("Restan: " + agenda.size());
    for (Contacto c : agenda) {
        System.out.println(c);
    }
}
}

```

Es la agenda de la Unidad 5, ahora con lista dinámica (sin límite de 3) y objetos `Contacto` en vez de arreglos paralelos. Compara con la versión anterior: el código es más corto y más claro. Las colecciones + POO son la combinación ganadora.

Ejemplo 2: Conteo de palabras en un texto

```

import java.util.HashMap;
import java.util.ArrayList;
import java.util.Collections;

public class ContadorPalabras {
    public static void main(String[] args) {
        String texto = "java es genial java es poderoso java es gratis";
        String[] palabras = texto.split(" ");

        HashMap<String, Integer> conteo = new HashMap<>();
        for (String p : palabras) {
            conteo.put(p, conteo.getOrDefault(p, 0) + 1);
        }

        // Mostrar ordenado por palabra
        ArrayList<String> ordenadas = new ArrayList<>(conteo.keySet());
        Collections.sort(ordenadas);
        for (String p : ordenadas) {
            System.out.println(p + ": " + conteo.get(p));
        }
    }
}

```

`texto.split(" ")` parte el texto en palabras (devuelve un arreglo). El patrón contador hace su magia, y luego ordenamos las llaves para una salida bonita. Este programa es un mini-analizador de texto real.

Ejemplo 3: Inventario de tienda con HashMap

```

import java.util.HashMap;

class ProductoT {
    private String codigo;
    private String nombre;
    private double precio;
    private int stock;

    public ProductoT(String codigo, String nombre, double precio, int stock) {
        this.codigo = codigo;
        this.nombre = nombre;
        this.precio = precio;
        this.stock = stock;
    }
}

```

```

    public String getCodigo() { return codigo; }
    public String getNombre() { return nombre; }
    public double getPrecio() { return precio; }
    public int getStock() { return stock; }

    public boolean vender(int cantidad) {
        if (cantidad > 0 && cantidad <= stock) {
            stock -= cantidad;
            return true;
        }
        return false;
    }

    @Override
    public String toString() {
        return codigo + " | " + nombre + " | $" + precio + " | stock: " + stock;
    }
}

public class Tienda {
    public static void main(String[] args) {
        HashMap<String, ProductoT> inventario = new HashMap<>();

        inventario.put("A01", new ProductoT("A01", "Laptop", 15000, 10));
        inventario.put("A02", new ProductoT("A02", "Mouse", 300, 50));
        inventario.put("A03", new ProductoT("A03", "Teclado", 800, 30));

        // Venta: busqueda instantanea por codigo
        String codigoVenta = "A02";
        ProductoT p = inventario.get(codigoVenta);
        if (p != null && p.vender(2)) {
            System.out.println("Venta exitosa: " + p);
        }

        // Reporte completo
        double valorTotal = 0;
        for (ProductoT prod : inventario.values()) {
            System.out.println(prod);
            valorTotal += prod.getPrecio() * prod.getStock();
        }
        System.out.println("Valor del inventario: $" + valorTotal);
    }
}

```

La llave es el código del producto: buscar "A02" es instantaneo aunque haya 100,000 productos. Así funcionan los sistemas reales de inventario y es la antesala del proyecto Tienda de la Unidad 10.

Ejemplo 4: Eliminar duplicados y detectar repetidos

```
import java.util.ArrayList;
import java.util.HashSet;

public class Duplicados {
    public static void main(String[] args) {
        ArrayList<String> correos = new ArrayList<>();
        correos.add("ana@mail.com");
        correos.add("luis@mail.com");
        correos.add("ana@mail.com");
        correos.add("mia@mail.com");
        correos.add("luis@mail.com");

        HashSet<String> unicos = new HashSet<>(correos);
        System.out.println("Original: " + correos.size()); // 5
        System.out.println("Unicos: " + unicos.size());      // 3

        // Cuales se repitieron?
        HashSet<String> vistos = new HashSet<>();
        HashSet<String> repetidos = new HashSet<>();
        for (String c : correos) {
            if (!vistos.add(c)) { // add devuelve false si ya existia
                repetidos.add(c);
            }
        }
        System.out.println("Repetidos: " + repetidos);
    }
}
```

El truco `if (!vistos.add(c))`: `add` devuelve `false` si el elemento ya estaba. En una sola pasada detectas repetidos. Patrón clásico de entrevistas.

Práctica

Ejercicio 1. Crea un `ArrayList<Integer>` con los números del 1 al 20, elimina los múltiplos de 3 (cuidado con el `ConcurrentModificationException`: usa `removeIf`), e imprime la lista resultante y su tamaño.

Ejercicio 2. Escribe un programa que pida 5 nombres, los guarde en un `ArrayList<String>`, los ordene con `Collections.sort`, y los imprima numerados. Luego pregunta un nombre a buscar e indica su posición con `indexOf` (o "no existe" si da -1).

Ejercicio 3. Crea un `HashMap<String, Double>` con 5 productos y sus precios. Calcula e imprime: el precio promedio, el producto más caro (recorre `entrySet`), y aplica 10% de descuento a todos (actualiza con `put`).

Solución (partes clave):

```
import java.util.HashMap;

public class Precios {
    public static void main(String[] args) {
        HashMap<String, Double> precios = new HashMap<>();
        precios.put("Laptop", 15000.0);
        precios.put("Mouse", 300.0);
        precios.put("Teclado", 800.0);
        precios.put("Monitor", 4500.0);
        precios.put("Silla", 2500.0);

        double suma = 0;
        String masCaro = "";
        double precioMax = 0;
        for (HashMap.Entry<String, Double> e : precios.entrySet()) {
            suma += e.getValue();
            if (e.getValue() > precioMax) {
                precioMax = e.getValue();
                masCaro = e.getKey();
            }
        }
        System.out.println("Promedio: " + (suma / precios.size()));
        System.out.println("Mas caro: " + masCaro + " ($" + precioMax + ")");

        // 10% de descuento a todos
        for (String k : precios.keySet()) {
            precios.put(k, precios.get(k) * 0.9);
        }
        System.out.println("Con descuento: " + precios);
    }
}
```

Ejercicio 4. Simula una votación: un arreglo de Strings con votos ("Ana", "Luis", "Ana", "Mía", "Luis", "Ana"...). Usa el patrón contador con `HashMap` para contar votos por candidato e imprime al ganador (el de mayor conteo).

Ejercicio 5. Crea la clase `Alumno` (nombre, calificación) con `equals`/`hashCode` basados en el nombre. Guarda varios en un `HashSet<Alumno>` incluyendo un duplicado (mismo nombre): verifica que el Set lo rechace. Luego pasalos a un `ArrayList` y ordenalos por calificación de mayor a menor.

Ejercicio 6. Escribe un traductor mini: `HashMap<String, String>` espanol->ingles con 10 palabras. Pide una palabra al usuario: si existe, muestra la traducción; si no, pregunta si quiere agregarla (y la agrega). Repite hasta que escriba "salir". Es un programa interactivo completo con colecciones.

Ejercicio 7 (reto). Implementa una **cola de atención** con `ArrayList<String>`: menu con 1) formar cliente (add al final), 2) atender siguiente (quita y muestra el primero), 3) ver fila, 0) salir. Luego reflexiona: qué pasa si atiendes mucho y la lista se hace grande? (Pista: quitar del inicio recorre todo; una `LinkedList` o una `Queue` real seria mejor. Ya sabes elegir.)

Iteradores: el recorrido profesional

El for-each es azucar sintactica para un **Iterator**. Conocerlo te permite quitar elementos durante el recorrido sin excepciones:

```
import java.util.ArrayList;
import java.util.Iterator;

ArrayList<Integer> numeros = new ArrayList<>();
// ... llenar con 1..10

Iterator<Integer> it = numeros.iterator();
while (it.hasNext()) {
    int n = it.next();
    if (n % 2 == 0) {
        it.remove(); // quita de forma segura durante el recorrido
    }
}
```

`hasNext()` pregunta si hay más, `next()` avanza y devuelve, `remove()` quita el actual. Es el mecanismo correcto cuando el for-each no basta. (En la práctica moderna, `removeIf` con lambda es más corto para casos simples.)

Rendimiento: qué tan rápido es cada operación (nociones)

No necesitas memorizar notación Big-O para este nivel, pero si la intuición:

- `ArrayList.get(i)`: instantaneo (salta directo a la posición).
- `ArrayList.add` al final: rapidísimo casi siempre.
- `ArrayList.remove(0)`: lento en listas grandes (recorre todo lo demás).
- `HashMap.get/put`: rapidísimo casi siempre, sin importar el tamaño.
- Buscar un valor en `ArrayList` (`contains`): revisa uno por uno (lento en listas enormes).
- Buscar una llave en `HashMap` (`containsKey`): rapidísimo.

Regla de diseño: si vas a buscar mucho por un identificador (código, nombre de usuario, CURP), guarda en `HashMap` con ese identificador como llave. Si el orden y el recorrido secuencial es lo importante, `ArrayList`.

Wrappers útiles: más allá de guardar números

Las clases envoltorio tienen métodos estáticos valiosos:

```
Integer.parseInt("42");           // String -> int (ya lo usabas)
Double.parseDouble("3.14");        // String -> double
Integer.toString(42);              // int -> String
Integer.max(5, 9);                 // 9
Double.isNaN(0.0 / 0.0);           // true (Not a Number)
Character.isDigit('5');             // true
Character.isLetter('a');            // true
Character.toUpperCase('b');         // 'B'
Boolean.parseBoolean("true");      // true
```

`Character.isDigit` e `isLetter` son perfectos para validar contraseñas o analizar texto carácter por carácter (como en el validador de la Unidad 3, pero más elegante).

Resumen de la unidad

- Las colecciones crecen solas; las dos esenciales son `ArrayList` (lista ordenada) y `HashMap` (llave->valor).
- `ArrayList`: `add`, `get(i)`, `set(i,v)`, `remove`, `size()`, `contains`, `indexOf`. Usa genéricos `<>` siempre.
- Autoboxing convierte `int` <-> `Integer` automáticamente; cuidado con `null` en unboxing.

- Patrones: filtrar, buscar (devuelve `null` o `-1` si no hay), transformar.
- `Collections.sort` ordena; para tus clases usa `Comparator` o lambdas.
- `HashMap`: `put`, `get` (null si no existe), `getOrDefault`, `containsKey`, `keySet`/`values`/`entrySet`.
- Las llaves son unicas; `put` existente sobrescribe; sin orden garantizado (`LinkedHashMap`/`TreeMap` si importa).
- Patrón contador: `map.put(k, map.getOrDefault(k, 0) + 1)`.
- `HashSet`: valores únicos, búsqueda rápida; `add` devuelve `false` si ya existía.
- Programa contra interfaces: `List<String> l = new ArrayList<>();`.
- Si tus objetos son llaves o van a Sets: sobrescribe `equals` y `hashCode` juntos.
- No modifiques una colección dentro de un for-each (usa `Iterator` o `removeIf`).

Mini glosario de la unidad

- **Colección:** estructura de datos que agrupa objetos y crece dinamicamente.
- **Genericos:** ``<Tipo>`` que indica que guarda una colección; chequeo en compilación.
- **Autoboxing/unboxing:** conversión automática entre primitivos y envoltorios.
- **Envoltorio (wrapper):** clase objeto para un primitivo (`Integer` para `int`).
- **Llave/valor:** los dos componentes de cada entrada de un Map.
- **Tabla hash:** estructura interna del HashMap para búsqueda casi instantánea.
- **Iterator:** objeto para recorrer colecciones con control total (incluye borrado seguro).
- **Lambda:** función anónima corta (`->`) para pasar comportamiento como parámetro.

Caso de estudio: sistema de votaciones

Una votación electrónica simple: los votantes eligen candidato, el sistema cuenta votos con un `HashMap`, evita votos duplicados con un `HashSet`, y al final muestra resultados ordenados.

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Scanner;

public class Votacion {
    public static void main(String[] args) {
```

```

Scanner sc = new Scanner(System.in);
HashMap<String, Integer> votos = new HashMap<>();
HashSet<String> yaVotaron = new HashSet<>();

// Candidatos iniciales
votos.put("Ana", 0);
votos.put("Luis", 0);
votos.put("Maria", 0);

System.out.println("=== VOTACION ===");
while (true) {
    System.out.print("Tu ID de votante (o 'fin'): ");
    String id = sc.nextLine();
    if (id.equalsIgnoreCase("fin")) break;

    if (yaVotaron.contains(id)) {
        System.out.println("Este ID ya voto. Un votante, un voto.");
        continue;
    }

    System.out.print("Candidato (Ana/Luis/Maria): ");
    String candidato = sc.nextLine();

    if (!votos.containsKey(candidato)) {
        System.out.println("Candidato invalido.");
        continue;
    }

    votos.put(candidato, votos.get(candidato) + 1);
    yaVotaron.add(id);
    System.out.println("Voto registrado para " + candidato + ".");
}

// Resultados ordenados de mayor a menor
System.out.println("\n=== RESULTADOS ===");
ArrayList<String> candidatos = new ArrayList<>(votos.keySet());
Collections.sort(candidatos, (a, b) -> votos.get(b) - votos.get(a));

int total = 0;
for (String c : candidatos) {
    int v = votos.get(c);
    total += v;
    System.out.println(c + ": " + v + " votos");
}
System.out.println("Total: " + total + " votos validos.");

```

```

        System.out.println("Ganador: " + candidatos.get(0));
        sc.close();
    }
}

```

Que observar:

- Tres colecciones, tres propósitos: `HashMap` para contar (candidato -> votos), `HashSet` para membresía rápida (quien ya voto), `ArrayList` para ordenar los resultados.
- `yaVotaron.contains(id)` es $O(1)$: verificar duplicados en un `HashSet` es instantáneo aunque haya millones de votantes. Con un `ArrayList` sería una búsqueda lineal lenta.
- El `continue` salta al siguiente votante cuando algo falla: el ciclo principal se mantiene limpio, sin pirámides de `if` anidados.
- El ordenamiento con `Collections.sort` y un comparador lambda `(a, b) -> votos.get(b) - votos.get(a)` ordena por votos descendente: es tu primer vistazo a las lambdas, que dominaras cuando veas programación funcional.

Reto: permite votos en blanco (candidato "Nulo") y calcula el porcentaje de cada candidato sobre el total. Muestra una barra simple con asteriscos proporcional al porcentaje.

Preguntas frecuentes de la unidad

P: `ArrayList` o arreglo normal? R: `ArrayList` casi siempre: crece solo, tiene métodos útiles y se integra con el resto de Java. El arreglo solo cuando el tamaño es fijo de por vida o necesitas el máximo rendimiento en código crítico.

P: Por qué no puedo hacer `ArrayList<int>`? R: Los generics no aceptan primitivos; usa el wrapper: `ArrayList<Integer>`. El autoboxing convierte solo entre `int` e `Integer` automáticamente.

P: `HashMap` no me devuelve los elementos en orden. Esta roto? R: No: `HashMap` no garantiza ningún orden. Si necesitas orden de inserción usa `LinkedHashMap`; si necesitas llaves ordenadas usa `TreeMap`. Cada mapa tiene su propósito.

P: Como recorro un `HashMap`? R: Tres formas: `keySet()` (solo llaves), `values()` (solo valores), `entrySet()` (pares llave-valor, la más eficiente si necesitas ambos). El for-each funciona con las tres.

P: `HashSet` me dejo meter dos objetos "iguales". ¿Por qué? R: Porque tu clase no sobrescribio `equals()` y `hashCode()`. El set usa `hashCode()` para ubicar y `equals()` para comparar; sin sobrescribirlos, usa identidad de memoria y dos objetos con los mismos datos se ven "distintos". Regla: si sobrescribes uno, sobrescribe el otro.

UNIDAD 9: Que no se caiga, que se guarde, que corra en paralelo

Introducción

Tus programas hasta ahora viven en un mundo perfecto: el usuario siempre escribe números validos, los archivos siempre existen, y todo pasa en secuencia. El mundo real no es así: los usuarios escriben letras donde van números, los archivos se borran, la red se cae. En esta unidad tu programa aprende a sobrevivir: con las **excepciones** maneja errores sin caerse; con el manejo de **archivos** guarda datos que sobreviven al apagar la computadora; y con los **hilos** hace varias cosas al mismo tiempo.

Estos tres temas son los que más diferencian un programa de juguete de un programa profesional. Un banco no puede caerse porque alguien escribió mal un número; una app no puede perder tus datos al cerrarse; un videojuego no puede congelarse mientras carga el siguiente nivel. Al terminar esta unidad, tus programas serán robustos, persistentes y concurrentes: las tres palabras magicas del software serio.

Explicación detallada

Excepciones: los accidentes pasan, lo importante es sobrevivir

Una **excepción** es un evento que interrumpe el flujo normal del programa: dividir entre cero, leer un archivo que no existe, convertir "hola" a número. Sin manejo, la excepción **truen**a el programa: Java imprime un rastro feo (stack trace) y se detiene. Con manejo, tú decides que hacer.

```

public class Division {
    public static void main(String[] args) {
        try {
            int resultado = 10 / 0; // zona riesgosa
            System.out.println(resultado);
        } catch (ArithmeticException e) {
            System.out.println("Error: no se puede dividir entre cero");
            System.out.println("Detalle tecnico: " + e.getMessage());
        }
        System.out.println("El programa sigue vivo");
    }
}

```

- ``try`` = "intenta este código riesgoso".
- ``catch (TipoExcepcion e)`` = "si ocurre ESTE tipo de error, atrapalo y haz esto".
- Después del ``try-catch``, el programa **continúa** como si nada. Sin el ``try``, el programa habría muerto en la división.

Analogía: el ``try-catch`` es como el cinturón de seguridad. No evita el accidente (la división entre cero ocurriría igual), pero evita que sea fatal (el programa sobrevive y sigue). Los buenos conductores (programadores) siempre lo usan en zonas riesgosas.

La jerarquía de excepciones (el árbol genealógico)

Todas las excepciones heredan de ``Throwable``. Las dos ramas importantes:

- **``Exception``**: errores que tu programa PUEDE y DEBE manejar (archivo no encontrado, número mal formado). Casi todo lo que verás es de esta rama.
- **``Error``**: problemas graves de la máquina virtual (``OutOfMemoryError``, ``StackOverflowError``). Normalmente NO los atrapas: si la JVM se quedó sin memoria, tu ``catch`` no va a salvar nada.

Dentro de ``Exception`` hay dos familias:

- **Checked (verificadas)**: el compilador TE OBLIGA a manejarlas o declararlas. Ejemplo: ``IOException`` (leer archivos). Si no pones ``try-catch`` o ``throws``, no compila. Son para situaciones que un programa bien escrito debe anticipar.
- **Unchecked (no verificadas, ``RuntimeException``)**: el compilador no te obliga. Ejemplo: ``NullPointerException``, ``ArrayIndexOutOfBoundsException``, ``ArithmeticException``, ``NumberFormatException``. Son bugs o datos inesperados; las manejas si quieres, pero Java no te fuerza.

Exception

```
+-- IOException (checked: hay que manejarla o declararla)
+-- SQLException (checked)
+-- RuntimeException (unchecked: el compilador no exige)
    +-- NullPointerException
    +-- ArrayIndexOutOfBoundsException
    +-- ArithmeticException
    +-- NumberFormatException
```

Múltiples catch y el orden

Puedes atrapar distintos errores con distintos `catch`:

```
try {
    int[] arr = new int[3];
    arr[5] = 10 / 0;
} catch (ArithmeticException e) {
    System.out.println("Error aritmetico: " + e.getMessage());
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Indice fuera del arreglo");
} catch (Exception e) {
    System.out.println("Otro error: " + e.getMessage());
}
```

El orden importa: los `catch` se evalúan de arriba a abajo, y el primero que coincida gana. Pon los MÁS ESPECÍFICOS primero y el MÁS GENERAL (`Exception`) al final. Si pones `catch (Exception e)` primero, los demás jamás se alcanzan (y el compilador se queja de código inalcanzable).

Multi-catch (Java 7+): varios tipos en un solo `catch` si el manejo es el mismo:

```
try {
    // código riesgoso
} catch (ArithmeticException | ArrayIndexOutOfBoundsException e) {
    System.out.println("Error de cálculo o índice: " + e.getMessage());
}
```

finally: lo que siempre se ejecuta

El bloque `finally` corre SIEMPRE: haya excepción o no, se atrape o no. Su uso clásico: cerrar recursos (archivos, conexiones, scanners).

```

import java.io.FileWriter;

public class Escritor {
    public static void main(String[] args) {
        FileWriter fw = null;
        try {
            fw = new FileWriter("salida.txt");
            fw.write("Hola archivo");
        } catch (Exception e) {
            System.out.println("Error: " + e.getMessage());
        } finally {
            try {
                if (fw != null) {
                    fw.close(); // se cierra pase lo que pase
                }
            } catch (Exception e) {
                // ignorar error al cerrar
            }
        }
    }
}

```

Es verboso, por eso Java 7 invento el **try-with-resources** (lo verás en la sección de archivos): cierra solo los recursos automáticamente, sin `finally` manual.

throws: pasar la responsabilidad

Si un método no quiere manejar una excepción checked, puede **declararla** con `throws` y pasarle el problema a quien lo llame:

```

import java.io.FileWriter;
import java.io.IOException;

public class Guardador {
    // "este metodo puede lanzar IOException; quien me llame, que la maneje"
    public static void guardar(String texto) throws IOException {
        FileWriter fw = new FileWriter("datos.txt");
        fw.write(texto);
        fw.close();
    }

    public static void main(String[] args) {
        try {
            guardar("Hola");
        } catch (IOException e) {
            System.out.println("No se pudo guardar: " + e.getMessage());
        }
    }
}

```

La cadena de responsabilidad: `guardar` declara, `main` atrapa. En programas grandes, los métodos de bajo nivel declaran y los de alto nivel (cerca del usuario) atrapan y muestran mensajes amigables. **Nunca dejes un `catch` vacío** que trague el error en silencio: al menos imprime el mensaje. Los errores silenciados son pesadillas de depuración.

Crea tus propias excepciones

Puedes definir excepciones para las reglas de TU negocio:

```

public class SaldoInsuficienteException extends Exception {
    public SaldoInsuficienteException(String mensaje) {
        super(mensaje);
    }
}

public class Cuenta {
    private double saldo;

    public void retirar(double monto) throws SaldoInsuficienteException {
        if (monto > saldo) {
            throw new SaldoInsuficienteException(
                "Saldo: " + saldo + ", intentaste retirar: " + monto);
        }
        saldo -= monto;
    }
}

```

`throw new ...` LANZA la excepción (verbo activo: tu la provocas); `throws` en la firma la DECLARA (advierde que puede ocurrir). Con excepciones propias, tus errores hablan el idioma de tu negocio ("saldo insuficiente") en vez del idioma de Java ("ArithmeticException"). En sistemas reales, cada módulo define sus excepciones.

Buenas prácticas con excepciones

1. **Atrapa lo específico, no `Exception` generico** (salvo en la última barrera del programa).
2. **Jamás `catch` vacío**: registra o muestra el error.
3. **No uses excepciones para lógica normal**: `try-catch` no es un `if`. Validar con `if` es más rápido y más claro que atrapar la excepción.
4. **Limpia recursos**: usa try-with-resources para archivos y conexiones.
5. **Mensajes utiles**: incluye el dato que falló ("no se pudo abrir 'datos.txt'"), no solo "error".
6. **Falla rápido**: si los datos de entrada son invalidos, lanza la excepción pronto, no después de 200 líneas.

Archivos: tus datos sobreviven al apagon

Todo lo que guardas en variables muere cuando el programa termina. Los **archivos** son la memoria permanente: escribes datos en disco y los lees después, incluso días más tarde o desde otro programa.

Escribir un archivo de texto:

```
import java.io.FileWriter;
import java.io.IOException;

public class EscribirArchivo {
    public static void main(String[] args) {
        // try-with-resources: cierra el archivo solo, sin finally manual
        try (FileWriter fw = new FileWriter("notas.txt")) {
            fw.write("Primera linea\n");
            fw.write("Segunda linea\n");
            System.out.println("Archivo escrito");
        } catch (IOException e) {
            System.out.println("Error al escribir: " + e.getMessage());
        }
    }
}
```

- `new FileWriter("notas.txt")` crea (o SOBREESCRIBE) el archivo. Cuidado: si existía, su contenido se pierde.
- Para AGREGAR sin borrar, usa `new FileWriter("notas.txt", true)` (el `true` = modo append).
- `fw.write(...)` escribe texto. `\n` salta de línea.
- El `try ( ... )` con paréntesis es el **try-with-resources**: al terminar (bien o mal), Java cierra el `FileWriter` automáticamente. Usalo SIEMPRE con archivos.

Leer un archivo de texto:

```

import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class LeerArchivo {
    public static void main(String[] args) {
        try (Scanner lector = new Scanner(new File("notas.txt"))) {
            while (lector.hasNextLine()) {
                String linea = lector.nextLine();
                System.out.println("> " + linea);
            }
        } catch (FileNotFoundException e) {
            System.out.println("El archivo no existe: " + e.getMessage());
        }
    }
}

```

El `Scanner` que ya conoces también lee archivos: en vez de `System.in`, le pasas un `File`. `hasNextLine()` pregunta si hay más líneas. Así de simple: lo que sabías del teclado se reutiliza para archivos.

Donde se crea el archivo? En la carpeta desde donde corres el programa (el "directorio de trabajo"). Si usas IntelliJ, suele ser la carpeta del proyecto. Para rutas exactas usa rutas absolutas (`"C:/datos/notas.txt"`), pero en práctica se prefieren rutas relativas al proyecto.

La clase `File`: representa un archivo o carpeta (no su contenido, sino su existencia, tamaño, etc.):

```

import java.io.File;

File f = new File("notas.txt");
System.out.println("Existe: " + f.exists());
System.out.println("Tamaño: " + f.length() + " bytes");
System.out.println("Es archivo: " + f.isFile());
System.out.println("Ruta absoluta: " + f.getAbsolutePath());

File carpeta = new File("misdatos");
carpeta.mkdir(); // crea la carpeta

```

Guardar objetos como texto (formato simple): para guardar una agenda, escribe cada contacto en una línea con separador:

Ana,555-1234
Luis,555-5678

Al leer, partes cada línea con ``split(",")``. Es el formato CSV (valores separados por comas), el más usado del mundo para intercambiar datos simples. En la práctica de esta unidad lo implementarás completo.

Hilos: varias cosas a la vez

Hasta ahora tu programa hace UNA cosa a la vez, en secuencia. Un **hilo** (thread) es una línea de ejecución independiente: con dos hilos, tu programa hace dos cosas "al mismo tiempo" (en realidad el procesador alterna rapidísimo entre ellos, o usa varios núcleos).

```
public class HiloDemo {
    public static void main(String[] args) {
        Thread hilo = new Thread(() -> {
            for (int i = 1; i <= 5; i++) {
                System.out.println("Hilo secundario: " + i);
                try {
                    Thread.sleep(500); // duerme 500 ms
                } catch (InterruptedException e) {
                    System.out.println("Hilo interrumpido");
                }
            }
        });

        hilo.start(); // INICIA el hilo (no lo llames run() directo!)

        for (int i = 1; i <= 5; i++) {
            System.out.println("Hilo principal: " + i);
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                System.out.println("Principal interrumpido");
            }
        }
        System.out.println("Fin del main");
    }
}
```

- ``new Thread(() -> { ... })``: creas un hilo con una lambda que contiene su tarea.

- `hilo.start()`: INICIA la ejecución en paralelo. **Jamás llames `run()` directamente**: eso ejecutaría el código en el hilo actual, sin paralelismo.
- `Thread.sleep(ms)`: pausa el hilo actual. Lanza `InterruptedException` (checked: hay que atraparla).
- La salida mezcla líneas de ambos hilos: están corriendo en paralelo de verdad.

Analogía: tu programa es una cocina con un cocinero (hilo principal). Crear otro hilo es contratar un segundo cocinero: mientras uno pica verdura, el otro atiende la parrilla. Todo más rápido, pero hay que coordinarlos para que no choquen (ver sincronización abajo).

`Runnable` explícito (la forma clásica, sin lambda):

```
public class Tarea implements Runnable {
    private String nombre;

    public Tarea(String nombre) {
        this.nombre = nombre;
    }

    @Override
    public void run() {
        for (int i = 1; i <= 3; i++) {
            System.out.println(nombre + ": paso " + i);
        }
    }

    public static void main(String[] args) {
        Thread t1 = new Thread(new Tarea("Descarga"));
        Thread t2 = new Thread(new Tarea("Proceso"));
        t1.start();
        t2.start();
    }
}
```

`join()`: esperar a que termine:

```
t1.start();
t2.start();
t1.join(); // el main espera a que t1 termine
t2.join(); // y a que t2 termine
System.out.println("Ambos terminaron, sigo");
```

`join` es como decir "no sigas hasta que este hilo acabe". Esencial cuando necesitas los resultados de los hilos antes de continuar.

El problema: condiciones de carrera. Si dos hilos modifican la MISMA variable sin coordinación, el resultado es impredecible:

```
class Contador {
    private int total = 0;
    public void incrementar() { total++; } // NO es atomico!
    public int getTotal() { return total; }
}
// Dos hilos incrementando 1000 veces cada uno...
// ...el total rara vez da 2000. Por que?
```

Porque `total++` son TRES pasos (leer, sumar 1, escribir). Si el hilo A lee 5 y antes de escribir el hilo B también lee 5, ambos escriben 6: se perdió un incremento. Esto se llama **condición de carrera** y es el bug más traicionero de la programación concurrente: a veces funciona, a veces no.

La solución: `synchronized`:

```
class ContadorSeguro {
    private int total = 0;
    public synchronized void incrementar() { total++; }
    public int getTotal() { return total; }
}
```

`synchronized` pone un candado: solo un hilo a la vez ejecuta el método. El otro espera su turno. Simple y efectivo para casos basicos. (Java tiene herramientas más avanzadas: `Lock`, `AtomicInteger`, `ConcurrentHashMap`, pools de hilos con `ExecutorService`. Para este nivel, `synchronized` y entender el problema bastan.)

Cuando usar hilos (casos reales):

- Descargar archivos mientras la interfaz sigue respondiendo.
- Un servidor atendiendo a varios clientes a la vez.
- Procesar imagenes o datos en paralelo para ir más rápido.
- Juegos: un hilo para la lógica, otro para el sonido, otro para la red.

Cuando NO usar hilos: si tu tarea es secuencial por naturaleza, los hilos solo agregan complejidad y bugs. La concurrencia es una herramienta poderosa y peligrosa: usala donde pague.

Errores comunes de la unidad

1. **Atrapar `Exception` generico y no hacer nada:** el programa "funciona" pero los errores se tragan en silencio.
2. **Olvidar cerrar archivos:** sin try-with-resources, un error a mitad deja el archivo abierto y bloqueado.
3. **`FileWriter` sin append borra el contenido:** si querias agregar, usa el constructor con `true`.
4. **Llamar `run()` en vez de `start()`:** no hay paralelismo, solo una llamada normal.
5. **No manejar `InterruptedException`:** es checked, el compilador la exige.
6. **Condiciones de carrera:** compartir variables entre hilos sin `synchronized`.
7. **Rutas de archivo incorrectas:** el archivo se crea donde corres el programa, no donde esta el `.java`. Verifica con `getAbsolutePath()`.

Ejemplos paso a paso

Ejemplo 1: Calculadora que no se cae

```

import java.util.Scanner;

public class CalculadoraSegura {
    public static double dividir(double a, double b) {
        if (b == 0) {
            throw new ArithmeticException("Division entre cero");
        }
        return a / b;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        try {
            System.out.print("Dividendo: ");
            double a = Double.parseDouble(sc.nextLine());
            System.out.print("Divisor: ");
            double b = Double.parseDouble(sc.nextLine());

            System.out.println("Resultado: " + dividir(a, b));
        } catch (NumberFormatException e) {
            System.out.println("Error: debes escribir numeros validos");
        } catch (ArithmeticException e) {
            System.out.println("Error: " + e.getMessage());
        } finally {
            sc.close();
            System.out.println("Calculadora cerrada");
        }
    }
}

```

Tres zonas protegidas: conversión de texto (NumberFormatException), división (ArithmeticException) y cierre del Scanner (finally). El usuario puede equivocarse todo lo que quiera: el programa responde con mensajes claros en vez de tronar.

Ejemplo 2: Agenda que se guarda en archivo (CSV)

```

import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Scanner;

class ContactoCSV {

```

```

String nombre;
String telefono;

ContactoCSV(String nombre, String telefono) {
    this.nombre = nombre;
    this.telefono = telefono;
}

// Convierte el objeto a una linea CSV
String aLinea() {
    return nombre + "," + telefono;
}

// Crea el objeto desde una linea CSV
static ContactoCSV desdeLinea(String linea) {
    String[] partes = linea.split(",");
    return new ContactoCSV(partes[0], partes[1]);
}
}

public class AgendaPersistente {
    static final String ARCHIVO = "agenda.csv";

    static void guardar(ArrayList<ContactoCSV> agenda) throws IOException {
        try (FileWriter fw = new FileWriter(ARCHIVO)) {
            for (ContactoCSV c : agenda) {
                fw.write(c.aLinea() + "\n");
            }
        }
    }

    static ArrayList<ContactoCSV> cargar() throws IOException {
        ArrayList<ContactoCSV> agenda = new ArrayList<>();
        File f = new File(ARCHIVO);
        if (!f.exists()) {
            return agenda; // primera vez: agenda vacia
        }
        try (Scanner lector = new Scanner(f)) {
            while (lector.hasNextLine()) {
                agenda.add(ContactoCSV.desdeLinea(lector.nextLine()));
            }
        }
        return agenda;
    }
}

```

```

public static void main(String[] args) {
    try {
        ArrayList<ContactoCSV> agenda = cargar();
        System.out.println("Contactos cargados: " + agenda.size());

        agenda.add(new ContactoCSV("Ana", "555-1234"));
        agenda.add(new ContactoCSV("Luis", "555-5678"));

        guardar(agenda);
        System.out.println("Agenda guardada en " + ARCHIVO);
    } catch (IOException e) {
        System.out.println("Error con el archivo: " + e.getMessage());
    }
}
}

```

Persistencia conseguida: corre el programa dos veces y verás que los contactos se acumulan (se cargan los viejos y se agregan nuevos). Los métodos ``aLinea`/`desdeLinea`` son el patrón de serialización manual: convertir objetos a texto y de regreso. Las bases de datos (Unidad 10) hacen esto de forma más potente, pero el principio es el mismo.

Ejemplo 3: Descargas simuladas en paralelo

```

public class Descargas {
    static void descargar(String archivo, int partes) {
        Thread t = new Thread(() -> {
            for (int i = 1; i <= partes; i++) {
                System.out.println(archivo + ": parte " + i + "/" + partes);
                try {
                    Thread.sleep(300);
                } catch (InterruptedException e) {
                    System.out.println(archivo + " interrumpido");
                    return;
                }
            }
            System.out.println(archivo + " COMPLETO");
        });
        t.start();
    }

    public static void main(String[] args) throws InterruptedException {
        long inicio = System.currentTimeMillis();

        descargar("video.mp4", 5);
        descargar("musica.mp3", 5);
        descargar("foto.jpg", 5);

        // Espera simple: en código real usarías join() con las referencias
        Thread.sleep(2000);

        long fin = System.currentTimeMillis();
        System.out.println("Tiempo total: " + (fin - inicio) + " ms");
    }
}

```

Tres descargas en paralelo terminan en ~1.5 segundos; en secuencia tardarían ~4.5. El tiempo lo demuestra: la concurrencia paga cuando hay esperas (red, disco, usuario).

Ejemplo 4: Contador con y sin sincronización

```

class ContadorInseguro {
    int total = 0;
    void incrementar() { total++; }
}

class ContadorSeguro {
    private int total = 0;
    synchronized void incrementar() { total++; }
    int getTotal() { return total; }
}

public class Carrera {
    public static void main(String[] args) throws InterruptedException {
        ContadorSeguro c = new ContadorSeguro();

        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 10000; i++) c.incrementar();
        });
        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 10000; i++) c.incrementar();
        });

        t1.start();
        t2.start();
        t1.join();
        t2.join();

        System.out.println("Total (deberia ser 20000): " + c.getTotal());
    }
}

```

Con `synchronized`, el resultado es 20000 siempre. Quita el `synchronized` y corre varias veces: verás números menores e inconsistentes. Ver el bug con tus propios ojos vale más que mil explicaciones.

Práctica

Ejercicio 1. Escribe `EdadSegura.java`: pide la edad con Scanner dentro de un `try-catch` que atrape `NumberFormatException` y repita (ciclo) hasta que sea un número valido entre 0 y 120. Combina excepciones con el validador de la Unidad 4.

Ejercicio 2. Define la excepción `MontoInvalidoException` (checked, extends `Exception`) y usala en una clase `Billetera` cuyo método `depositar` la lance si el monto es ≤ 0 . En el `main`, atrápala y muestra un mensaje amigable.

Ejercicio 3. Escribe `Diario.java`: un programa que agregue líneas a un archivo `diario.txt` (modo `append`). Cada ejecución pide un texto al usuario, le antepone la fecha/hora actual (`java.time.LocalDateTime.now()`), y lo agrega. Al inicio muestra las últimas 5 líneas del archivo si existe.

Solución (esqueleto):

```

import java.io.File;
import java.io.FileWriter;
import java.time.LocalDateTime;
import java.util.ArrayList;
import java.util.Scanner;

public class Diario {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        File f = new File("diario.txt");

        // Mostrar ultimas lineas si existe
        if (f.exists()) {
            try (Scanner lector = new Scanner(f)) {
                ArrayList<String> lineas = new ArrayList<>();
                while (lector.hasNextLine()) {
                    lineas.add(lector.nextLine());
                }
                int desde = Math.max(0, lineas.size() - 5);
                System.out.println("--- Ultimas entradas ---");
                for (int i = desde; i < lineas.size(); i++) {
                    System.out.println(lineas.get(i));
                }
            } catch (Exception e) {
                System.out.println("Error al leer: " + e.getMessage());
            }
        }

        // Agregar nueva entrada (append = true)
        System.out.print("Nueva entrada: ");
        String texto = sc.nextLine();
        try (FileWriter fw = new FileWriter("diario.txt", true)) {
            fw.write(LocalDateTime.now() + " | " + texto + "\n");
            System.out.println("Guardado.");
        } catch (Exception e) {
            System.out.println("Error al escribir: " + e.getMessage());
        }
        sc.close();
    }
}

```

Ejercicio 4. Escribe `CopiarArchivo.java`: lee `origen.txt` y lo copia a `copia.txt` línea por línea. Maneja `FileNotFoundException` con mensaje claro si el origen no existe. Luego verifica que ambos archivos tengan el mismo contenido.

Ejercicio 5. Escribe `Reloj.java`: un hilo que imprima la hora actual cada segundo, 10 veces (`LocalTime.now()` + `Thread.sleep(1000)`), mientras el hilo principal imprime "trabajando..." cada 700ms, 10 veces. Observa cómo se intercalan.

Ejercicio 6. Modifica el `ContadorSeguro` del Ejemplo 4: agrega un método `synchronized` para decrementar y otro para obtener el valor. Crea 4 hilos (2 incrementan 5000 veces, 2 decrementan 5000 veces) y verifica que el total final sea 0. Luego quita `synchronized` y observa el caos.

Ejercicio 7 (reto). Escribe `DescargadorPro.java`: 3 hilos "descargan" archivos con duraciones distintas (partes distintas). El `main` usa `join()` en los tres (guarda las referencias `Thread`) y al final imprime "Todas las descargas completas" con el tiempo total. Sin `join` el mensaje saldría antes de tiempo: compruébalo quitándolo.

Resumen de la unidad

- Las excepciones son errores en ejecución: `try` intenta, `catch` atrapa por tipo, el programa sobrevive.
- `Exception` (maneja) vs `Error` (grave, no lo atrapes); checked (el compilador obliga) vs unchecked (`RuntimeException`, no obliga).
- Ordena los `catch` de específico a general; multi-catch con `{}|`.
- `finally` siempre se ejecuta; try-with-resources cierra archivos solo (usalo siempre).
- `throws` declara, `throw new` lanza; crea tus propias excepciones para tu negocio.
- Archivos: `FileWriter` escribe (con `true` agrega sin borrar), `Scanner(new File(...))` lee línea por línea.
- Guarda objetos como CSV: `aLinea()` / `desdeLinea()` con `split(",")`.
- Hilos: `new Thread(tarea)`, `start()` (jamás `run()` directo), `sleep`, `join` para esperar.
- Condición de carrera: dos hilos, una variable, resultado impredecible; `synchronized` lo evita.
- Usa hilos para esperas en paralelo (red, disco); no para lógica secuencial simple.

Stack trace: lee el rastro del crimen

Cuando una excepción no se atrapa, Java imprime el **stack trace**: la ruta exacta del error.

```
Exception in thread "main" java.lang.NullPointerException
    at Agenda.buscar(Agenda.java:15)
    at Agenda.main(Agenda.java:30)
```

Como leerlo (de arriba hacia abajo):

1. **La primera línea:** que paso (`NullPointerException`) y en que hilo.
2. **La segunda línea:** DONDE nació el error (clase, método, archivo:línea). `Agenda.java:15` = línea 15.
3. **Las siguientes:** la cadena de llamadas que llevo ahí (`main` línea 30 llamó a `buscar`, que falló en la 15).

El 90% de las veces, la línea 2 del trace es exactamente donde esta tu bug. Los principiantes se asustan con el trace; los profesionales lo leen como un mapa del tesoro: te dice el archivo y la línea. Aprende a amarlo.

Logging: más allá del println

En programas reales, los errores no se imprimen con `println`: se **registran** (logging) con niveles (INFO, WARNING, SEVERE) en archivos de log, usando librerías como `java.util.logging` (incluida en Java):

```
import java.util.logging.Logger;
import java.util.logging.Level;

public class ConLog {
    private static final Logger LOG = Logger.getLogger(ConLog.class.getName());

    public static void main(String[] args) {
        LOG.info("Iniciando programa");
        try {
            int x = 10 / 0;
        } catch (ArithmeticException e) {
            LOG.log(Level.SEVERE, "Error en calculo", e);
        }
        LOG.info("Terminando programa");
    }
}
```

El log incluye fecha, nivel y el stack trace completo, y puede configurarse para escribir a archivo. Cuando tu programa corra en un servidor sin pantalla, el log será tus ojos. Acostumbrate desde ahora: `LOG.severe(...)` en vez de `System.out.println("ERROR...")`.

Mini glosario de la unidad

- **Excepción:** evento anormal que interrumpe el flujo normal; se maneja con try-catch.
- **Stack trace:** rastro impreso con la cadena de llamadas donde nació el error.
- **Checked/unchecked:** excepciones que el compilador obliga a manejar / las que no.
- **finally:** bloque que siempre se ejecuta, haya excepción o no.
- **Try-with-resources:** try que cierra recursos automáticamente.
- **Persistencia:** guardar datos para que sobrevivan al fin del programa.
- **CSV:** formato de texto con valores separados por comas.
- **Hilo (thread):** línea de ejecución independiente dentro de un programa.
- **Condición de carrera:** bug por acceso no coordinado a datos compartidos entre hilos.
- **synchronized:** candado que permite un solo hilo a la vez en un método/bloque.
- **Logging:** registro sistematico de eventos y errores con niveles de gravedad.

Caso de estudio: respaldo paralelo con hilos

Un programa que respalda (copia) varios archivos. Secuencial tardaría la suma de todos; con hilos, los archivos se copian en paralelo y el tiempo total es el del archivo más lento.

```

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.util.ArrayList;

public class RespaldoParalelo {

    static void copiar(String origen, String destino) {
        try (FileInputStream in = new FileInputStream(origen);
            FileOutputStream out = new FileOutputStream(destino)) {
            byte[] buffer = new byte[8192];
            int leidos;
            while ((leidos = in.read(buffer)) != -1) {
                out.write(buffer, 0, leidos);
            }
            System.out.println("OK: " + origen + " -> " + destino);
        } catch (Exception e) {
            System.out.println("FALLO " + origen + ": " + e.getMessage());
        }
    }

    public static void main(String[] args) throws InterruptedException {
        String[][] trabajos = {
            {"datos1.csv", "respaldo/datos1.csv"},
            {"datos2.csv", "respaldo/datos2.csv"},
            {"datos3.csv", "respaldo/datos3.csv"}
        };

        ArrayList<Thread> hilos = new ArrayList<>();
        long inicio = System.currentTimeMillis();

        for (String[] t : trabajos) {
            Thread hilo = new Thread(() -> copiar(t[0], t[1]));
            hilos.add(hilo);
            hilo.start();
        }

        for (Thread h : hilos) {
            h.join();
        }

        long fin = System.currentTimeMillis();
        System.out.println("Respaldo completo en " + (fin - inicio) + " ms");
    }
}

```

Que observar:

- Cada hilo copia UN archivo de forma independiente: no comparten datos, así que no hay condición de carrera ni se necesita `synchronized`. El paralelismo más seguro es el que no comparte nada.
- El `buffer` de 8 KB: leer byte por byte seria lentísimo; leer por bloques es el estandar para copiar archivos.
- El `main` declara `throws InterruptedException` para no envolver los `join()` en try-catch: valido aquí porque es el punto de entrada y no hay quien atrape arriba.
- Medir el tiempo con `currentTimeMillis()` antes y después: la forma más simple de comprobar que el paralelismo sirvió. Si los archivos están en el mismo disco duro mecanico, la ganancia es menor (el disco es el cuello de botella); en red o SSD, la diferencia es notable.

Lección de diseno: no todo se paraleliza igual. Tareas independientes (copiar archivos, procesar imagenes, descargar URLs) son paralelismo ideal. Tareas que comparten un contador necesitan sincronización. Antes de crear hilos, pregunta: que comparten? Si la respuesta es "nada", adelante.

Preguntas frecuentes de la unidad

P: Atrapar `Exception` generica es malo? R: Atrapar lo generico esconde bugs: atrapas también lo que no esperabas y el programa sigue como si nada. Atrapa lo específico (`IOException`, `NumberFormatException`) y deja que lo inesperado truene para que lo veas y lo arregles.

P: Mi archivo se crea pero sale vacío. Que paso? R: Casi seguro olvidaste cerrar el `FileWriter` (o el `flush()`). Los datos se quedan en el buffer de memoria y nunca llegan al disco. El try-with-resources elimina este error por completo: usalo siempre.

P: `FileWriter` me borró mi archivo. Como lo recupero? R: No se puede: el modo por defecto trunca. Lección aprendida a la mala por todos. De ahora en adelante: si vas a agregar, usa el constructor con `true`; y para datos importantes, haz respaldo antes de experimentar.

P: Cuantos hilos puedo crear? R: Tecnicamente miles, pero cada hilo consume memoria (~1 MB de stack) y el cambio de contexto tiene costo. Para tareas pesadas usa un pool de hilos (`ExecutorService`) con un número cercano a tus nucleos de CPU. Crear hilos sin control es como abrir mil cajas en el super con un solo cajero.

P: `println` en varios hilos mezcla las líneas. Es normal? R: Si dos hilos imprimen al mismo tiempo, sus textos pueden intercalarse a mitad de línea. `println` es thread-safe por llamada, pero una "línea lógica" de varios `print` no es atómica. Para logs serios usa un logger con sincronización o un `StringBuilder` por hilo.

UNIDAD 10: Proyectos reales y herramientas profesionales

Introducción

Llegaste a la unidad final del contenido, y es la más emocionante: aquí dejas de hacer ejercicios y construyes **proyectos completos**, como los de un trabajo real. Armaras una calculadora con menu, una agenda con persistencia en archivo, y una tienda con POO. Luego aprenderas **Maven** (el administrador de dependencias que usan las empresas) y **JDBC** (cómo conectar Java con una base de datos MySQL de verdad). Cerraremos con tu graduación y la ruta para seguir creciendo.

Esta unidad cambia tu perfil: ya no eres "alguien que sabe sintaxis de Java", eres "alguien que construye sistemas". Los proyectos que hagas aquí van a tu portafolio: cuando busques trabajo o clientes, nadie te pide el examen del libro; te piden "muéstrame que has construido". Así que tomate estos proyectos en serio, terminalos, y luego mejóralos con tus propias ideas.

Proyecto 1: Calculadora en consola

Objetivo: calculadora con menu repetido que sume, reste, multiplique, divida y calcule potencia, sin caerse ante errores.

Diseño: un `main` con menu `do-while` (Unidades 3-4), un método por operación (Unidad 5), validación de entrada con try-catch (Unidad 9), y división protegida.

```
import java.util.Scanner;

public class Calculadora {
    static Scanner sc = new Scanner(System.in);

    static double pedirNumero(String mensaje) {
        while (true) {
            try {
                System.out.print(mensaje);
                return Double.parseDouble(sc.nextLine());
            } catch (NumberFormatException e) {
                System.out.println("Escribe un numero valido.");
            }
        }
    }
}
```

```

}

static double sumar(double a, double b) { return a + b; }
static double restar(double a, double b) { return a - b; }
static double multiplicar(double a, double b) { return a * b; }

static double dividir(double a, double b) {
    if (b == 0) {
        throw new ArithmeticException("No se puede dividir entre cero");
    }
    return a / b;
}

static double potencia(double base, double exp) {
    return Math.pow(base, exp);
}

public static void main(String[] args) {
    int opcion;
    System.out.println("=== CALCULADORA JAVA ===");

    do {
        System.out.println("\n1. Sumar  2. Restar  3. Multiplicar");
        System.out.println("4. Dividir  5. Potencia  0. Salir");
        System.out.print("Elige: ");
        try {
            opcion = Integer.parseInt(sc.nextLine());
        } catch (NumberFormatException e) {
            System.out.println("Opcion invalida.");
            continue;
        }

        if (opcion >= 1 && opcion <= 5) {
            double a = pedirNumero("Primer numero: ");
            double b = pedirNumero("Segundo numero: ");
            try {
                double r = 0;
                switch (opcion) {
                    case 1: r = sumar(a, b); break;
                    case 2: r = restar(a, b); break;
                    case 3: r = multiplicar(a, b); break;
                    case 4: r = dividir(a, b); break;
                    case 5: r = potencia(a, b); break;
                }
            }
            System.out.println("Resultado: " + r);
        }
    } while (opcion != 0);
}

```

```

        } catch (ArithmeticException e) {
            System.out.println("Error: " + e.getMessage());
        }
    } else if (opcion != 0) {
        System.out.println("Opcion invalida.");
    }
} while (opcion != 0);

System.out.println("Gracias por usar la calculadora.");
sc.close();
}
}

```

Que prácticas aquí: menu con `do-while` + `switch`, métodos por operación, `pedirNumero` que repite hasta dato valido (el guardián de la Unidad 4), excepciones para la división entre cero, y `static` para compartir el Scanner. Es pequeno pero completo: entrada, proceso, salida, validación y robustez.

Mejoras que puedes agregarle tu: historial de operaciones (guardalas en un `ArrayList<String>` y agrega opción "ver historial"), raíz cuadrada, o guardar el historial en archivo (Unidad 9).

Proyecto 2: Agenda de contactos (CRUD + persistencia)

Objetivo: agenda con agregar, listar, buscar, eliminar y persistencia en CSV. Un **CRUD** completo (Create, Read, Update, Delete... aquí haremos Create, Read, Delete; el Update queda de ejercicio).

Diseno: clase `Contacto` (POO, Unidad 6), `ArrayList<Contacto>` dinámico (Unidad 8), menu `do-while`, y archivo CSV para persistencia (Unidad 9). Cada pieza que aprendiste tiene su lugar.

```

import java.io.File;
import java.util.ArrayList;
import java.util.Scanner;
import java.io.FileWriter;

class Contacto {
    private String nombre;
    private String telefono;
    private String email;

    public Contacto(String nombre, String telefono, String email) {

```

```

        this.nombre = nombre;
        this.telefono = telefono;
        this.email = email;
    }

    public String getNombre() { return nombre; }
    public String getTelefono() { return telefono; }
    public String getEmail() { return email; }

    public String aCSV() {
        return nombre + "," + telefono + "," + email;
    }

    public static Contacto desdeCSV(String linea) {
        String[] p = linea.split(",");
        return new Contacto(p[0], p[1], p[2]);
    }

    @Override
    public String toString() {
        return nombre + " | " + telefono + " | " + email;
    }
}

public class AgendaApp {
    static final String ARCHIVO = "agenda.csv";
    static Scanner sc = new Scanner(System.in);
    static ArrayList<Contacto> agenda = new ArrayList<>();

    static void cargar() {
        File f = new File(ARCHIVO);
        if (!f.exists()) return;
        try (Scanner lector = new Scanner(f)) {
            while (lector.hasNextLine()) {
                agenda.add(Contacto.desdeCSV(lector.nextLine()));
            }
            System.out.println("Se cargaron " + agenda.size() + " contactos.");
        } catch (Exception e) {
            System.out.println("Error al cargar: " + e.getMessage());
        }
    }

    static void guardar() {
        try (FileWriter fw = new FileWriter(ARCHIVO)) {
            for (Contacto c : agenda) {

```

```

        fw.write(c.aCSV() + "\n");
    }
} catch (Exception e) {
    System.out.println("Error al guardar: " + e.getMessage());
}
}

static void agregar() {
    System.out.print("Nombre: ");
    String nombre = sc.nextLine();
    System.out.print("Telefono: ");
    String tel = sc.nextLine();
    System.out.print("Email: ");
    String email = sc.nextLine();
    agenda.add(new Contacto(nombre, tel, email));
    guardar();
    System.out.println("Contacto agregado.");
}

static void listar() {
    if (agenda.isEmpty()) {
        System.out.println("Agenda vacia.");
        return;
    }
    for (int i = 0; i < agenda.size(); i++) {
        System.out.println((i + 1) + ". " + agenda.get(i));
    }
}

static void buscar() {
    System.out.print("Nombre a buscar: ");
    String nombre = sc.nextLine();
    boolean encontrado = false;
    for (Contacto c : agenda) {
        if (c.getNombre().toLowerCase().contains(nombre.toLowerCase())) {
            System.out.println(c);
            encontrado = true;
        }
    }
    if (!encontrado) {
        System.out.println("Sin resultados.");
    }
}

static void eliminar() {

```

```

        listar();
        if (agenda.isEmpty()) return;
        System.out.print("Numero a eliminar: ");
        try {
            int n = Integer.parseInt(sc.nextLine());
            if (n >= 1 && n <= agenda.size()) {
                Contacto c = agenda.remove(n - 1);
                guardar();
                System.out.println("Eliminado: " + c.getNombre());
            } else {
                System.out.println("Numero invalido.");
            }
        } catch (NumberFormatException e) {
            System.out.println("Escribe un numero.");
        }
    }

    public static void main(String[] args) {
        cargar();
        int opcion;
        do {
            System.out.println("\n=== AGENDA ===");
            System.out.println("1. Agregar  2. Listar  3. Buscar  4. Eliminar  0. Salir");
            System.out.print("Elige: ");
            try {
                opcion = Integer.parseInt(sc.nextLine());
            } catch (NumberFormatException e) {
                opcion = -1;
            }

            switch (opcion) {
                case 1: agregar(); break;
                case 2: listar(); break;
                case 3: buscar(); break;
                case 4: eliminar(); break;
                case 0: System.out.println("Adios!"); break;
                default: System.out.println("Opcion invalida.");
            }
        } while (opcion != 0);
        sc.close();
    }
}

```

Que prácticas aquí: CRUD completo, búsqueda parcial con `contains` (mejor que igualdad exacta para usabilidad), persistencia automática (cada cambio se guarda: el usuario nunca pierde datos), carga al inicio, y validación de índices. Este programa, con una interfaz gráfica encima, es una app real de contactos.

Mejoras para tu portafolio: opción editar (el Update del CRUD), ordenar por nombre antes de listar, exportar a otro formato, o buscar también por teléfono.

Proyecto 3: Tienda con POO (tu primer sistema)

Objetivo: sistema de tienda con productos, clientes y pedidos, usando herencia y encapsulamiento de verdad. Este es el proyecto que demuestra que entiendes POO.

Diseño de clases (haz el ejercicio mental de la Unidad 6: sustantivos -> clases):

- `Producto`: código, nombre, precio, stock. Métodos: `vender(cantidad)`, `reabastecer(cantidad)`, `valorInventario`.
- `Cliente`: nombre, email. Métodos: getters, `toString`.
- `Pedido`: cliente, lista de productos + cantidades, fecha. Métodos: `agregarProducto(producto, cantidad)`, `calcularTotal()`, `mostrarTicket`.
- `Tienda` (con `main`): catálogo (`HashMap<String, Producto>` por código), lista de clientes, creación de pedidos, menu.

```
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Scanner;

class Producto {
    private String codigo;
    private String nombre;
    private double precio;
    private int stock;

    public Producto(String codigo, String nombre, double precio, int stock) {
        this.codigo = codigo;
        this.nombre = nombre;
        setPrecio(precio);
        setStock(stock);
    }

    public String getCodigo() { return codigo; }
    public String getNombre() { return nombre; }
```

```

    public double getPrecio() { return precio; }
    public int getStock() { return stock; }

    public void setPrecio(double precio) {
        if (precio > 0) this.precio = precio;
    }

    public void setStock(int stock) {
        if (stock >= 0) this.stock = stock;
    }

    public boolean vender(int cantidad) {
        if (cantidad > 0 && cantidad <= stock) {
            stock -= cantidad;
            return true;
        }
        return false;
    }

    @Override
    public String toString() {
        return codigo + " | " + nombre + " | $" + precio + " | stock: " + stock;
    }
}

class Cliente {
    private String nombre;
    private String email;

    public Cliente(String nombre, String email) {
        this.nombre = nombre;
        this.email = email;
    }

    public String getNombre() { return nombre; }
    public String getEmail() { return email; }

    @Override
    public String toString() {
        return nombre + " (" + email + ")";
    }
}

class LineaPedido {
    private Producto producto;

```

```

    private int cantidad;

    public LineaPedido(Producto producto, int cantidad) {
        this.producto = producto;
        this.cantidad = cantidad;
    }

    public double subtotal() {
        return producto.getPrecio() * cantidad;
    }

    @Override
    public String toString() {
        return cantidad + " x " + producto.getNombre() + " = $" + subtotal();
    }
}

class Pedido {
    private static int folioSiguiente = 1;
    private int folio;
    private Cliente cliente;
    private ArrayList<LineaPedido> lineas;

    public Pedido(Cliente cliente) {
        this.folio = folioSiguiente++;
        this.cliente = cliente;
        this.lineas = new ArrayList<>();
    }

    public boolean agregarProducto(Producto p, int cantidad) {
        if (p.vender(cantidad)) {
            lineas.add(new LineaPedido(p, cantidad));
            return true;
        }
        return false;
    }

    public double calcularTotal() {
        double total = 0;
        for (LineaPedido l : lineas) {
            total += l.subtotal();
        }
        return total;
    }
}

```

```

    public void mostrarTicket() {
        System.out.println("\n==== TICKET #" + folio + " =====");
        System.out.println("Cliente: " + cliente);
        for (LineaPedido l : lineas) {
            System.out.println("  " + l);
        }
        System.out.println("TOTAL: $" + calcularTotal());
        System.out.println("=====");
    }
}

```

Y la tienda que orchestra (versión compacta del `main` con menu):

```

public class TiendaApp {
    static Scanner sc = new Scanner(System.in);
    static HashMap<String, Producto> catalogo = new HashMap<>();
    static ArrayList<Cliente> clientes = new ArrayList<>();

    public static void main(String[] args) {
        // Datos de ejemplo
        catalogo.put("A01", new Producto("A01", "Laptop", 15000, 5));
        catalogo.put("A02", new Producto("A02", "Mouse", 300, 20));
        catalogo.put("A03", new Producto("A03", "Teclado", 800, 15));
        clientes.add(new Cliente("Ana", "ana@mail.com"));

        int opcion;
        do {
            System.out.println("\n=== TIENDA ===");
            System.out.println("1. Ver catalogo  2. Nuevo pedido  0. Salir");
            System.out.print("Elige: ");
            try {
                opcion = Integer.parseInt(sc.nextLine());
            } catch (NumberFormatException e) {
                opcion = -1;
            }

            if (opcion == 1) {
                for (Producto p : catalogo.values()) {
                    System.out.println(p);
                }
            } else if (opcion == 2) {
                hacerPedido();
            } else if (opcion != 0) {
                System.out.println("Opcion invalida.");
            }
        } while (opcion != 0);
    }
}

```

```

        sc.close();
    }

    static void hacerPedido() {
        System.out.print("Nombre del cliente: ");
        String nombre = sc.nextLine();
        Cliente cliente = null;
        for (Cliente c : clientes) {
            if (c.getNombre().equalsIgnoreCase(nombre)) {
                cliente = c;
            }
        }
        if (cliente == null) {
            System.out.print("Email del nuevo cliente: ");
            cliente = new Cliente(nombre, sc.nextLine());
            clientes.add(cliente);
        }

        Pedido pedido = new Pedido(cliente);
        while (true) {
            System.out.print("Codigo de producto (o 'fin'): ");
            String codigo = sc.nextLine();
            if (codigo.equalsIgnoreCase("fin")) break;
            Producto p = catalogo.get(codigo);
            if (p == null) {
                System.out.println("Codigo no existe.");
                continue;
            }
            System.out.print("Cantidad: ");
            try {
                int cant = Integer.parseInt(sc.nextLine());
                if (pedido.agregarProducto(p, cant)) {
                    System.out.println("Agregado.");
                } else {
                    System.out.println("Stock insuficiente (hay " + p.getStock() +
                    ").");
                }
            } catch (NumberFormatException e) {
                System.out.println("Cantidad invalida.");
            }
        }
        pedido.mostrarTicket();
    }
}

```

Que demuestra este proyecto: composición (`Pedido`` tiene `LineaPedido`s` que tienen `Producto``), encapsulamiento (el stock solo cambia via `vender``), `static`` para folios consecutivos, `HashMap`` para búsqueda por código, validación en cada capa, y objetos colaborando: el `Pedido`` no toca el stock directamente, le pide al `Producto`` que venda. Este es código con arquitectura, no un script.

Extension natural: persistencia en archivos (guarda catálogo y pedidos en CSV), descuentos por cliente frecuente (herencia: `ClienteVIP`` extends `Cliente`` con 10% de descuento... polimorfismo en el calculo del total), y luego la base de datos con JDBC (siguiente sección).

Maven: el administrador de dependencias

Hasta ahora, todo tu código usa solo lo que trae Java. Pero los programas reales usan **librerías externas**: conectores de bases de datos, frameworks web, utilidades. Antes, conseguir una librería era un calvario manual: descargar el `.jar``, copiarlo a tu proyecto, rezar que fuera la versión correcta y que sus propias dependencias también estuvieran. **Maven** automatiza todo eso: declaras que necesitas y el lo descarga, lo conecta y maneja las versiones.

Instalación: descarga Maven de maven.apache.org, descomprimelo y agrega su carpeta `bin`` al PATH. Verifica con `mvn version``. (IntelliJ IDEA ya trae Maven integrado: puedes usarlo sin instalar nada.)

**El `pom.xml``:** cada proyecto Maven tiene un archivo `pom.xml`` (Project Object Model) que describe el proyecto y sus dependencias:

```

<project>
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.mitienda</groupId>
  <artifactId>tienda-app</artifactId>
  <version>1.0.0</version>

  <properties>
    <maven.compiler.source>17</maven.compiler.source>
    <maven.compiler.target>17</maven.compiler.target>
  </properties>

  <dependencies>
    <!-- Conector MySQL: lo veremos en la seccion JDBC -->
    <dependency>
      <groupId>com.mysql</groupId>
      <artifactId>mysql-connector-j</artifactId>
      <version>8.0.33</version>
    </dependency>
  </dependencies>
</project>

```

Cada dependencia se identifica con 3 coordenadas: `groupId` (quien la hizo), `artifactId` (cómo se llama) y `version` (cual). Con esa declaración, Maven descarga el conector MySQL de internet (del repositorio central Maven) y lo pone disponible para tu código. Sin Maven, estarías cazando el `.jar` a mano.

Comandos esenciales (desde la carpeta del proyecto, donde esta el `pom.xml`):

Comando	Que hace	--- ---	`mvn compile`	Compila el código		`mvn package`	Compila y empaqueta en un `.jar`		`mvn exec:java`	Ejecuta (con plugin exec)		`mvn dependency:tree`	Muestra el árbol de dependencias		`mvn clean`	Borra lo compilado
---------	----------	---------	---------------	-------------------	--	---------------	----------------------------------	--	-----------------	---------------------------	--	-----------------------	----------------------------------	--	-------------	--------------------

Estructura de carpetas estandar de un proyecto Maven:

```

tienda-app/
+-- pom.xml
+-- src/
    +-- main/
        +-- java/
            +-- com/mitienda/
                +-- TiendaApp.java
    +-- test/
        +-- java/

```

El código va en `src/main/java`, siguiendo la estructura del paquete (`com.mitienda`). Esta convención la usan millones de proyectos: cualquier programador Java entiende un proyecto Maven al instante.

Por qué importa para tu carrera: el 99% de los empleos Java usan Maven o Gradle (su primo moderno). "Se usar Maven" es una línea esperada en tu CV. Y sin el, no puedes usar JDBC comodamente, que es lo siguiente.

JDBC: Java habla con MySQL

Tu tienda guarda datos en CSV. Funciona, pero tiene límites: dos programas no pueden escribir a la vez sin corromper, buscar entre millones de registros es lento, y no hay seguridad ni respaldos. Las **bases de datos** resuelven todo eso, y **JDBC** (Java Database Connectivity) es el puente entre Java y las bases de datos.

Que necesitas:

1. MySQL instalado (descargalo de mysql.com, o usa XAMPP que lo incluye fácil).
2. Una base de datos y una tabla. En la consola de MySQL:

```

CREATE DATABASE tienda;
USE tienda;
CREATE TABLE productos (
    codigo VARCHAR(10) PRIMARY KEY,
    nombre VARCHAR(100),
    precio DOUBLE,
    stock INT
);

```

3. El conector MySQL declarado en tu `pom.xml` (el `dependency` de la sección anterior).

Conectar y consultar (los 5 pasos de JDBC):

```

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;

public class JdbcDemo {
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/tienda";
        String usuario = "root";
        String password = "tu_password";

        // try-with-resources cierra conexion, statement y resultset solos
        try (Connection con = DriverManager.getConnection(url, usuario, password);
            Statement st = con.createStatement();
            ResultSet rs = st.executeQuery("SELECT * FROM productos")) {

            while (rs.next()) {
                String codigo = rs.getString("codigo");
                String nombre = rs.getString("nombre");
                double precio = rs.getDouble("precio");
                int stock = rs.getInt("stock");
                System.out.println(codigo + " | " + nombre + " | $" + precio + " | " + stock);
            }

        } catch (Exception e) {
            System.out.println("Error de base de datos: " + e.getMessage());
        }
    }
}

```

Los 5 pasos, siempre en este orden:

1. `DriverManager.getConnection(url, usuario, password)`: abre la conexión. La URL dice el motor (`mysql`), el servidor (`localhost`), el puerto (`3306`) y la base (`tienda`).
2. `con.createStatement()`: crea el canal para enviar SQL.
3. `st.executeQuery(sql)`: ejecuta la consulta y devuelve un `ResultSet` (los resultados).
4. `rs.next()`: avanza fila por fila (empieza ANTES de la primera: el primer `next()` te pone en la fila 1). `getString`/`getDouble`/`getInt` leen columnas por nombre.
5. **Cerrar**: el try-with-resources lo hace solo, en orden inverso.

Insertar datos (usa `executeUpdate` para INSERT/UPDATE/DELETE):

```
String sql = "INSERT INTO productos (codigo, nombre, precio, stock) VALUES ('A04',
'Monitor', 4500, 8)";
int filas = st.executeUpdate(sql);
System.out.println("Filas insertadas: " + filas);
```

PreparedStatement: JAMÁS concatenes datos del usuario en el SQL. Esto es crítico:

```
// PELIGROSO: inyeccion SQL
String sql = "SELECT * FROM usuarios WHERE nombre = '" + nombreUsuario + "'";
// Si el usuario escribe: ' OR '1'='1 -> ve TODOS los usuarios. Hackeo clasico.

// CORRECTO: PreparedStatement con parametros (?)
String sql2 = "SELECT * FROM productos WHERE codigo = ?";
PreparedStatement ps = con.prepareStatement(sql2);
ps.setString(1, codigo); // el ? se reemplaza de forma SEGURA
ResultSet rs = ps.executeQuery();
```

El `PreparedStatement` separa el SQL de los datos: los datos jamás se interpretan como código. La **inyección SQL** es una de las vulnerabilidades más explotadas de la historia; con `PreparedStatement` quedas blindado. Además es más rápido si repites la consulta (la base la pre-compila).

Tu tienda con base de datos (el siguiente nivel): reemplaza el `HashMap` del catálogo por métodos que lean/escriban MySQL:

```
public static Producto buscarEnBD(Connection con, String codigo) throws Exception {
    String sql = "SELECT * FROM productos WHERE codigo = ?";
    try (PreparedStatement ps = con.prepareStatement(sql)) {
        ps.setString(1, codigo);
        try (ResultSet rs = ps.executeQuery()) {
            if (rs.next()) {
                return new Producto(codigo, rs.getString("nombre"),
                                    rs.getDouble("precio"), rs.getInt("stock"));
            }
        }
    }
    return null;
}
```

El objeto `Producto` se construye desde la fila de la base: tus clases POO + JDBC = sistema profesional completo. Este patrón (mapear filas a objetos) es lo que hacen los frameworks como Hibernate/JPA automáticamente; entenderlo a mano te hace entenderlos a ellos.

Práctica de la unidad

Los "ejercicios" de esta unidad son **extensiones de los proyectos**: elige al menos dos y terminalas. Son lo que pondras en tu portafolio.

Ejercicio 1. Agrega a la Calculadora: opción 6 "Raíz cuadrada" (pide un solo número; si es negativo, muestra error sin tronar) y opción 7 "Ver historial" (guarda cada operación como String en un `ArrayList<String>`, formato " $5 + 3 = 8$ ").

Ejercicio 2. Agrega a la Agenda la opción 5 "Editar": lista los contactos, pide el número, y permite cambiar teléfono y email (el nombre no se cambia: es el identificador). Guarda en el CSV después de editar.

Ejercicio 3. Agrega a la Tienda persistencia en CSV: al iniciar, carga el catálogo desde ``catalogo.csv`` (si existe; si no, usa los datos de ejemplo y lo crea). Cada venta actualiza el archivo. Así el stock sobrevive entre ejecuciones.

Ejercicio 4. Crea ``ClienteVIP`` extends `Cliente`` con atributo ``descuento`` (ej. 0.10) y modifica ``Pedido.calcularTotal()`` para aplicar el descuento si el cliente es VIP (usa ``instanceof``). Prueba con un cliente normal y uno VIP comprando lo mismo.

Ejercicio 5. Convierte la Calculadora en proyecto Maven: crea la estructura de carpetas, el ``pom.xml`` con Java 17, y compílala con ``mvn package``. Ejecuta el ``jar`` generado.

Ejercicio 6. Instala MySQL (o XAMPP), crea la base ``tienda`` y la tabla ``productos``, inserta 5 productos con SQL, y escribe un programa JDBC que los liste. Luego modifica la Tienda para que el catálogo se cargue desde la base de datos en vez de datos quemados.

Ejercicio 7 (reto final). Combina todo: la Tienda con base de datos MySQL (catálogo y pedidos en tablas), descuentos VIP con herencia, e historial de pedidos guardado. Este es un sistema de punto de venta simplificado pero real. Si lo terminas, tienes proyecto de portafolio nivel junior.

Graduacion y tu siguiente paso

Llegaste al final del contenido. Haz una pausa y mira atras: hace 10 unidades no sabías compilar un Hola Mundo. Hoy disenas clases, manejas colecciones, sobrevives a errores, guardas datos en archivos y bases de datos, corres hilos en paralelo y construyes sistemas con POO. Eso, en cualquier medida honesta, es **programar en Java**.

Pero seamos claros: este libro te dio la base solida, no el techo. La programación se domina construyendo, no leyendo. Tu misión ahora:

1. **Termina los 3 proyectos y subelos a GitHub.** Aprende Git (control de versiones) ya: es tan importante como Java para trabajar en equipo. Un GitHub con tus proyectos es tu carta de presentación.
2. **Aprende Spring Boot.** Es el framework que piden el 90% de los empleos Java. Con tu base de POO + colecciones + JDBC, Spring Boot te tomara semanas, no meses. En lucasegaacademy.com encontraras la ruta.
3. **Android usa Java.** Si te gustan las apps de celular, tu siguiente curso natural es Android: ya sabes el lenguaje, solo aprendes el framework.
4. **Práctica algoritmos.** Las entrevistas tecnicas preguntan lógica (los ejercicios de las Unidades 4-5 son el calentamiento). Plataformas como HackerRank o LeetCode te entrenan.
5. **Construye algo tuyo.** El mejor proyecto es el que resuelve TU problema: la app que tu negocio necesita, la herramienta que tu escuela pide. La motivación propia termina lo que los ejercicios no.

Felicidades, programador Java. Nos vemos en el siguiente curso.

Resumen de la unidad

- Proyecto 1 (Calculadora): menu ``do-while``, métodos por operación, validación con `try-catch`, historial opcional.
- Proyecto 2 (Agenda): CRUD completo con ``ArrayList``, búsqueda parcial, persistencia CSV automatica.
- Proyecto 3 (Tienda): POO real con ``Producto``, ``Cliente``, ``Pedido``, ``LineaPedido``; composición, encapsulamiento, ``static`` para folios, ``HashMap`` para catálogo.
- Maven: declara dependencias en ``pom.xml`` (`groupId/artifactId/versión`); comandos ``compile``, ``package``, ``clean``; estructura ``src/main/java``.
- JDBC en 5 pasos: ``getConnection`` -> ``createStatement`` -> ``executeQuery`/`executeUpdate`` -> recorrer ``ResultSet`` -> cerrar (`try-with-resources`).
- ``PreparedStatement`` con ``?`` SIEMPRE para datos del usuario: blindaje contra inyección SQL.
- Siguiente paso: Git/GitHub, Spring Boot, Android, algoritmos, y construir tus propios proyectos.

Mini glosario de la unidad

- **CRUD**: Create, Read, Update, Delete: las 4 operaciones basicas sobre datos.
- **Maven**: herramienta de construcción y gestion de dependencias de Java.
- **pom.xml**: archivo que describe un proyecto Maven y sus dependencias.
- **Dependencia**: libreria externa que tu proyecto necesita, gestionada por Maven.
- **JDBC**: API de Java para conectar con bases de datos relacionales.
- **ResultSet**: objeto con las filas resultado de una consulta SQL.
- **PreparedStatement**: consulta SQL precompilada con parámetros seguros.
- **Inyeccion SQL**: ataque que mete código SQL malicioso via entradas de usuario.
- **Portafolio**: conjunto de proyectos publicos (ej. GitHub) que demuestran tu nivel.

Maven profundo: scopes, árbol de dependencias y plugins

Scopes de dependencia: no todas las librerias se necesitan igual. El `scope` dice cuando se usa:

```
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter</artifactId>
  <version>5.10.0</version>
  <scope>test</scope>  <!-- solo para pruebas, no va al .jar final -->
</dependency>
```

| Scope | Significado | |---|---| | `compile` (por defecto) | Se necesita siempre | | `test` | Solo para pruebas (JUnit) | | `provided` | La necesita, pero el servidor la provee (ej. APIs de servidor) | | `runtime` | Solo al correr, no al compilar (ej. el conector MySQL) |

El árbol de dependencias: las librerias traen sus propias dependencias (transitivas). `mvn dependency:tree` te muestra todo el árbol. A veces dos librerias piden versiones distintas de la misma dependencia: Maven elige la más cercana a tu `pom` (estrategia "nearest wins"). Si algo se comporta raro, el árbol de dependencias es el primer lugar donde mirar.

Plugins: Maven hace todo mediante plugins. Dos que debes conocer:

```

<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.11.0</version>
      <configuration>
        <source>17</source>
        <target>17</target>
      </configuration>
    </plugin>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-shade-plugin</artifactId>
      <version>3.5.0</version>
      <!-- empaqueta las dependencias DENTRO del .jar (jar gordo/fat jar) -->
    </plugin>
  </plugins>
</build>

```

El shade plugin crea un "fat jar": tu código + todas las dependencias en un solo archivo ejecutable con `java -jar tienda.jar`. Así se distribuyen las apps Java reales.

Perfiles: puedes tener configuraciones distintas por entorno (desarrollo vs producción) con ``<profiles>``. Avanzado, pero ya sabes que existe.

JDBC profundo: transacciones

En un banco, transferir dinero son DOS operaciones: restar de una cuenta y sumar a otra. Si el programa se cae en medio, el dinero desaparece (o se duplica). Las **transacciones** resuelven esto: agrupan operaciones en una unidad atómica (todo o nada).

```

con.setAutoCommit(false); // apaga el modo automatico
try {
    // Operacion 1: restar de la cuenta origen
    PreparedStatement ps1 = con.prepareStatement(
        "UPDATE cuentas SET saldo = saldo - ? WHERE id = ?");
    ps1.setDouble(1, 500);
    ps1.setInt(2, 1);
    ps1.executeUpdate();

    // Operacion 2: sumar a la cuenta destino
    PreparedStatement ps2 = con.prepareStatement(
        "UPDATE cuentas SET saldo = saldo + ? WHERE id = ?");
    ps2.setDouble(1, 500);
    ps2.setInt(2, 2);
    ps2.executeUpdate();

    con.commit(); // todo salio bien: confirma los cambios
    System.out.println("Transferencia exitosa");
} catch (Exception e) {
    con.rollback(); // algo fallo: DESHACE todo lo hecho
    System.out.println("Transferencia cancelada, sin cambios: " + e.getMessage());
} finally {
    con.setAutoCommit(true);
}

```

`commit()` confirma, `rollback()` deshace. Sin transacciones no hay sistemas financieros serios. Cuando uses Spring Boot, las transacciones se manejan con una anotación (`@Transactional`); entender el mecanismo manual te hace entender la magia.

Batch: para insertar miles de registros, no ejecutes miles de `executeUpdate` (lento: un viaje a la base por cada uno). Agrupa en lote:

```

PreparedStatement ps = con.prepareStatement(
    "INSERT INTO productos (codigo, nombre, precio, stock) VALUES (?, ?, ?, ?)");
for (Producto p : lista) {
    ps.setString(1, p.getCodigo());
    ps.setString(2, p.getNombre());
    ps.setDouble(3, p.getPrecio());
    ps.setInt(4, p.getStock());
    ps.addBatch(); // acumula, no ejecuta aun
}
int[] resultados = ps.executeBatch(); // ejecuta TODO junto

```

Un solo viaje a la base para miles de inserts: de minutos a segundos.

Git y GitHub: tu portafolio profesional (introducción)

Git es el control de versiones: guarda el historial de cambios de tu código, te permite volver atrás, y trabajar en equipo sin pisarse. **GitHub** es la red social del código: ahí publicas tus proyectos y los empleadores los revisan.

Flujo básico (instalalo de git-scm.com):

```
git init                # convierte la carpeta en repositorio
git add .               # prepara todos los archivos
git commit -m "Calculadora terminada" # guarda la foto del momento
```

Y para subirlo a GitHub (crea la cuenta y el repositorio en github.com):

```
git remote add origin https://github.com/tu-usuario/calculadora-java.git
git push -u origin main
```

Comandos de todos los días: `git status` (que cambió?), `git diff` (muéstrame los cambios), `git log` (historial), `git checkout -- archivo` (deshaz cambios de un archivo). Aprende Git esta misma semana: cada proyecto de este libro merece su repositorio.

Que poner en tu GitHub para buscar trabajo junior:

- Los 3 proyectos de esta unidad, terminados y con README que explique que hacen y como correrlos.
- Un README de perfil que diga quien eres y que sabes.
- Commits frecuentes (muestran constancia) y código limpio (muestran profesionalismo).

Como seguir aprendiendo después de este libro (ruta concreta)

Mes 1-2: Spring Boot. El framework de backend Java. Con el construyes APIs REST (las que consumen las apps de celular y las páginas web). Tu base de POO + JDBC hace que Spring sea "JDBC con superpoderes". Recurso: spring.io/guides + el curso de Spring en lucasegaacademy.com.

Mes 2-3: Bases de datos a fondo. SQL avanzado (joins, subconsultas, índices) + JPA/Hibernate (el mapeo objeto-relacional automatico). El curso de SQL de la academia + práctica.

Mes 3-4: Frontend básico o Android. Si vas por backend: HTML/CSS/JS básico para entender a tus compañeros frontend. Si vas por móvil: Android con Java/Kotlin (ya sabes Java: es cuesta abajo).

Mes 4+: Proyecto grande. Un sistema completo: backend Spring + base de datos + frontend o app móvil. Un proyecto así en tu GitHub vale más que 10 cursos en tu CV.

Siempre: inglés técnico. La documentación, los errores y las mejores respuestas de Stack Overflow están en inglés. No necesitas ser bilingüe; necesitas leer documentación técnica. Práctica un poco cada día.

Checklist: de ejercicio a proyecto profesional

Antes de subir un proyecto a tu GitHub, pasalo por esta lista. La diferencia entre "código de estudiante" y "código contratable" está en estos detalles:

Funciona

- ☐ Corre sin errores desde cero (borra los .class y prueba).
- ☐ Maneja entradas invalidas sin tronar (letras donde van números, opciones fuera de rango).
- ☐ Los datos se guardan (si aplica): cierra y abre el programa y verifica que sigan ahí.

Código limpio

- ☐ Nombres descriptivos en variables, métodos y clases (nada de `x`, `temp2`, `cosa`).
- ☐ Métodos cortos que hacen una sola cosa.
- ☐ Sin código duplicado: lo repetido está en un método.
- ☐ Sin código muerto: borra lo comentado y lo que ya no se usa.

Documentacion

- ☐ README.md que explique: quéhace, como compilarlo/correrlo, y un ejemplo de uso.
- ☐ Comentarios solo donde el código no es obvio (el porqué, no el QUE).
- ☐ Javadoc en las clases y métodos publicos de proyectos grandes.

Versionado

- ☐ Repositorio Git con commits descriptivos ("Agrega búsqueda por nombre", no "cambios").
- ☐ .gitignore que excluya los .class y archivos del IDE.

Presentacion

- [] El README tiene capturas o ejemplo de sesión.
- [] Menciona las tecnologías (Java 17, Maven, MySQL...).
- [] Incluye ideas de mejora futura (muestra que piensas en el producto).

Un proyecto que cumple este checklist, aunque sea una agenda de consola, impresiona más que un proyecto ambicioso a medias. Los reclutadores revisan 30 segundos tu GitHub: qué esos segundos cuenten.

Preguntas frecuentes de la unidad

P: Maven o Gradle? Cual aprendo? R: Maven primero: es el estandar historico, su `pom.xml` es declarativo y fácil de leer, y la mayoría de empleos lo piden. Gradle después: más flexible y rápido, popular en Android. Saber Maven te hace entender Gradle en días.

P: `mvn package` falla con error de dependencias. Que hago? R: En orden: 1) revisa tu conexión a internet (Maven descarga del repositorio central), 2) verifica las coordenadas en el `pom.xml` (un typo en el artifactId = no encontrado), 3) corre `mvn clean` y reintenta, 4) mira el log completo: dice exactamente que artefacto falló.

P: Donde queda el `.jar` que genera Maven? R: En la carpeta `target/` de tu proyecto. `mvn package` compila a `target/classes` y empaqueta a `target/tu-artefacto-1.0.0.jar`.

P: Tengo que aprender SQL antes de JDBC? R: Lo básico si: `CREATE TABLE`, `INSERT`, `SELECT`, `UPDATE`, `DELETE`. JDBC solo es el puente; el que hace el trabajo es el SQL. El curso de SQL de Lucas EGA Academy te pone al día en una semana.

P: Mis proyectos de consola se ven "pobres" comparados con apps reales. Es normal? R: Totalmente. La consola es tu laboratorio, no tu escaparate. La lógica que aprendiste (POO, colecciones, JDBC) es IDENTICA a la que usan las apps con interfaz gráfica; la interfaz es solo la capa de arriba. Cuando aprendas Spring Boot o Android, conectaras esta misma lógica a una interfaz moderna y todo encajara.

APÉNDICE A: Referencia rápida de Java

Tu machete de consulta. No memorices todo: memoriza donde buscarlo.

Palabras clave reservadas

No puedes usarlas como nombres de variables, métodos o clases:

abstract	assert	boolean	break	byte
case	catch	char	class	const
continue	default	do	double	else
enum	extends	final	finally	float
for	goto	if	implements	import
instanceof	int	interface	long	native
new	package	private	protected	public
return	short	static	strictfp	super
switch	synchronized	this	throw	throws
transient	try	void	volatile	while

Las que más usaras: ``public``, ``private``, ``class``, ``static``, ``void``, ``new``, ``if``, ``else``, ``for``, ``while``, ``return``, ``extends``, ``import``, ``try``, ``catch``.

Precedencia de operadores (de mayor a menor)

1. Postfijo: ``x++``, ``x--``
2. Unario: ``++x``, ``--x``, ``!``, ``-x``
3. Multiplicativos: ``*``, ``/``, ``%``
4. Aditivos: ``+``, ``-``
5. Relacionales: ``<``, ``<=``, ``>``, ``>=``
6. Igualdad: ``==``, ``!=``
7. AND lógico: ``&&``
8. OR lógico: ``||``
9. Ternario: ``cond ? a : b``
10. Asignación: ``=``, ``+=``, ``-=``, ``*=``, ``/=``, ``%=``

En duda, usa paréntesis: ``(a + b) * c`` siempre es claro.

Métodos esenciales de String

Método	Ejemplo	Resultado
<code>length()</code>	<code>"Hola".length()</code>	4
<code>charAt(i)</code>	<code>"Hola".charAt(1)</code>	'o'
<code>equals(otro)</code>	<code>"a".equals("A")</code>	false
<code>equalsIgnoreCase(otro)</code>	<code>"a".equalsIgnoreCase("A")</code>	true
<code>toUpperCase()</code>	<code>"hola".toUpperCase()</code>	"HOLA"
<code>toLowerCase()</code>	<code>"HOLA".toLowerCase()</code>	"hola"
<code>trim()</code>	<code>" hola ".trim()</code>	"hola"
<code>contains(s)</code>	<code>"hola".contains("ol")</code>	true
<code>startsWith(s)</code>	<code>"hola".startsWith("ho")</code>	true
<code>endsWith(s)</code>	<code>"hola".endsWith("la")</code>	true
<code>indexOf(s)</code>	<code>"hola".indexOf("l")</code>	2
<code>substring(i, j)</code>	<code>"hola".substring(1, 3)</code>	"ol"
<code>split(sep)</code>	<code>"a,b".split(",")</code>	["a", "b"]
<code>replace(a, b)</code>	<code>"hola".replace('o', 'O')</code>	"hOla"
<code>isEmpty()</code>	<code>"".isEmpty()</code>	true

Clase Math

```
Math.abs(-5)           // 5
Math.pow(2, 3)         // 8.0
Math.sqrt(16)          // 4.0
Math.max(3, 7)         // 7
Math.min(3, 7)         // 3
Math.round(3.6)        // 4
Math.floor(3.9)        // 3.0
Math.ceil(3.1)         // 4.0
Math.random()          // double entre 0.0 y 1.0
Math.PI                // 3.14159...
```

Colecciones: que usar cuando

Necesidad	Estructura	Métodos clave
Lista ordenada, tamaño variable	<code>ArrayList<T></code>	<code>add</code> , <code>get(i)</code> , <code>set(i, v)</code> , <code>remove(i)</code> , <code>size()</code>
Conjunto sin duplicados	<code>HashSet<T></code>	<code>add</code> , <code>contains</code> , <code>remove</code> , <code>size()</code>
Llave -> valor	<code>HashMap<K, V></code>	<code>put</code> , <code>get</code> , <code>getOrDefault</code> , <code>containsKey</code> , <code>keySet()</code>
Arreglo fijo y rápido	<code>T[]</code>	<code>a[i]</code> , <code>a.length</code>
Cola (primero en entrar, primero en salir)	<code>LinkedList<T></code> / <code>Queue</code>	<code>offer</code> , <code>poll</code> , <code>peek</code>
Pila (último en entrar, primero en salir)	<code>Stack<T></code>	<code>push</code> , <code>pop</code> , <code>peek</code>

Scanner: lectura sin trampas

```
import java.util.Scanner;
Scanner sc = new Scanner(System.in);

String texto = sc.nextLine();           // línea completa
int n = Integer.parseInt(sc.nextLine()); // entero seguro
double d = Double.parseDouble(sc.nextLine()); // decimal seguro
sc.close();
```

printf: formato de salida

```
System.out.printf("Hola %s, tienes %d anos%n", nombre, edad);
System.out.printf("Precio: $%.2f%n", 19.99); // $19.99 (2 decimales)
System.out.printf("%-10s | %5d%n", "Ana", 95); // columnas alineadas
```

Marcadores: `%s` texto, `%d` entero, `%f` decimal, `%c` carácter, `%n` salto de línea portable.

Modificadores de acceso

Modificador	Misma clase	Mismo paquete	Subclase	Todos	---	---	---	---	---	private	Si
No	No	No	(por defecto)	Si	Si	No	No	protected	Si	Si	Si
Si	Si									public	Si
Si	Si										Si

Regla práctica: atributos `private`, métodos `public` (o `private` si son ayudantes internos).

Conversiones de tipos (casting)

De	A	Como	Se pierde algo?	---	---	---	---	int	double	Automatico	double d = 5;
No	double	int	Cast: int n = (int) 5.9;	Si: trunca a 5	int	String	String s = "" + 5;				
			Integer.toString(5)	No	String	int	Integer.parseInt("42")	Falla si no es número			
String	double	Double.parseDouble("3.14")	Falla si no es número		char	int					
Automatico	int n = 'A';	No (da el código ASCII: 65)	int	char	Cast: char c = (char) 65;	No ('A')	long	int	Cast: int n = (int) 9999999999L;	Si el valor no cabe, se corrompe	

Regla de oro: las conversiones "hacia arriba" (int -> double) son automáticas y seguras; las "hacia abajo" (double -> int) exigen cast porque puedes perder información. Java te obliga a escribir el cast para que el riesgo sea tu decisión consciente.

Archivos: resumen de operaciones

| Operación | Clases | Patrón | |---|---|---| | Escribir texto | ``FileWriter`` / ``PrintWriter`` |
try-with-resources + ``write`/`println`` | | Leer texto línea por línea | ``Scanner(File)`` |
``hasNextLine()`` + ``nextLine()`` | | Verificar existencia | ``File`` | ``new File(ruta).exists()`` | | Agregar
sin borrar | ``FileWriter`` | ``new FileWriter(ruta, true)`` | | Copiar binario | ``FileInputStream`` /
``FileOutputStream`` | buffer de bytes en ciclo | | Crear carpeta | ``File`` | ``new File(ruta).mkdirs()`` |
| Listar carpeta | ``File`` | ``new File(ruta).list()`` | | Borrar | ``File`` | ``new File(ruta).delete()`` |

Hilos: resumen de operaciones

| Operación | Como | |---|---| | Crear hilo | ``new Thread(() -> { ... })`` o clase que extiende
``Thread`` | | Iniciar paralelismo | ``hilo.start()`` (jamás ``run()`` directo) | | Esperar a que termine |
``hilo.join()`` | | Pausar | ``Thread.sleep(milis)`` dentro de try-catch | | Proteger recurso compartido
| ``synchronized`` en el método o bloque | | Pool de hilos | ``ExecutorService pool = Executors.newFixedThreadPool(4)`` |

APÉNDICE B: Catálogo de errores comunes

Los errores más frecuentes, que significan y como arreglarlos. Cuando veas uno, ven aquí antes de entrar en panico.

Errores de compilación

1. **`;' expected`** Significa: olvidaste el punto y coma al final de una instrucción. Arreglo: revisa la línea señalada y la anterior; el `;` falta ahí.
2. **`cannot find symbol`** Significa: usaste un nombre que no existe (variable no declarada, método mal escrito, import faltante). Arreglo: verifica la ortografía (Java distingue mayúsculas), que la variable este declarada antes de usarse, y que el `import` este presente.
3. **`class X is public, should be declared in a file named X.java`** Significa: el archivo no se llama igual que la clase publica. Arreglo: renombra el archivo a `X.java` exacto.
4. **`incompatible types: possible lossy conversion from double to int`** Significa: intentas guardar un `double` en un `int` sin cast. Arreglo: usa `(int)` si es intencional, o cambia el tipo de la variable.
5. **`non-static variable x cannot be referenced from a static context`** Significa: desde `main` (estático) intentas usar un atributo o método no estático sin crear objeto. Arreglo: crea el objeto primero (`MiClase obj = new MiClase(); obj.metodo();`) o declara el miembro `static`.
6. **`unreported exception IOException; must be caught or declared to be thrown`** Significa: una excepción checked sin try-catch ni `throws`. Arreglo: envuelve en try-catch (recomendado) o agrega `throws` a la firma.
7. **`method X in class Y cannot be applied to given types`** Significa: llamaste un método con número o tipo de argumentos incorrecto. Arreglo: revisa la firma del método y pasa los argumentos correctos.
8. **`missing return statement`** Significa: un método con retorno no devuelve valor en todos los caminos posibles. Arreglo: asegura un `return` en cada rama (`if`/`else`) o al final.

Errores de ejecución (excepciones)

9. **`NullPointerException`** Significa: usaste un objeto que vale `null` (no inicializado, o un `get` que devolvio null). Arreglo: inicializa antes de usar; verifica `!= null` cuando el valor puede no existir.
10. **`ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 5`** Significa: índice fuera del rango 0 a length-1. Arreglo: revisa la condición del ciclo (`i < a.length`, no `<=`).
11. **`NumberFormatException: For input string: "abc"`** Significa: `parseInt`/`parseDouble` recibió texto no numerico. Arreglo: valida con try-catch o verifica antes de convertir.
12. **`ArithmeticException: / by zero`** Significa: división entera entre cero. Arreglo: verifica el divisor antes de dividir.
13. **`InputMismatchException`** Significa: `nextInt()` recibió texto no numerico. Arreglo: lee con `nextLine()` + `parseInt` dentro de try-catch.
14. **`FileNotFoundException`** Significa: el archivo no existe en la ruta indicada. Arreglo: verifica la ruta y el directorio de trabajo; usa `File.exists()` para mensaje amigable.
15. **`ConcurrentModificationException`** Significa: modificaste una colección mientras la recorrias con for-each. Arreglo: usa `Iterator.remove()` o recolecta los cambios y aplicalos después del ciclo.

Errores lógicos (los peores: compilan y corren, pero dan mal resultado)

16. **Division entera accidental:** `5 / 2` da 2, no 2.5. Si esperabas decimales, un operando debe ser double.
17. **Comparar Strings con `==`:** compara referencias, no contenido. Usa `equals()`.
18. **Off-by-one:** el ciclo corre una vez de más o de menos. Verifica con un ejemplo pequeno a mano.
19. **Olvidar el `break` en switch:** la ejecución cae al siguiente case.
20. **El `nextLine()` fantasma:** después de `nextInt()`, el Enter queda en el buffer. Limpia con `nextLine()` extra o lee todo como texto.

Rutina de depuración universal: 1) lee el mensaje completo (dice la línea), 2) reproduce el error con el caso más pequeño posible, 3) imprime valores intermedios con ``println``, 4) explica tu código en voz alta línea por línea (el "método del patito de hule": al explicarlo, el bug se revela solo).

APÉNDICE C: 30 retos extra con soluciones

Retos rápidos para practicar después del examen. Tres por unidad. Intenta cada uno antes de leer su solución.

Retos de las Unidades 1-2

Reto 1. Escribe un programa que imprima tu nombre dentro de un marco de asteriscos. Solución: usa ``println`` con líneas de ``*``. Lo importante es contar los caracteres para que el marco cierre parejo:

```
System.out.println("*****");
System.out.println("*   Ana   *");
System.out.println("*****");
```

Reto 2. Pide al usuario su año de nacimiento y calcula su edad (usa 2026 como año actual). Solución:

```
int anio = Integer.parseInt(sc.nextLine());
int edad = 2026 - anio;
System.out.println("Tienes " + edad + " años.");
```

Reto 3. Convierte grados Celsius a Fahrenheit ($F = C * 9/5 + 32$). Cuidado con la división entera. Solución: ``double f = c * 9.0 / 5 + 32``. El ``9.0`` fuerza la división decimal. Con ``9/5`` (enteros) daría 1 y el resultado saldría mal: la trampa clásica de la Unidad 2.

Retos de las Unidades 3-4

Reto 4. Pide una calificación (0-100) e imprime "Aprobado" (≥ 60) o "Reprobado". Solución: un ``if-else`` simple. Valida el rango 0-100 con un ``if`` previo que rechace valores fuera de rango.

Reto 5. Imprime los múltiplos de 3 del 1 al 30 usando ``for`` y ``%``. Solución:

```
for (int i = 1; i <= 30; i++) {
    if (i % 3 == 0) System.out.print(i + " ");
}
```

Reto 6. Pide números hasta que el usuario escriba 0, y al final muestra cuantos fueron pares. Solución: ``while`` con centinela 0 (el guardián), contador que aumenta cuando ``n % 2 == 0``. El 0 no se cuenta: verificalo antes de procesar.

Retos de la Unidad 5

Reto 7. Escribe el método ``mayor(int[] a)`` que devuelva el valor más grande del arreglo. Solución: inicializa con ``a[0]``, recorre desde 1 comparando. Si el arreglo puede venir vacío, decide que devolver (0 o lanzar excepción) y documentalo.

Reto 8. Invierte un arreglo en su lugar (sin crear otro): intercambia ``a[i]`` con ``a[n-1-i]``. Solución: ciclo hasta la mitad con variable temporal para el intercambio. El algoritmo clásico de dos apuntadores.

Reto 9. Cuenta cuantas veces aparece un valor X en un arreglo. Solución: recorre, compara con ``==``, acumula. Es el patrón "recorrer y contar" en su forma más pura.

Retos de la Unidad 6

Reto 10. Crea la clase ``Circulo`` con radio privado, constructor validado y método ``area()``. Solución: ``area()`` devuelve ``Math.PI * radio * radio``. El setter rechaza radios ``<= 0``. Encapsulamiento básico pero completo.

Reto 11. Agrega a ``Circulo`` el método ``esMasGrandeQue(Circulo otro)`` que compare areas. Solución: ``return this.area() > otro.area();``. Comparar objetos entre si es una habilidad POO fundamental.

Reto 12. Crea la clase ``Estudiante`` con nombre, matricula y promedio; el constructor rechaza promedios fuera de 0-10. Solución: validación en el constructor via el setter, como en la Unidad 6. Si el promedio es invalido, dejalo en 0 e imprime advertencia.

Retos de la Unidad 7

Reto 13. Crea la jerarquia ``Empleado`` -> ``Gerente`` (bono) y ``Vendedor`` (comisión), cada uno con método ``salarioTotal()`` sobrescrito. Solución: el padre guarda ``salarioBase``; cada hija suma su extra. Un arreglo ``Empleado[]`` polimorfico suma la nomina total llamando ``salarioTotal()`` sin saber el tipo.

Reto 14. Define la interfaz `Descontable` con método `aplicarDescuento(double porcentaje)` e implementala en una clase `Producto`. Solución: la interfaz declara el QUE, la clase define el COMO. Otra clase no relacionada (ej. `Servicio`) podría implementarla después.

Reto 15. Sobrescribe `toString()` en tus clases del reto 13 para que impriman nombre y salario total. Solución: `@Override public String toString() { return nombre + ": $" + salarioTotal(); }`. Así `System.out.println(empleado)` muestra algo útil.

Retos de la Unidad 8

Reto 16. Con un `ArrayList<String>`, elimina todas las palabras de menos de 4 letras. Solución: usa `removeIf(p -> p.length() < 4)`. La versión con for-each + `remove` lanzaría `ConcurrentModificationException`.

Reto 17. Dado un texto, cuenta cuantas palabras distintas hay (usa `HashSet`). Solución: `split(" ")`, mete cada palabra en minúsculas al set, imprime `set.size()`. El set elimina duplicados solo.

Reto 18. Invierte un `HashMap<String, Integer>` (nombre->edad) a `HashMap<Integer, String>` (edad->nombre). Solución: recorre `entrySet()` del original y haz `invertido.put(entrada.getValue(), entrada.getKey())`. Ojo: si dos personas tienen la misma edad, una se pierde (las llaves son únicas).

Retos de la Unidad 9

Reto 19. Escribe un método que divida dos enteros y devuelva -1 si el divisor es cero, sin lanzar excepción. Solución: `if (b == 0) return -1; return a / b;`. A veces es mejor prevenir que atrapar: la excepción es para lo inesperado, no para el flujo normal.

Reto 20. Lee un archivo CSV de productos y crea un `ArrayList` con cada línea como elemento; si el archivo no existe, devuelve lista vacía. Solución: `File.exists()` primero; try-with-resources con `Scanner`; `hasNextLine()/nextLine()`. Combina las técnicas de la unidad en un método reutilizable.

Reto 21. Crea 3 hilos que impriman "Hilo 1", "Hilo 2", "Hilo 3" cada segundo, 3 veces cada uno. Solución: un método que reciba el nombre y lo use en un `Runnable`; `start()` en los tres; `join()` para esperarlos. Observa el orden mezclado: prueba de que corren en paralelo.

Retos de la Unidad 10

Reto 22. Convierte tu Calculadora en proyecto Maven con ``mvn package`` y ejecuta el jar. Solución: estructura ``src/main/java``, ``pom.xml`` con ``maven.compiler.source/target`` 17, ``mvn package``, ``java -jar target/*.jar``. Si falla, ``mvn dependency:tree`` y los logs te dicen que paso.

Reto 23. Escribe el ``INSERT`` con ``PreparedStatement`` para agregar un producto leído por teclado. Solución:

```
PreparedStatement ps = con.prepareStatement(
    "INSERT INTO productos (codigo, nombre, precio, stock) VALUES (?, ?, ?, ?)");
ps.setString(1, codigo); ps.setString(2, nombre);
ps.setDouble(3, precio); ps.setInt(4, stock);
ps.executeUpdate();
```

Los ``?`` en orden, los ``set`` en el mismo orden. Jamás concatenes los valores.

Reto 24. Agrega a la Tienda el método ``valorTotalInventario()`` que sume `precio*stock` de todo el catálogo. Solución: recorre ``catalogo.values()``, acumula ``p.getPrecio() * p.getStock()``. Es el mismo patrón "recorrer y acumular" de la Unidad 5, ahora sobre objetos.

Retos integradores (todo el libro)

Reto 25. Sistema de calificaciones: lee N alumnos (nombre + 3 calificaciones) con Scanner, guardalos en objetos ``Alumno`` dentro de un ``ArrayList``, calcula el promedio de cada uno con un método, y guarda el reporte en ``reporte.txt``. Solución: combina Unidades 2 (Scanner), 5 (ArrayList, métodos), 6 (clase Alumno) y 9 (escritura de archivo). Si lo haces sin mirar, estas listo para proyectos reales.

Reto 26. Buscador de contactos: un ``HashMap<String, String>`` (nombre->teléfono) cargado desde ``contactos.csv``, con menu para buscar y agregar. Al salir, guarda los cambios. Solución: la Agenda de la Unidad 10 simplificada con mapa en vez de lista de objetos. ``containsKey`` para buscar, ``put`` para agregar, reescritura del CSV al salir.

Reto 27. Juego de dados contra la computadora: 3 rondas, cada uno tira 2 dados (``Random``), gana la ronda quien saque más. Lleva el marcador y declara al ganador final. Solución: ``random.nextInt(6) + 1`` por dado; ciclo ``for`` de 3 rondas; contadores de victorias; ``if-else`` final. Puro control de flujo + aleatorios.

Reto 28. Validador de password: pide una contraseña y verifica que tenga al menos 8 caracteres, una mayúscula, un número y un símbolo. Repite hasta que sea valida. Solución:

guardián ``do-while``; recorre con ``charAt`` usando ``Character.isUpperCase``, ``isDigit``; el símbolo es "lo que no es letra ni dígito". Un validador real en miniatura.

Reto 29. FizzBuzz POO: clase ``FizzBuzz`` con método ``evaluar(int n)`` que devuelva "Fizz" (múltiplo de 3), "Buzz" (múltiplo de 5), "FizzBuzz" (ambos) o el número. Un ``main`` lo prueba del 1 al 20. Solución: el orden de los ``if`` importa (primero el caso ambos). Es el ejercicio más famoso de las entrevistas: hazlo dormido.

Reto 30. Tu proyecto libre: elige un problema TUYO (tus gastos, tus tareas, tu negocio) y construye la versión 1 en Java con menú, persistencia y al menos 3 clases. Subelo a GitHub. Solución: no hay solución única; hay un programador nuevo. Este reto no se califica: se vive.

APÉNDICE D: Glosario general

Todos los terminos del libro, de la A a la Z, en un solo lugar.

- **Algoritmo:** secuencia de pasos para resolver un problema.
- **API:** interfaz que permite a un programa usar funcionalidades de otro.
- **Arreglo:** colección de tamaño fijo de elementos del mismo tipo.
- **Atributo:** variable que guarda el estado de un objeto.
- **Autoboxing:** conversión automática entre primitivos y sus clases envoltorio.
- **Biblioteca (librería):** código reutilizable creado por terceros.
- **Booleano:** tipo de dato que solo puede ser `true` o `false`.
- **Buffer:** zona de memoria temporal; también, datos pendientes de leer en el Scanner.
- **Bug:** error en el programa.
- **Bytecode:** código intermedio `.class` que ejecuta la JVM.
- **Cast:** conversión explícita de un tipo a otro, ej. `(int) 3.9`.
- **Checked exception:** excepción que el compilador obliga a manejar.
- **Clase:** molde para crear objetos.
- **Clase abstracta:** clase que no se puede instanciar y puede tener métodos abstractos.
- **Colección:** objeto que agrupa otros objetos (List, Set, Map).
- **Compilador:** programa que traduce código fuente a bytecode (`javac`).
- **Composicion:** relación "tiene un" entre clases.
- **Condición de carrera:** bug por acceso no coordinado entre hilos.
- **Constante:** valor que no cambia; se declara con `final`.
- **Constructor:** método especial que crea e inicializa objetos.
- **CRUD:** Create, Read, Update, Delete.
- **Depuración:** proceso de encontrar y corregir errores.
- **Dependencia:** librería externa gestionada por Maven.
- **Diagrama de clases:** representación gráfica de clases y sus relaciones.
- **Encapsulamiento:** proteger los datos con `private` y exponerlos con métodos.

- **Excepción:** evento anormal que interrumpe el flujo normal.
- **Framework:** plataforma de código que estructura aplicaciones completas (ej. Spring).
- **Garbage collector:** sistema automatico que libera memoria de objetos sin uso.
- **Generics:** tipos parametrizados como ``ArrayList<String>``.
- **Getter/Setter:** métodos para leer/modificar atributos privados.
- **Herencia:** relación "es un" donde la hija reutiliza código del padre.
- **Hilo (thread):** línea de ejecución independiente dentro de un programa.
- **IDE:** entorno de desarrollo integrado (IntelliJ, Eclipse, VS Code).
- **Índice:** posición de un elemento dentro de un arreglo (empieza en 0).
- **Inmutabilidad:** propiedad de objetos que no pueden cambiar (ej. String).
- **Inyeccion SQL:** ataque que mete código SQL via entradas de usuario.
- **Instancia:** objeto concreto creado a partir de una clase.
- **Interfaz:** contrato de métodos que una clase se compromete a implementar.
- **Iteración:** cada vuelta de un ciclo.
- **JAR:** archivo empaquetado con clases Java listas para distribuir.
- **JDBC:** API de Java para conectar con bases de datos.
- **JDK:** kit de desarrollo de Java (compilador + JRE).
- **JRE:** entorno de ejecución de Java (JVM + librerías).
- **JVM:** máquina virtual que ejecuta el bytecode.
- **Lambda:** función anonima compacta, ej. ``x -> x * 2``.
- **Logging:** registro sistematico de eventos con niveles de gravedad.
- **Maven:** herramienta de construcción y gestion de dependencias.
- **Método:** función que pertenece a una clase u objeto.
- **Módulo:** operador ``%`` que devuelve el residuo de una división.
- **Null:** valor que indica "ningun objeto".
- **Objeto:** instancia de una clase con estado y comportamiento.
- **Off-by-one:** error de una unidad en limites de ciclos o índices.
- **Paquete:** organización de clases en carpetas con namespace.

- **Parámetro/Argumento:** variable declarada en la firma / valor pasado al llamar.
- **Persistencia:** guardar datos para que sobrevivan al fin del programa.
- **Polimorfismo:** un mismo método se comporta distinto según el objeto real.
- **pom.xml:** descriptor de proyecto Maven.
- **PreparedStatement:** consulta SQL precompilada con parámetros seguros.
- **Refactorizar:** reestructurar código sin cambiar su comportamiento.
- **ResultSet:** filas resultado de una consulta JDBC.
- **Sentinela:** valor especial que termina la entrada de datos (ej. 0 o "fin").
- **Sobrecarga:** mismo nombre de método con distintos parámetros.
- **Sobrescritura:** la hija redefine un método del padre con la misma firma.
- **Stack trace:** rastro impreso con la cadena de llamadas de un error.
- **String:** cadena de texto inmutable.
- **Transacción:** conjunto de operaciones de base de datos que es todo o nada.
- **Try-with-resources:** try que cierra recursos automáticamente.
- **Tipo primitivo:** dato básico (`int`, `double`, `char`, `boolean`...).
- **Variable:** contenedor con nombre para un valor.
- **Wrapper:** clase que envuelve un primitivo (`Integer`, `Double`...).
- **Anotacion:** marca con `@` que da información al compilador o framework (ej. `@Override`).
- **Assertion:** afirmación que debe cumplirse; si falla, el programa truena (para pruebas).
- **Backend:** parte del sistema que procesa datos en el servidor.
- **Branch (rama):** línea de trabajo independiente en Git.
- **Breakpoint:** marca en el depurador donde la ejecución se detiene.
- **CamelCase:** convención de nombres `miVariable`, `MiClase`.
- **Classpath:** ruta donde Java busca las clases y librerías.
- **Commit:** foto guardada de los cambios en Git.
- **Compilación condicional:** código que se incluye/excluye según configuración.
- **Deploy:** poner una aplicación en producción.
- **Depurador (debugger):** herramienta para ejecutar paso a paso e inspeccionar variables.

- **DTO:** objeto simple para transportar datos entre capas.
- **Enum:** tipo con un conjunto fijo de valores (ej. días de la semana).
- **Expresión:** combinación de valores y operadores que produce un resultado.
- **Factory:** patrón que crea objetos sin exponer la lógica de creación.
- **Frontend:** parte visible del sistema con la que interactúa el usuario.
- **Hotspot:** la JVM de Oracle/OpenJDK, optimiza el código en caliente.
- **Indentación:** sangría del código para mostrar estructura.
- **Javadoc:** comentarios especiales que generan documentación HTML.
- **JSON:** formato ligero de intercambio de datos.
- **LTS:** versión con soporte a largo plazo (ej. Java 17, 21).
- **Namespace:** espacio de nombres que evita colisiones (los paquetes).
- **Null-safety:** técnicas para evitar NullPointerException.
- **POJO:** objeto Java simple, sin dependencias de framework.
- **Pull request:** propuesta de cambios en Git para revisión.
- **Refactor:** ver refactorizar.
- **REST:** estilo de arquitectura para APIs web con HTTP.
- **Scope:** alcance donde una variable es visible.
- **SDK:** kit de desarrollo de software (el JDK es el SDK de Java).
- **Sentencia:** instrucción completa terminada en `;`.
- **Singleton:** patrón que garantiza una sola instancia de una clase.
- **SOAP:** protocolo más formal y verboso para servicios web.
- **Stack:** pila de llamadas de métodos; también estructura LIFO.
- **Heap:** zona de memoria donde viven los objetos.
- **Terminal/Consola:** ventana de texto para comandos.
- **Token:** pieza léxica del código (identificador, operador...).

APÉNDICE E: Instalación paso a paso (JDK, IntelliJ, MySQL, Maven)

Todo lo que necesitas instalado para sacarle jugo al libro, explicado sin asumir nada.

Instalar el JDK 17

Windows:

1. Ve a adoptium.net y descarga "Temurin 17 (LTS)" para Windows x64 (el `.msi`).
2. Ejecuta el instalador. En las opciones, marca "Set JAVA_HOME variable" y "Add to PATH".
3. Abre una terminal NUEVA (cmd o PowerShell) y escribe ``java --version`` y ``javac --version``. Ambas deben responder con la versión 17.

Mac:

1. Descarga Temurin 17 para macOS (`.pkg`) e instalalo con doble clic.
2. Abre la Terminal y verifica con ``java --version``.

Linux (Ubuntu/Debian):

```
sudo apt update
sudo apt install openjdk-17-jdk
java --version
```

Si ``java`` funciona pero ``javac`` no: instalaste el JRE en vez del JDK. Repite la descarga eligiendo el JDK completo.

Instalar IntelliJ IDEA (recomendado)

1. Ve a jetbrains.com/idea y descarga la edición **Community** (gratis).
2. Instalalo con las opciones por defecto.
3. Al abrirlo: "New Project" -> lenguaje Java -> selecciona el JDK 17 (si no aparece, dale al engrane y buscalo).
4. Crea un archivo en ``src``, escribe tu código, y corre con el triángulo verde. IntelliJ compila solo: te olvidas de ``javac``.

Alternativas: VS Code con la extensión "Extension Pack for Java" (ligero y gratis), o Eclipse (clásico). Cualquiera sirve; lo importante es que compiles y corras.

Instalar MySQL (para la Unidad 10)

Opción fácil (recomendada): XAMPP (apachefriends.org):

1. Descarga XAMPP para tu sistema e instalalo.
2. Abre el panel de control de XAMPP e inicia el módulo **MySQL** (y Apache si quieres).
3. Abre `http://localhost/phpmyadmin` en tu navegador: ahí creas la base `tienda` y la tabla `productos` con clics, sin escribir SQL a mano (aunque te recomiendo practicar el SQL del libro).

Opción directa: descarga el instalador de MySQL Community de mysql.com. Durante la instalación anota la contraseña del usuario `root`: la necesitaras en el código JDBC.

Verifica la conexión con el programa `JdbcDemo` de la Unidad 10 antes de seguir: si lista los productos, todo está bien.

Instalar Maven

1. Ve a maven.apache.org/download.cgi y descarga el `.zip` binario.
2. Descomprimelo en una carpeta sin espacios (ej. `C:\maven` en Windows, `~/maven` en Mac/Linux).
3. Agrega la carpeta `bin` al PATH del sistema.
4. Verifica en una terminal nueva: `mvn --version`.

Atajo: si usas IntelliJ IDEA, ya trae Maven integrado (clic derecho en el `pom.xml` -> "Add as Maven Project"). Puedes saltarte la instalación manual hasta que la necesites en terminal.

Estructura de carpetas sugerida para el curso

```
java-curso/  
+-- unidad01/  
+-- unidad02/  
...  
+-- proyectos/  
    +-- calculadora/  
    +-- agenda/  
    +-- tienda/          <- aqui ira el proyecto Maven con pom.xml
```

Una carpeta por unidad y una para proyectos. Cuando algo no compila, lo primero que revisas es en que carpeta estas parado: el 90% de los "no funciona" son eso.

Configuración recomendada del IDE

Un IDE bien configurado te ahorra horas. Ajustes que recomiendo en IntelliJ IDEA (o tu IDE favorito):

Codificación: verifica que los archivos usen UTF-8 (Settings -> Editor -> File Encodings). Evita que los acentos se vean raros al compartir código.

Formato automatico: aprende el atajo de reformatear (`Ctrl+Alt+L` en IntelliJ, `Shift+Alt+F` en VS Code). Código bien indentado se lee solo; código desordenado esconde bugs.

Inspecciones: activa las advertencias del IDE (variables sin usar, posibles null, casts innecesarios). Son como un mentor que revisa tu código en tiempo real: hazles caso.

Plantillas en vivo (live templates): escribe `sout` y presiona Tab: se expande a `System.out.println()`. `psvm` -> el `main` completo. `fori` -> el ciclo for clásico. Memoriza 5 plantillas y tu velocidad se duplica.

Depurador: aprende a poner un breakpoint (clic en el margen izquierdo), correr en modo debug, y avanzar paso a paso viendo variables. Es 10 veces más rápido que llenar el código de `println` para encontrar un bug.

Terminal integrada: usa la terminal del IDE (`Alt+F12` en IntelliJ) para compilar con `javac`/`mvn` sin salir del editor. Alternar entre ventanas rompe tu concentración.

Plugins utiles: Key Promoter X (te ensena los atajos mientras trabajas), Rainbow Brackets (colorea paréntesis anidados), y el de Git integrado que ya trae.

Recursos recomendados para seguir

- **Documentacion oficial:** docs.oracle.com/javase (la referencia definitiva; en ingles, pero es el ingles técnico que necesitas practicar).
- **Lucas EGA Academy** (lucasegaacademy.com): los cursos que siguen a este libro (Spring Boot, SQL avanzado, Android).
- **Práctica de algoritmos:** HackerRank y LeetCode (empieza por los fáciles; 2-3 por semana).
- **Preguntas y respuestas:** Stack Overflow (busca antes de preguntar; el 99% de los errores ya tienen respuesta).
- **Código real:** GitHub: lee proyectos pequenos en Java, no solo escribas el tuyo. Leer código ajeno te ensena más patrones que cualquier libro.
- **Comunidad en espanol:** grupos de programadores Java en Discord, Telegram y Reddit ([r/learnjava](https://www.reddit.com/r/learnjava)). Preguntar bien (con código mínimo que reproduzca el problema) te da respuestas rápidas.

EXAMEN FINAL

Instrucciones

Este examen tiene 50 preguntas que cubren las 10 unidades del libro. Hay tres secciones:

- **Preguntas 1-30:** opción múltiple (una sola respuesta correcta).
- **Preguntas 31-40:** verdadero o falso.
- **Preguntas 41-50:** abiertas y de código (escribe la respuesta o el programa).

Como hacerlo: busca un lugar tranquilo, aparta 90 minutos, y responde TODO sin ver las soluciones. Puedes usar papel para los ejercicios de código, pero no el libro ni internet: la gracia es medir lo que realmente se te quedó. Anota tus respuestas y luego compáralas con la sección "Respuestas y soluciones", donde cada una viene explicada.

Como calificarte:

- 45-50: excelente. Eres programador Java sólido. A construir proyectos.
- 40-44: muy bien. Repasa las unidades donde fallaste.
- 35-39: bien, pero hay huecos. Relee las secciones de tus errores.
- Menos de 35: repasa el libro completo antes de seguir. No es derrota: es diagnóstico.

Sin calculadora especial ni trucos. Empieza.

Preguntas de opción múltiple

PREGUNTA 1. Cual es el orden correcto del ciclo de vida de un programa Java?

A) Ejecutar, compilar, escribir B) Escribir, compilar, ejecutar C) Compilar, escribir, ejecutar D) Escribir, ejecutar, compilar

PREGUNTA 2. Que contiene el JDK?

A) Solo la máquina virtual (JVM) B) El JRE más el compilador javac y herramientas de desarrollo C) Solo el editor de código D) Unicamente las librerías graficas

PREGUNTA 3. Si tu clase publica se llama `Factura`, como debe llamarse el archivo?

A) factura.java B) Factura.java C) FACTURA.java D) Como quieras, el nombre no importa

PREGUNTA 4. Que imprime este código?

```
System.out.print("Hola");  
System.out.println("Mundo");  
System.out.print("Java");
```

A) HolaMundoJava (todo en una línea) B) Hola / MundoJava (en dos líneas) C) HolaMundo / Java (en dos líneas) D) Cada palabra en su propia línea

PREGUNTA 5. Cual es el resultado de `7 / 2` en Java si ambos son `int`?

A) 3.5 B) 3 C) 4 D) Error de compilación

PREGUNTA 6. Que valor tiene `x` después de ejecutar `int x = 17 % 5;`?

A) 3 B) 2 C) 3.4 D) 0

PREGUNTA 7. Cual es la forma CORRECTA de comparar dos Strings `a` y `b` por su contenido?

A) `if (a == b)` B) `if (a = b)` C) `if (a.equals(b))` D) `if (a === b)`

PREGUNTA 8. Qué hace `Scanner sc = new Scanner(System.in);`?

A) Crea un archivo llamado System.in B) Conecta un lector de datos al teclado C) Imprime texto en pantalla D) Convierte texto a número

PREGUNTA 9. Un usuario escribe `25` y presiona Enter. Con `int edad = sc.nextInt();` seguido de `sc.nextLine()`, que pasa con el `nextLine()`?

A) Espera a que el usuario escriba su nombre normalmente B) Lee el Enter pendiente y devuelve un texto vacío C) Da error de compilación D) Repite la pregunta de la edad

PREGUNTA 10. Cual es el resultado de `boolean r = (5 > 3) && (2 > 4);`?

A) true B) false C) Error de compilación D) null

PREGUNTA 11. Cuantas veces se ejecuta el bloque en este ciclo?

```
for (int i = 0; i < 10; i++) {  
    System.out.println(i);  
}
```

A) 9 B) 10 C) 11 D) Infinitas

PREGUNTA 12. Que imprime este código?

```
int i = 1;
while (i < 4) {
    System.out.print(i + " ");
    i += 2;
}
```

A) 1 2 3 B) 1 3 C) 1 3 5 D) 2 4

PREGUNTA 13. En un `switch`, que pasa si olvidas el `break` en un `case`?

A) Error de compilación B) El programa se detiene C) La ejecución cae al siguiente `case` (fall-through) D) Se ejecuta el `default` automáticamente

PREGUNTA 14. Dado `int[] a = {10, 20, 30};`, cual es el ÚLTIMO índice valido?

A) 3 B) 2 C) 30 D) a.length

PREGUNTA 15. Que imprime este código?

```
int[] n = {5, 10, 15};
System.out.println(n[1] + n[2]);
```

A) 15 B) 25 C) 30 D) Error: índice fuera de rango

PREGUNTA 16. Qué hace `return` dentro de un método?

A) Repite el método desde el inicio B) Devuelve un valor y termina el método inmediatamente C) Imprime el valor en pantalla D) Guarda el valor en un archivo

PREGUNTA 17. En `public static int sumar(int a, int b)`, que son `a` y `b`?

A) Argumentos B) Parámetros C) Atributos D) Variables globales

PREGUNTA 18. Si un método recibe un `int` y lo modifica dentro, que pasa con la variable original?

A) También se modifica B) No se modifica (se paso una copia del valor) C) Se duplica D) Da error de compilación

PREGUNTA 19. Qué significa `private` en un atributo?

A) Que es gratis B) Que solo se puede acceder desde dentro de la misma clase C) Que no se puede modificar nunca D) Que es invisible para el programador

PREGUNTA 20. Cual es la firma correcta de un constructor para la clase `Libro`?

A) ``public void Libro(String t)`` B) ``public Libro(String titulo)`` C) ``public libro(String titulo)`` D) ``public String Libro(String titulo)``

PREGUNTA 21. Que palabra clave crea la relación de herencia en Java?

A) implements B) extends C) inherits D) super

PREGUNTA 22. Una clase hija quiere llamar al constructor de su padre. Como lo hace?

A) ``padre(...)`` B) ``super(...)`` como primera línea del constructor C) ``this(...)`` en cualquier línea D) No se puede

PREGUNTA 23. Qué es el polimorfismo?

A) Tener muchos constructores B) Una variable del tipo padre que ejecuta el método de la hija según el objeto real C) Usar muchos ``if`` anidados D) Crear varias clases con el mismo nombre

PREGUNTA 24. Para que sirve la anotación ``@Override``?

A) Es obligatoria para que el código compile B) Le dice al compilador que estas sobreescribiendo a proposito y te avisa si la firma está mal C) Hace el método más rápido D) Convierte el método en estático

PREGUNTA 25. Cual es la diferencia entre sobrecarga y sobrescritura?

A) No hay diferencia B) Sobrecarga: mismo nombre, distintos parámetros, misma clase. Sobrescritura: misma firma, la hija redefine el método del padre C) Sobrecarga solo funciona con Strings D) Sobrescritura solo funciona con constructores

PREGUNTA 26. Una clase abstracta...

A) Se puede instanciar con ``new`` normalmente B) No se puede instanciar; puede tener métodos abstractos que las hijas deben implementar C) No puede tener métodos con código D) No puede tener constructor

PREGUNTA 27. Cuantas interfaces puede implementar una clase en Java?

A) Solo una B) Ninguna C) Múltiples D) Máximo dos

PREGUNTA 28. Que método se usa para agregar un elemento al final de un ``ArrayList<String>``?

A) ``add()`` B) ``push()`` C) ``insert()`` D) ``append()``

PREGUNTA 29. Dado ``HashMap<String, Integer> m``, que devuelve ``m.get("clave_inexistente")``?

A) 0 B) Error de compilación C) ``null`` D) Una excepción

PREGUNTA 30. Para que sirve ``getOrDefault(llave, 0)`` en el patrón contador?

A) Borra la llave B) Devuelve el valor actual o 0 si la llave no existe C) Ordena el mapa D) Duplica el valor

PREGUNTA 31. En un bloque ``try-catch``, si ocurre una excepción dentro del ``try`` y hay un ``catch`` compatible, el programa continua después del bloque en vez de tronar.

A) Verdadero B) Falso

PREGUNTA 32. El bloque ``finally`` solo se ejecuta si NO hubo excepción.

A) Verdadero B) Falso

PREGUNTA 33. ``new FileWriter("datos.txt")`` agrega texto al final del archivo si ya existe, sin borrarlo.

A) Verdadero B) Falso

PREGUNTA 34. Para iniciar la ejecución paralela de un hilo se debe llamar a su método ``run()`` directamente.

A) Verdadero B) Falso

PREGUNTA 35. La palabra ``synchronized`` en un método evita que dos hilos lo ejecuten al mismo tiempo sobre el mismo objeto.

A) Verdadero B) Falso

PREGUNTA 36. En Maven, las dependencias se declaran en el archivo ``pom.xml`` con sus coordenadas `groupId`, `artifactId` y versión.

A) Verdadero B) Falso

PREGUNTA 37. Es seguro concatenar lo que escribe el usuario directamente dentro de un SQL, siempre que uses ``Statement``.

A) Verdadero B) Falso

PREGUNTA 38. ``PreparedStatement`` con parámetros ``?`` protege contra inyección SQL.

A) Verdadero B) Falso

PREGUNTA 39. Una clase puede extender de dos clases padres al mismo tiempo en Java (``extends A, B``).

A) Verdadero B) Falso

PREGUNTA 40. `ArrayList` tiene tamaño fijo como los arreglos: una vez creado no puede crecer.

A) Verdadero B) Falso

Preguntas abiertas y de código

PREGUNTA 41. Escribe un programa completo que pida al usuario un número entero N y calcule la suma de todos los números impares del 1 al N. Debe validar que N sea positivo (repetir la pregunta si no lo es).

PREGUNTA 42. Escribe la clase `Rectangulo` con atributos privados `base` y `altura`, constructor que valide (mayores a 0), getters/setters, y métodos `area()` y `perimetro()`. Luego un `main` que cree dos rectangulos e imprima el de mayor area.

PREGUNTA 43. Dada la clase abstracta `Figura` con el método abstracto `double area()`, escribe las clases `Circulo` (atributo radio) y `Cuadrado` (atributo lado) que la extiendan e implementen `area()`. Luego un `main` con un arreglo polimorfo de 2 figuras que imprima sus areas.

PREGUNTA 44. Escribe un método `public static int contarVocales(String texto)` que devuelva cuantas vocales (a, e, i, o, u, sin importar mayusculas) hay en el texto. Usa un ciclo y `charAt`.

PREGUNTA 45. Con un `HashMap<String, Integer>` llamado `inventario` (producto -> stock), escribe el código que descuente 3 unidades del producto "mouse" solo si hay stock suficiente; si no hay suficiente o no existe, muestra el mensaje apropiado.

PREGUNTA 46. Explica con tus palabras la diferencia entre `Exception` checked y `RuntimeException` (unchecked), y da un ejemplo de cada una.

PREGUNTA 47. Escribe un programa que lea el archivo `datos.txt` línea por línea e imprima cada línea numerada (1: primera línea, 2: segunda...). Debe manejar el caso de que el archivo no exista con un mensaje amigable (sin stack trace).

PREGUNTA 48. Crea dos hilos que impriman números del 1 al 5: uno imprime "A-i" y otro "B-i", cada uno con pausa de 400ms entre números. El `main` debe esperar a que ambos terminen (con `join`) antes de imprimir "Fin".

PREGUNTA 49. Escribe el código JDBC completo (con try-with-resources) que conecte a ``jdbc:mysql://localhost:3306/tienda`` como ``root``, liste todos los productos (``codigo``, ``nombre``, ``precio``) de la tabla ``productos``, e imprima cada uno. Usa las credenciales como variables.

PREGUNTA 50. Disena (en papel o código) las clases para un sistema de biblioteca: identifica las clases, sus atributos y métodos principales, y explica que relación POO usarias entre ellas (herencia, composición o interfaces) y por qué.

Respuestas y soluciones

RESPUESTA 1. B) Escribir, compilar, ejecutar. Primero escribes el código fuente (``.java``), luego lo compilas con ``javac`` (que genera el bytecode ``.class`` y te avisa de errores de sintaxis), y finalmente lo ejecutas con ``java`` sobre la JVM. Este orden no es negociable: no puedes ejecutar lo que no compilaste, ni compilar lo que no escribiste. (Unidad 1)

RESPUESTA 2. B) El JRE más el compilador javac y herramientas de desarrollo. La JVM ejecuta, el JRE es JVM + librerías para correr programas, y el JDK es JRE + compilador + herramientas para programar. Tu necesitas el JDK porque vas a crear programas, no solo correrlos. (Unidad 1)

RESPUESTA 3. B) Factura.java. En Java, si la clase es pública, el archivo debe llamarse exactamente igual, respetando mayúsculas y minúsculas. ``factura.java`` o cualquier otro nombre produce error de compilación. Es una de las primeras lecciones que todo programador Java aprende a golpes. (Unidad 1)

RESPUESTA 4. C) HolaMundo en la primera línea y Java en la segunda. ``print("Hola")`` imprime sin saltar, ``println("Mundo")`` imprime y salta de línea, ``print("Java")`` imprime en la nueva línea sin saltar al final. Resultado: línea 1 = "HolaMundo", línea 2 = "Java". La diferencia entre ``print`` y ``println`` es solo el salto final. (Unidad 1)

RESPUESTA 5. B) 3. División entera: si ambos operandos son ``int``, Java trunca el decimal sin redondear. ``7 / 2`` = 3 (no 3.5, no 4). Para obtener 3.5, al menos un operando debe ser ``double``: ``7.0 / 2``. Esta trampa aparece en exámenes, entrevistas y bugs reales. (Unidad 2)

RESPUESTA 6. B) 2. El módulo ``%`` devuelve el residuo: 17 entre 5 cabe 3 veces (15) y sobran 2. El módulo es tu herramienta para saber si un número es par (``n % 2 == 0``), obtener el último dígito (``n % 10``) y ciclos repetitivos. (Unidad 2)

RESPUESTA 7. C) `if (a.equals(b))`. El `==` con objetos compara referencias (direcciones de memoria), no contenido: dos Strings iguales pueden dar `false` con `==` por como Java optimiza memoria. `.equals()` compara el texto real. La opción B (`=`) asigna en vez de comparar, y la D (`===`) no existe en Java. (Unidades 2 y 3)

RESPUESTA 8. B) Conecta un lector de datos al teclado. `System.in` es la entrada estandar (el teclado) y `Scanner` es la clase que sabe leer de ahí. Necesitas el `import java.util.Scanner;` arriba. Al terminar, ciérralo con `sc.close()`. (Unidad 2)

RESPUESTA 9. B) Lee el Enter pendiente y devuelve un texto vacío. `nextInt()` lee el número pero deja el salto de línea en el buffer; el siguiente `nextLine()` lo consume sin esperar al usuario. Es la trampa clásica del Scanner. Soluciones: un `nextLine()` extra para limpiar, o leer todo con `nextLine()` y convertir con `Integer.parseInt`. (Unidad 2)

RESPUESTA 10. B) false. `(5 > 3)` es `true`, `(2 > 4)` es `false`, y `true && false` es `false` (el AND exige que ambos sean verdaderos). Con `||` habría dado `true`. Lee los operadores lógicos como frases: "5 es mayor que 3 Y 2 es mayor que 4" es falso porque la segunda parte falla. (Unidad 2)

RESPUESTA 11. B) 10. El ciclo va de `i = 0` mientras `i < 10`: valores 0, 1, 2, ..., 9. Son 10 iteraciones. El error típico (off-by-one) es decir 9 u 11. Regla: en `for (i = 0; i < N; i++)` siempre hay exactamente N iteraciones. (Unidad 4)

RESPUESTA 12. B) 1 3. `i` empieza en 1; mientras sea menor que 4, imprime y suma 2: imprime 1 (`i=1`), imprime 3 (`i=3`), luego `i=5` y sale. Observa que el incremento puede ser cualquier valor, no solo 1. (Unidad 4)

RESPUESTA 13. C) La ejecución cae al siguiente `case` (fall-through). Java no sale del `switch` solo; ejecuta el `case` que coincida y sigue con los siguientes hasta encontrar un `break` o terminar. Olvidar el `break` es una de las fuentes de bugs más comunes en principiantes. (Unidad 3)

RESPUESTA 14. B) 2. Los índices van de 0 a `length - 1`. Un arreglo de 3 elementos tiene índices 0, 1, 2. Pedir el índice 3 lanza `ArrayIndexOutOfBoundsException`. (Unidad 5)

RESPUESTA 15. B) 25. `n[1]` es 10 y `n[2]` es 15; `10 + 15 = 25`. Cuidado: el índice 1 es el SEGUNDO elemento, no el primero. (Unidad 5)

RESPUESTA 16. B) Devuelve un valor y termina el método inmediatamente. Todo lo que este después del `return` en ese camino ya no se ejecuta. En métodos `void` puedes usar `return;` solo para salir antes. (Unidad 5)

RESPUESTA 17. B) Parámetros. Los parámetros son las variables declaradas en la firma del método (los "lugares" que reciben valores); los argumentos son los valores concretos que pasas al llamarlo. ``sumar(int a, int b)``: a y b son parámetros; en ``sumar(5, 3)``, 5 y 3 son argumentos. (Unidad 5)

RESPUESTA 18. B) No se modifica (se paso una copia del valor). Java pasa los primitivos por valor: el método recibe una copia independiente. Para que un método "devuelva" un resultado, usa ``return``. (Con objetos se pasa copia de la referencia, pero eso es tema de profundización.) (Unidad 5)

RESPUESTA 19. B) Que solo se puede acceder desde dentro de la misma clase. Es el corazon del encapsulamiento: el atributo queda protegido y el acceso externo pasa por getters/setters, donde puedes validar. ``private`` no significa "gratis" ni "inmodificable": significa acceso controlado. (Unidad 6)

RESPUESTA 20. B) ``public Libro(String titulo)``. El constructor se llama EXACTAMENTE igual que la clase (mayusculas incluidas) y NO lleva tipo de retorno, ni siquiera ``void``. La A falla por el ``void``, la C por la minúscula, la D por el ``String`` de retorno. (Unidad 6)

RESPUESTA 21. B) `extends`. ``class Perro extends Animal`` crea la relación "es un": Perro hereda los atributos y métodos de Animal. ``implements`` es para interfaces. (Unidad 7)

RESPUESTA 22. B) ``super(...)`` como primera línea del constructor. El constructor de la hija debe invocar al del padre antes de hacer cualquier otra cosa, porque el padre debe estar construido antes que la hija. Si el padre no tiene constructor sin parámetros, el ``super(...)`` es obligatorio. (Unidad 7)

RESPUESTA 23. B) Una variable del tipo padre que ejecuta el método de la hija según el objeto real. ``Animal a = new Perro(); a.hacerSonido();`` ejecuta el ``hacerSonido`` de Perro. La decisión se toma en tiempo de ejecución según el objeto real, no según el tipo de la variable. Es la base de los frameworks y del código flexible. (Unidad 7)

RESPUESTA 24. B) Le dice al compilador que estas sobreescribiendo a proposito y te avisa si la firma está mal. Si escribes mal el nombre o los parámetros sin ``@Override``, Java cree que es un método nuevo y tu "sobrescritura" nunca se ejecuta: un bug silencioso y cruel. Con ``@Override``, el compilador te lo senala. No es obligatoria, pero los profesionales la usan siempre. (Unidad 7)

RESPUESTA 25. B) Sobrecarga: mismo nombre, distintos parámetros, misma clase. Sobrescritura: misma firma, la hija redefine el método del padre. **Sobrecarga = varios métodos con el mismo nombre que se distinguen por parámetros (ej. varios constructores). Sobrescritura = la hija cambia el comportamiento heredado (ej. `hacerSonido` en Perro).** (Unidad 7)

RESPUESTA 26. B) No se puede instanciar; puede tener métodos abstractos que las hijas deben implementar. **La clase abstracta es un molde incompleto: define la estructura comun y deja los detalles a las hijas. Si puede tener métodos con código y constructor (para que las hijas lo usen via `super`).** (Unidad 7)

RESPUESTA 27. C) Múltiples. Una clase implementa todas las interfaces que necesite: ``class Robot implements Caminable, Hablable, Recargable``. Es como Java evita la herencia múltiple de clases pero permite la "herencia" múltiple de comportamientos. (Unidad 7)

RESPUESTA 28. A) ``add()``. ``lista.add("Ana")`` agrega al final. ``push()`` es de pilas/Stack, ``insert()`` no existe en ArrayList, ``append()`` es de StringBuilder. (Unidad 8)

RESPUESTA 29. C) ``null``. ``get`` con una llave inexistente devuelve ``null`` sin tronar. Por eso el patrón contador usa ``getOrDefault``, y por eso debes tener cuidado de no llamar métodos sobre ese ``null``. (Unidad 8)

RESPUESTA 30. B) Devuelve el valor actual o 0 si la llave no existe. El patrón ``mapa.put(letra, mapa.getOrDefault(letra, 0) + 1)`` cuenta frecuencias en una sola línea elegante, sin ``if`` de existencia. (Unidad 8)

RESPUESTA 31. A) Verdadero. Esa es exactamente la magia del try-catch: el error se "absorbe" y el programa sigue su curso después del bloque. Sin el catch, la excepción subiría y tronaría el programa. (Unidad 9)

RESPUESTA 32. B) Falso. El ``finally`` se ejecuta SIEMPRE: haya excepción o no, se haya atrapado o no. Su proposito es liberar recursos (cerrar archivos, conexiones) pase lo que pase. (Unidad 9)

RESPUESTA 33. B) Falso. ``FileWriter`` por defecto SOBREESCRIBE el archivo (lo trunca). Para agregar al final necesitas el modo append: ``new FileWriter("datos.txt", true)``. Confundirlos es cómo se pierden datos. (Unidad 9)

RESPUESTA 34. B) Falso. Llamar ``run()`` directamente ejecuta el código en el hilo actual, secuencialmente, sin paralelismo. Para lanzar un hilo nuevo se llama ``start()``, que crea el hilo y luego invoca ``run()`` dentro de el. (Unidad 9)

RESPUESTA 35. A) Verdadero. `synchronized` pone un candado por objeto: solo un hilo a la vez ejecuta ese método sobre el mismo objeto, y los demás esperan su turno. Es la solución a las condiciones de carrera. (Unidad 9)

RESPUESTA 36. A) Verdadero. Cada `<dependency>` en el `pom.xml` lleva sus tres coordenadas: quien la hizo (`groupId`), cómo se llama (`artifactId`) y cual versión. Maven la descarga del repositorio central automáticamente. (Unidad 10)

RESPUESTA 37. B) Falso. Concatenar entradas del usuario en SQL es la receta de la inyección SQL: un usuario malicioso puede inyectar código y leer, borrar o robar tu base de datos. Nunca, jamás, bajo ninguna circunstancia. Usa `PreparedStatement`. (Unidad 10)

RESPUESTA 38. A) Verdadero. Los parámetros `?` separan el SQL de los datos: los datos jamás se interpretan como código. Es la defensa estandar contra inyección SQL y además es más rápido en consultas repetidas. (Unidad 10)

RESPUESTA 39. B) Falso. Java no permite herencia múltiple de clases (el problema del diamante). Si necesitas varios comportamientos, implementa múltiples interfaces. (Unidad 7)

RESPUESTA 40. B) Falso. Esa es justamente la ventaja del `ArrayList` sobre el arreglo: crece dinámicamente con `add()` y se achica con `remove()`. El arreglo `int[]` si tiene tamaño fijo. (Unidad 8)

RESPUESTA 41. Suma de impares con validación. Los puntos clave: el guardián `do-while` para N positivo, el ciclo que suma solo impares (`i % 2 != 0`), y acumulador.

```

import java.util.Scanner;

public class SumaImpares {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n;
        do {
            System.out.print("Escribe un numero positivo: ");
            n = Integer.parseInt(sc.nextLine());
            if (n <= 0) {
                System.out.println("Debe ser mayor que cero.");
            }
        } while (n <= 0);

        int suma = 0;
        for (int i = 1; i <= n; i++) {
            if (i % 2 != 0) {
                suma += i;
            }
        }
        System.out.println("Suma de impares del 1 al " + n + ": " + suma);
        sc.close();
    }
}

```

Si usaste `nextInt()` en vez de `nextLine()` + `parseInt`, también vale, pero recuerda la trampa del Enter pendiente. Si tu validación repite la pregunta hasta dato valido, está perfecta. (Unidades 2, 4)

RESPUESTA 42. Clase Rectangulo encapsulada. **Los puntos clave:** atributos `private`, validación en constructor y setters, y métodos que calculan a partir de los atributos.

```

public class Rectangulo {
    private double base;
    private double altura;

    public Rectangulo(double base, double altura) {
        setBase(base);
        setAltura(altura);
    }

    public double getBase() { return base; }
    public double getAltura() { return altura; }

    public void setBase(double base) {
        if (base > 0) this.base = base;
    }

    public void setAltura(double altura) {
        if (altura > 0) this.altura = altura;
    }

    public double area() { return base * altura; }
    public double perimetro() { return 2 * (base + altura); }

    public static void main(String[] args) {
        Rectangulo r1 = new Rectangulo(5, 3);
        Rectangulo r2 = new Rectangulo(4, 4);
        Rectangulo mayor = (r1.area() > r2.area()) ? r1 : r2;
        System.out.println("Mayor area: " + mayor.area());
    }
}

```

Nota el truco profesional: el constructor llama a los setters para no duplicar la validación. Si tu constructor valida directamente con `if`, también es correcto. (Unidad 6)

RESPUESTA 43. Figuras polimorficas. Los puntos clave: `extends Figura`, `@Override` en `area()`, y el arreglo del tipo padre.

```

abstract class Figura {
    public abstract double area();
}

class Circulo extends Figura {
    private double radio;
    public Circulo(double radio) { this.radio = radio; }
    @Override
    public double area() { return Math.PI * radio * radio; }
}

class Cuadrado extends Figura {
    private double lado;
    public Cuadrado(double lado) { this.lado = lado; }
    @Override
    public double area() { return lado * lado; }
}

public class PruebaFiguras {
    public static void main(String[] args) {
        Figura[] figuras = { new Circulo(2), new Cuadrado(3) };
        for (Figura f : figuras) {
            System.out.println("Area: " + f.area());
        }
    }
}

```

El arreglo es de tipo `Figura` (padre) pero guarda hijas: al llamar `f.area()` se ejecuta la versión de cada hija. Eso es polimorfismo en acción. (Unidad 7)

RESPUESTA 44. Contar vocales. **Los puntos clave:** recorrer con `charAt`, normalizar a minúsculas una vez, y comparar contra las 5 vocales.

```

public static int contarVocales(String texto) {
    int contador = 0;
    String minusculas = texto.toLowerCase();
    for (int i = 0; i < minusculas.length(); i++) {
        char c = minusculas.charAt(i);
        if (c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u') {
            contador++;
        }
    }
    return contador;
}

```

Alternativa valida: usar un String `"aeiou"` y `indexOf(c) != -1`. Si olvidaste el `toLowerCase`, las mayusculas no se contarían: medio punto menos, pero la lógica es la correcta. (Unidades 4, 5)

RESPUESTA 45. Descuento con validación. Los puntos clave: verificar existencia con `containsKey`, verificar stock suficiente, y mensajes distintos para cada caso.

```

if (!inventario.containsKey("mouse")) {
    System.out.println("El producto no existe.");
} else if (inventario.get("mouse") >= 3) {
    inventario.put("mouse", inventario.get("mouse") - 3);
    System.out.println("Descontado. Stock restante: " + inventario.get("mouse"));
} else {
    System.out.println("Stock insuficiente (hay " + inventario.get("mouse") +
    ").");
}

```

El orden importa: primero existencia (para no hacer `get` sobre llave inexistente y operar con `null`), luego suficiencia. (Unidad 8)

RESPUESTA 46. Checked vs unchecked. Las checked (`Exception` y sus hijas, menos `RuntimeException`) son errores que el programador puede anticipar y debe manejar: el compilador te **OBLIGA** a atraparlas o declararlas (ej. `IOException` al leer un archivo, `SQLException`). Las unchecked (`RuntimeException` y sus hijas) son errores de lógica del programador que el compilador no te obliga a manejar (ej. `NullPointerException` por usar un objeto nulo, `ArrayIndexOutOfBoundsException` por índice inválido, `ArithmeticException` por dividir entre cero). La filosofía: las checked son "el mundo exterior puede fallar, prepárate"; las unchecked son "tu código tiene un bug, arreglalo". (Unidad 9)

RESPUESTA 47. Leer archivo numerado, robusto. **Los puntos clave:** verificar existencia con `File.exists()`, `try-with-resources`, y contador de línea.

```
import java.io.File;
import java.util.Scanner;

public class LectorNumerado {
    public static void main(String[] args) {
        File archivo = new File("datos.txt");
        if (!archivo.exists()) {
            System.out.println("El archivo datos.txt no existe. Crealo primero.");
            return;
        }
        try (Scanner lector = new Scanner(archivo)) {
            int numero = 1;
            while (lector.hasNextLine()) {
                System.out.println(numero + ": " + lector.nextLine());
                numero++;
            }
        } catch (Exception e) {
            System.out.println("Error al leer el archivo: " + e.getMessage());
        }
    }
}
```

Sin el `exists()` previo, el programa lanzaría `FileNotFoundException` con stack trace: funcional pero nada amigable. El mensaje claro es lo que distingue un programa profesional. (Unidad 9)

RESPUESTA 48. Dos hilos con join. **Los puntos clave:** `start()` (no `run()`), `Thread.sleep()` dentro de `try-catch`, y `join()` en el main.

```

public class DosHilos {
    public static void main(String[] args) {
        Thread hiloA = new Thread(() -> {
            for (int i = 1; i <= 5; i++) {
                System.out.println("A-" + i);
                try { Thread.sleep(400); } catch (InterruptedException e) {}
            }
        });
        Thread hiloB = new Thread(() -> {
            for (int i = 1; i <= 5; i++) {
                System.out.println("B-" + i);
                try { Thread.sleep(400); } catch (InterruptedException e) {}
            }
        });

        hiloA.start();
        hiloB.start();

        try {
            hiloA.join();
            hiloB.join();
        } catch (InterruptedException e) {
            System.out.println("Interrumpido");
        }
        System.out.println("Fin");
    }
}

```

Si llamaste `run()` en vez de `start()`, los números saldrían ordenados (A1..A5, B1..B5): ejecución secuencial, cero paralelismo. Sin `join()`, el "Fin" saldría antes de que terminen los hilos. (Unidad 9)

RESPUESTA 49. JDBC completo. Los puntos clave: los 5 pasos en orden, try-with-resources, y lectura por nombre de columna.

```

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;

public class ListarProductos {
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/tienda";
        String usuario = "root";
        String password = "tu_password";

        try (Connection con = DriverManager.getConnection(url, usuario, password);
            Statement st = con.createStatement();
            ResultSet rs = st.executeQuery("SELECT codigo, nombre, precio FROM
productos")) {

            while (rs.next()) {
                System.out.println(rs.getString("codigo") + " | "
                    + rs.getString("nombre") + " | $" + rs.getDouble("precio"));
            }
        } catch (Exception e) {
            System.out.println("Error de base de datos: " + e.getMessage());
        }
    }
}

```

Recuerda: necesitas el conector MySQL en el `pom.xml` (o el `.jar` en el classpath) y la base `tienda` creada. El `rs.next()` empieza antes de la primera fila: sin el `while`, no lees nada. (Unidad 10)

RESPUESTA 50. Diseno de biblioteca. **No hay una sola respuesta correcta, pero un buen diseno identifica estos elementos:**

Clases: `Libro` (título, autor, isbn, disponible), `Usuario` (nombre, id, lista de prestamos), `Prestamo` (libro, usuario, fechaPrestamo, fechaDevolucion), `Biblioteca` (catálogo, usuarios; métodos prestar, devolver, buscar).

Relaciones:

- **Composicion:** `Biblioteca` TIENE `Libro`s y `Usuario`s (los administra; `ArrayList<Libro> catalogo`). `Prestamo` TIENE un `Libro` y un `Usuario` (referencias).

- **Herencia:** `UsuarioVIP` extends `Usuario` con más días de prestamo, o `LibroDigital` extends `Libro` con atributo `tamanoMB` y método `descargar()`. El prestamo polimorfo calcula fechas distintas según el tipo.
- **Interfaces:** `Prestable` con métodos `prestar()` y `devolver()`, implementada por `Libro` (y mañana por `Revista`, `DVD` sin cambiar el sistema).

Justificación: composición para las relaciones "tiene un" (la biblioteca tiene libros), herencia para especializaciones "es un" (el libro digital es un libro), interfaces para comportamientos compartidos entre clases no relacionadas (todo lo prestable se presta igual). Si tu diseño separa responsabilidades (el Libro no maneja usuarios, la Biblioteca orquesta) y usa encapsulamiento, está bien encaminado. (Unidades 6, 7, 8)

Cierre del libro

Terminaste el examen y, con el, el **Curso Completo de Java** de Lucas EGA Academy. Si llegaste hasta aquí respondiendo honestamente, ya no eres un principiante: eres un programador Java con base sólida, proyectos en tu haber y criterio para seguir aprendiendo solo.

Guarda este libro. Vuelve a él cuando algo se te olvide (los buenos programadores consultan, no memorizan todo). Y sobre todo: sigue construyendo. El siguiente proyecto te está esperando.

Felicidades. Nos vemos en el siguiente curso. [Lucas EGA Academy \(lucasegaacademy.com\)](https://lucasegaacademy.com)

Temario del examen por pregunta

Para que sepas exactamente que repasar según donde falles:

Preguntas	Unidad	Temas evaluados	--- --- ---	1-4	Unidad 1	Ciclo
escribir-compile-ejecutar, JDK/JRE/JVM, nombre de archivo, print vs println 5-10 Unidad 2						
División entera, módulo, comparación de Strings, Scanner y su trampa del buffer, operadores lógicos 11-13 Unidades 3-4 Conteo de iteraciones del for, while con incremento, fall-through del switch 14-18 Unidad 5 Índices de arreglos, acceso a elementos, return, parámetros vs argumentos, paso por valor 19-20 Unidad 6 Modificador private, firma del constructor 21-27 Unidad 7 extends, super(), polimorfismo, @Override, sobrecarga vs						

sobrescritura, clases abstractas, interfaces | | 28-30 | Unidad 8 | add en ArrayList, get con llave inexistente, getOrDefault | | 31-35 | Unidad 9 | try-catch y flujo, finally, FileWriter y modo append, start vs run, synchronized | | 36-38 | Unidad 10 | Dependencias en pom.xml, inyección SQL, PreparedStatement | | 39-40 | Unidades 7-8 | Herencia múltiple (no existe), ArrayList vs arreglo fijo | | 41 | Unidades 2, 4 | Suma de impares con validación de entrada | | 42 | Unidad 6 | Clase Rectangulo encapsulada con validación | | 43 | Unidad 7 | Jerarquia Figura con polimorfismo | | 44 | Unidades 4, 5 | Conteo de vocales con charAt | | 45 | Unidad 8 | Descuento de inventario en HashMap con validaciones | | 46 | Unidad 9 | Checked vs unchecked con ejemplos | | 47 | Unidad 9 | Lectura de archivo numerada y robusta | | 48 | Unidad 9 | Dos hilos paralelos con join | | 49 | Unidad 10 | JDBC completo con try-with-resources | | 50 | Unidades 6, 7, 8 | Diseno POO de un sistema de biblioteca |

Si fallaste varias de un mismo bloque, relee esa unidad completa antes de reintentar. El examen no es un tramite: es tu mapa de huecos.

Agradecimientos

Este libro no lo escribió una sola persona, aunque lleve mi nombre. Lo escribieron también mis estudiantes: cada pregunta que me hicieron en los comentarios del curso, cada error que reportaron, cada "profe, no entendí" me obligo a explicar mejor. Si una sección de este libro se siente clara, es porque alguien antes se atoro ahí y me lo dijo.

Gracias a la comunidad de Lucas EGA Academy por no dejarme publicar nada a medias. Gracias a quienes probaron los ejercicios y encontraron los bugs que yo no vi. Y gracias a ti, lector, por llegar hasta la última página: los libros se terminan de escribir cuando alguien los lee completos.

Carta final del autor

Si llegaste hasta aquí, hiciste algo que pocos hacen: terminar. La mayoría compra libros (o los descarga) y los deja a la mitad. Tu no. Leiste 10 unidades, hiciste ejercicios, sufriste errores de compilación, los arreglaste, y respondiste un examen de 50 preguntas. Eso habla de ti más que cualquier certificado.

Te voy a decir algo que nadie te dice en los cursos: el momento más peligroso es justo este. Terminar un libro se siente como llegar a la meta, y la tentación es descansar "un rato" que se convierte en meses. No lo hagas. El conocimiento de programación es como un músculo: lo que no se usa, se atrofia. Mañana mismo abre tu editor y escribe algo. Lo que sea: el reto 30 del Apéndice C, una mejora a la tienda, un programa inútil que te de risa. Pero escribe.

Y cuando te estanques (porque te vas a estancar, todos nos estancamos), recuerda esto: cada programador que admiras estuvo exactamente donde estas tu, mirando un error que no entendía, a las 2 de la mañana, preguntándose si esto era para él. La diferencia entre ellos y los que abandonaron no fue el talento: fue quedarse una hora más.

Java te dio una base que te sirve para todo: backend con Spring, apps con Android, sistemas empresariales, y la forma de pensar que te permite aprender cualquier otro lenguaje en semanas. Lo difícil ya lo hiciste. Lo que sigue es construir.

Gracias por confiar en Lucas EGA Academy para tu formación. Este libro es tuyo: subrayalo, rompeló, regalalo, vuelve a él cuando lo necesites. Y cuando consigas tu primer trabajo o tu primer cliente gracias a lo que aprendiste aquí, me debes un mensaje. Me encantaría leerlo.

A programar. Siempre.

Esteban lucasegaacademy.com

Si encuentras un error en el libro (de código, de ortografía, de concepto), reportalo en lucasegaacademy.com para corregirlo en la siguiente edición. Los libros vivos son mejores que los libros perfectos.

Tu plan de 30 días después del libro

No dejes que el impulso se enfrie. Aquí tienes un plan concreto, día por día en bloques semanales:

Semana 1: consolida. Rehaz sin mirar los 3 proyectos de la Unidad 10 (calculadora, agenda, tienda). Si te atorras, consulta el libro: eso está permitido, es repaso activo. Sube los 3 a GitHub con buenos README.

Semana 2: profundiza. Elige tu punto débil del examen (el bloque donde más fallaste) y construye un mini-proyecto que lo ejercite: si fue JDBC, una app que guarde y consulte datos reales; si fue hilos, un descargador paralelo; si fue POO, un sistema con 5+ clases.

Semana 3: herramientas. Aprende Git a fondo (ramas, merges, pull requests) y publica todo tu trabajo. Instala Maven y convierte tus proyectos en proyectos Maven. Lee código ajeno en GitHub 30 minutos al día.

Semana 4: siguiente nivel. Empieza Spring Boot (el curso de la academia) o Android, según tu meta. Con tu base, el primer "Hola Mundo" web o móvil te tomara días, no meses.

Regla de oro del plan: programa todos los días, aunque sean 30 minutos. Un mes de constancia te pone por encima del 90% de quienes "quieren aprender". Marca cada día cumplido en un calendario visible: la racha se vuelve su propia motivación.

Colibrí Editorial EGA

Curso Completo de Java

Curso completo de Java: fundamentos, datos, decisiones, ciclos, POO, colecciones, archivos y proyectos reales, con apéndices, retos y examen final.

lucasegaacademy.com

© 2026 Esteban Gutiérrez · Colibrí Editorial EGA · Hecho en México