La evolución de Java a través de sus LTS: de Java 8 a Java 25

3 de noviembre de 2026 java jdk lts java-8 java-11 java-17 java-21 java-25

Imagina que te toca mantener un servicio escrito en Java 8 y alguien del equipo propone pasarlo a Java 21. En la reunión salen palabras como records, virtual threads o pattern matching, y la pregunta que nadie dice en voz alta es: ¿qué cambia de verdad en el código que escribo todos los días?

Esta guía responde esa pregunta, versión por versión. No es una lista de todo lo que cambió en el JDK: es un recorrido por las novedades que más vas a encontrar en código real, explicadas como me hubiera gustado que me las explicaran a mí.

Para cada novedad vas a ver lo mismo: **por qué nació**, cómo se resolvía el problema sin ella, cómo queda con ella y cuándo conviene no usarla. Si recién llegas a Java, el texto principal no da nada por sabido. Si ya llevas años, los recuadros **Para profundizar** tienen el detalle fino.

Todo el código de los ejemplos lo compilé y lo ejecuté en el JDK que indica su etiqueta, y las salidas que ves son las que obtuve.

▼ EN ESTE ARTÍCULO

01. Antes de empezar: por qué seguimos las LTS
02. Java 8: el salto funcional
03. Java 11: menos ceremonia, más JDK
04. Java 17: datos y decisiones sin ceremonia
05. Java 21: concurrencia barata y datos que se desarman
06. Java 25: menos ruido, más intención
07. Apéndice: todas las novedades en una tabla
08. Para cerrar
09. Referencias

Antes de empezar: por qué seguimos las LTS

Desde 2018, Java publica una versión nueva cada seis meses. Algunas son **LTS** (*long-term support*, soporte a largo plazo): versiones que el proveedor se compromete a mantener con parches durante años.

Muchos equipos se quedan en una LTS y saltan directamente a la siguiente, así que es una forma natural de ordenar la historia.

En esta guía uso la clasificación de Oracle, donde las LTS son 8, 11, 17, 21 y 25 (Oracle Java SE Support Roadmap). Cada proveedor de JDK publica sus propios plazos de soporte, así que si tu equipo usa otro, conviene revisar los suyos.

Lo importante es esto: **muchas novedades no nacen en una LTS**. Los records, por ejemplo, aparecieron como *preview* (una versión de prueba que hay que activar a mano) en Java 14 y quedaron definitivos en Java 16. Java 17 es la primera LTS que los trae. Por eso cada novedad lleva tres datos:

Dato	Qué responde
Definitiva en	Desde qué versión se usa sin activar nada especial
Primera LTS	En qué capítulo de esta guía la vas a encontrar
Origen	Si antes pasó por preview o incubadora, y en qué versiones

Las cinco LTS de Java y lo que trae cada una

Java 8 2014

- Lambdas — Lenguaje
- Streams — API
- Optional — API
- java.time — API
- Métodos default — Lenguaje

Java 11 2018

- var — Lenguaje · definitiva en Java 10
- List.of, Set.of, Map.of — API · definitiva en Java 9
- HttpClient — API
- Métodos nuevos de String — API
- java Hola.java — Herramientas

Java 17 2021

- Switch expressions — Lenguaje · definitiva en Java 14
- Text blocks — Lenguaje · definitiva en Java 15
- Records — Lenguaje · definitiva en Java 16
- Pattern matching en instanceof — Lenguaje · definitiva en Java 16
- Sealed classes — Lenguaje

Java 21 2023

- Virtual threads — JVM
- Pattern matching en switch — Lenguaje
- Record patterns — Lenguaje
- Sequenced collections — API

Java 25 2025

- Variables sin nombre (_) — Lenguaje · definitiva en Java 22
- Stream Gatherers — API · definitiva en Java 24
- Scoped values — API

Compact source files
Lenguaje

Constructores flexibles
Lenguaje

Dos aclaraciones antes de arrancar. Una función en preview sigue siendo preview aunque venga dentro de una LTS: no la cuento como parte de esa versión hasta que es definitiva. Y el recorrido termina en Java 25: al escribir esto ya salieron Java 26 y 27, ninguna de las dos es LTS, y la próxima LTS que Oracle tiene planificada es Java 29, en septiembre de 2027 (Oracle Java SE Support Roadmap).

Java 8: el salto funcional

Java 8 salió en marzo de 2014 y es, probablemente, el cambio más grande en la forma de escribir Java desde su creación. Si solo pudieras aprender un capítulo de esta guía, sería este: casi todo lo que vino después se apoya en él.

Lambdas: pasar comportamiento sin escribir una clase

Lambdas e interfaces funcionales · lenguaje · definitiva en Java 8 · primera LTS: Java 8 · JEP 126

Imagina que tienes una lista de pedidos y quieres ordenarla por importe y, cuando dos importes empatan, por el nombre del cliente.

Por qué nace. Para ordenar, **`Collections.sort`** necesita que le digas cómo se comparan dos pedidos. Antes de Java 8, ese "cómo" solo se podía pasar como un objeto, casi siempre una **clase anónima**: una clase sin nombre que declaras y creas en el mismo lugar. Lo que te importa son las líneas que comparan; el resto (**`new Comparator<Pedido>()`**, **`@Override`**, la firma de **`compare`**) es ceremonia que se repite cada vez. El tutorial oficial de Java lo reconoce: con interfaces de un solo método, esa sintaxis resulta "poco manejable y poco clara" (The Java Tutorials, Lambda Expressions).

El JEP 126 agrega la otra mitad: otros lenguajes ya tenían *closures* (funciones que se pasan como un valor), y sin ellas Java no podía ofrecer la *iteración interna* que verás en Streams.

El mismo orden, antes y después:

ANTES · JAVA 7

Antes.java

```
static void ordenar(List<Pedido> pedidos)
{
    Collections.sort(pedidos, new
Comparator<Pedido>() {
        @Override
```

DESPUÉS · JAVA 8

Despues.java

```
static void ordenar(List<Pedido> pedidos)
{

pedidos.sort(Comparator.comparing(Pedido:
:getImporte)
```

```
        public int compare(Pedido a,
Pedido b) {
            int porImporte =
Integer.compare(a.getImporte(),
b.getImporte());
            if (porImporte != 0) {
                return porImporte;
            }
            return
a.getCliente().compareTo(b.getCliente());
        }
    });
}
```

```
.thenComparing(Pedido::getCliente));
}
```

SALIDA

```
Pedidos ordenados: [Luis 40, Bruno 90, Ana 120, Marta 120]
```

El "después" usa **`List.sort`**, un método que Java 8 le agregó a **`List`**; cómo pudo hacerlo sin romper nada lo verás al final del capítulo, en los métodos default.

Qué es. Una **lambda** es una función corta escrita en el lugar donde se usa. **`(a, b) -> Integer.compare(a.getImporte(), b.getImporte())`** se lee "dados dos pedidos, compáralos por importe". Java la convierte en una instancia de una **interfaz funcional**: una interfaz con un solo método abstracto, como **`Comparator`**. La anotación **`@FunctionalInterface`** le pide al compilador que compruebe esa regla, aunque cualquier interfaz que la cumpla sirve, anotada o no (javadoc de **`FunctionalInterface`**). El paquete **`java.util.function`** trae las de uso general: **`Predicate`** (una condición), **`Function`** (una transformación), **`Consumer`** (usa un valor) y **`Supplier`** (produce uno).

Una **referencia a método**, como **`Pedido::getImporte`**, es una lambda más corta para cuando solo llamas a un método que ya existe: equivale a **`pedido -> pedido.getImporte()`**. **`Comparator.comparing`** recibe esa función y construye el comparador, y **`thenComparing`** agrega el desempate.

Por qué es mejor. El "después" dice el criterio en el mismo orden en que lo pensaste: por importe, luego por cliente. No hay un **`if (porImporte != 0)`** que escribir bien ni una clase que leer para encontrar la regla.

Cuándo no usarla. Cuando la lógica pasa de dos o tres líneas, se lee mejor como un método con nombre que pasas con una referencia. Y una lambda solo puede usar las variables locales de afuera que no cambian después de asignarse.

> – Para profundizar
>
> la regla exacta es que toda variable local, parámetro o parámetro de excepción que una lambda use sin declararlo tiene que ser **`final`** o *effectively final*: que nunca se reasigne después de recibir su

valor (JLS 8, §15.27.2). A diferencia de una clase anónima, dentro de una lambda **`this`** es el objeto que la rodea (JLS 8, §15.8.3). Y **`comparing(Pedido::getImporte)`** convierte cada **`int`** en **`Integer`** para compararlo; **`comparingInt`** se ahorra esa conversión.

Ejercicio. Ordena los pedidos de mayor a menor importe, con los empates todavía por cliente de la A a la Z. Pista: **`Comparator`** tiene un método que invierte el orden, y dónde lo pones importa.

– Solución

`reversed()` invierte el comparador construido *hasta ese punto*, así que va justo después del importe:

Ejercicio.java

```
pedidos.sort(Comparator.comparing(Pedido::getImporte)
        .reversed()
        .thenComparing(Pedido::getCliente));
```

Imprime **`De mayor a menor: [Ana 120, Marta 120, Bruno 90, Luis 40]`**. Con **`reversed()`** al final de la cadena se invierte también el desempate, y Marta queda antes que Ana: **`Con reversed al final: [Marta 120, Ana 120, Bruno 90, Luis 40]`**.

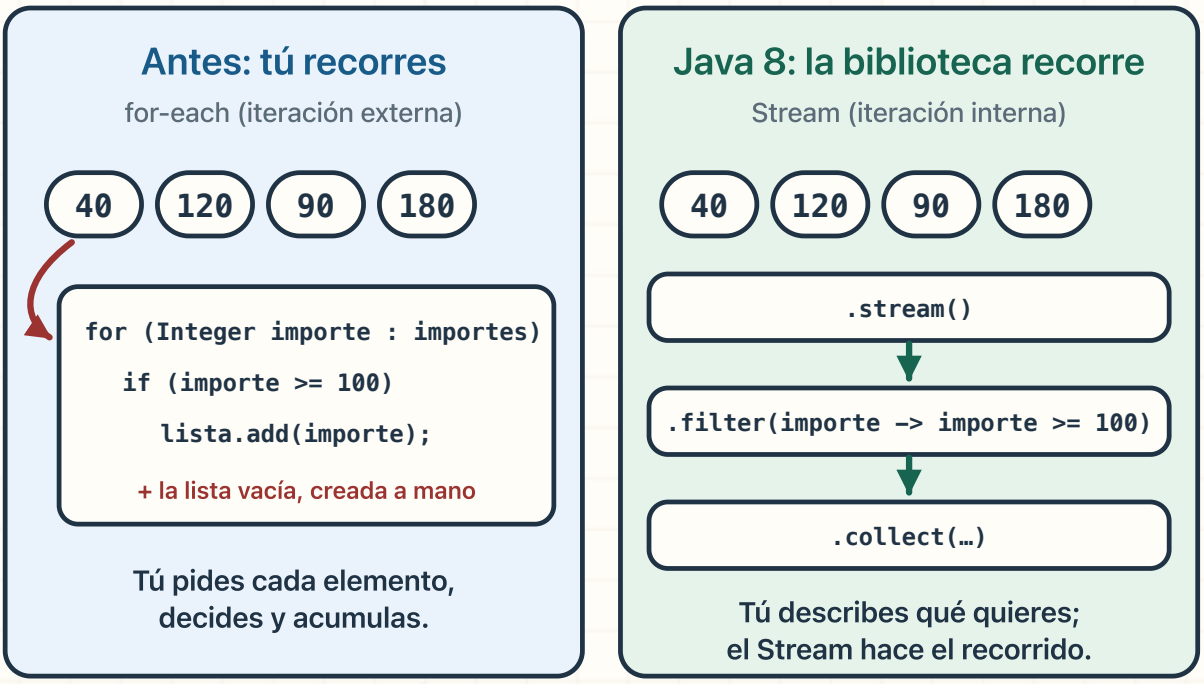
Streams: describir qué quieres, no cómo recorrer

Streams · API · definitiva en Java 8 · primera LTS: Java 8 · JEP 107

Imagina que tienes los importes de varios pedidos y quieres quedarte con los que llegan a 100.

Por qué nace. Antes de Java 8, la única forma de recorrer una colección era que **tu código** la recorriera: un bucle **`for`** que pide cada elemento, decide qué hacer con él y lo guarda en una lista que creaste a mano. A eso se le llama *iteración externa*. Funciona, pero tiene dos costos. El primero es que lo que quieres decir ("los importes de 100 o más") queda escondido entre la lista vacía, el bucle, el **`if`** y el **`add`**. El segundo es menos visible: como el recorrido lo controla tu código, la biblioteca no puede hacer nada para optimizarlo; por ejemplo, repartirlo entre varios núcleos.

Los JEP que dieron origen a esta parte de Java lo dicen así: el JEP 126 propone pasar a la *iteración interna*, donde la colección decide cómo recorrerse, y el JEP 107 busca operaciones de filtrar, transformar y reducir como las que otros lenguajes ya tenían, con la puerta abierta a ejecutarlas en paralelo sin reescribir el programa.



Con un bucle, tu código maneja el recorrido. Con un Stream, describes qué quieres y el recorrido lo hace la biblioteca.

El mismo problema, antes y después. A la izquierda, lo mejor que permitía Java 7; a la derecha, Java 8:

ANTES · JAVA 7

Antes.java

```
static List<Integer>
importesDesde100(List<Integer> importes)
{
    List<Integer> seleccionados = new
ArrayList<>();
    for (Integer importe : importes) {
        if (importe >= 100) {
            seleccionados.add(importe);
        }
    }
    return seleccionados;
}
```

DESPUÉS · JAVA 8

Despues.java

```
static List<Integer>
importesDesde100(List<Integer> importes)
{
    return importes.stream()
            .filter(importe -> importe >=
100)

.collect(Collectors.toCollection(ArrayLis
t::new));
}
```

SALIDA

```
Importes de 100 o más: [120, 180]
```

Las dos versiones devuelven lo mismo, en el mismo orden, y en las dos la lista resultante se puede seguir modificando. Eso último no es casualidad: lo pedí a propósito, y lo explico en el recuadro de abajo.

Qué es. Un **Stream** es una secuencia de elementos que pasa por una cadena de operaciones. No guarda nada: es la descripción de qué hacer con cada elemento. La cadena tiene tres partes:

1. **La fuente**: de dónde salen los elementos. Aquí, **`importes.stream()`**.
2. **Operaciones intermedias**: transforman la secuencia. **`filter`** deja pasar solo los elementos que cumplen una condición.
3. **La operación terminal**: produce el resultado. **`collect`** reúne los elementos en una colección.

La condición **`importe -> importe >= 100`** es una lambda como las de la sección anterior: se lee "dado un importe, devuelve si es mayor o igual a 100".

Lo que más sorprende al principio es el orden en que pasan las cosas. Un Stream no filtra la lista entera y después la copia: **cada elemento recorre la cadena completa antes de que entre el siguiente**. Recórrelo paso a paso:

Cómo recorre un Stream los importes



La lista tiene cuatro importes. Escribir la cadena no ejecuta nada todavía: el trabajo empieza cuando llamas a collect.

Entra el 40. filter comprueba la condición: no llega a 100, así que se descarta y no sigue.

Entra el 120. Pasa el filtro y llega directamente a la lista resultado, antes de que el siguiente importe empiece.

Entra el 90. Tampoco llega a 100: se descarta.

Entra el 180. Pasa y se agrega detrás del 120: el orden de la lista original se conserva. Resultado: [120, 180].

Por qué es mejor. El código con Stream se lee de arriba abajo como una frase: "de los importes, quédate con los que llegan a 100 y reúnelos en una lista". No hay lista vacía que crear ni **`add`** que olvidar, y cada paso tiene un nombre que dice lo que hace. Cuando la cadena crece (filtrar, transformar, ordenar, agrupar), la diferencia con un bucle anidado se nota todavía más.

Cuándo no usarlo. El bucle sigue siendo válido, y a veces es mejor:

- Cuando necesitas cortar el recorrido con lógica compleja (varios **`break`** o **`return`** según el estado), un bucle lo expresa con más claridad.
- Cuando el cuerpo lanza excepciones chequeadas (como **`IOException`**): las interfaces que usa un Stream, como la de **`filter`**, no las declaran, así que hay que capturarlas o envolverlas dentro de

la lambda.

- Cuando esperas que sea más rápido. Un Stream no es más rápido por ser un Stream, ni corre en paralelo a menos que lo pidas con **`parallel()`**, que tiene sus propias trampas.

> **– Para profundizar**
>
> el "antes" y el "después" son equivalentes porque escogí el colector con cuidado. **`Collectors.toList()`** no garantiza nada sobre el tipo ni la mutabilidad de la lista que devuelve (lo dice su javadoc), así que para comparar en igualdad de condiciones con el bucle pedí un **`ArrayList`** explícito con **`Collectors.toCollection(ArrayList::new)`**. Desde Java 16 existe **`Stream.toList()`**, más corto, pero devuelve una lista **inmodificable**: si después alguien hace **`add`**, falla en ejecución. La evaluación perezosa (nada corre hasta la operación terminal) es también lo que permite operaciones como **`findFirst`** o **`limit`**, que cortan el recorrido sin procesar el resto. Si quieres ir más a fondo, Streams en Java: de filter a Collectors expertos recorre los Collectors avanzados y las trampas de **`parallel()`**.

Ejercicio. Con la misma lista de importes, cuenta cuántos pedidos llegan a 100 sin crear ninguna lista intermedia. Pista: hay una operación terminal que cuenta.

> **– Solución**
>
> **`count()`** es una operación terminal que devuelve cuántos elementos llegaron hasta ella, como un **`long`**:
>
> Ejercicio.java
>
> ```
> long pedidosGrandes = importes.stream()
>         .filter(importe -> importe >= 100)
>         .count();
> ```
>
> Imprime **`Pedidos de 100 o más: 2`**. Ningún elemento se guarda en una colección: el Stream solo cuenta los que pasan el filtro.

> **Lo que debes llevarte:** un Stream describe *qué* quieres hacer con una colección y deja el recorrido a la biblioteca. Cada elemento pasa por toda la cadena antes que el siguiente, el orden se conserva y nada corre hasta la operación terminal. No es más rápido por sí solo, y el bucle sigue siendo la mejor opción cuando el control del recorrido es complejo.

Optional: decir en el tipo que el resultado puede faltar

Optional · API · definitiva en Java 8 · primera LTS: Java 8 · sin JEP propio

Ahora buscas un pedido por su identificador para saber de qué cliente es. Puede que ese pedido no exista.

Por qué nace. **`Optional`** no tiene un JEP que cuente su motivo: su javadoc de Java 8 solo lo describe como un contenedor que puede tener o no un valor no nulo. Así que lo que sigue es mi lectura. Antes de Java 8, la forma habitual de decir "no lo encontré" era devolver **`null`**. El problema es que **`null`** no aparece en la firma: **`Pedido buscarPedido(...)`** se ve igual devuelva siempre un pedido o a veces nada. Quien llama tiene que saber que debe comprobarlo. Si lo olvida, el programa compila igual y falla en ejecución con una **`NullPointerException`** (la NPE, el error de usar una referencia que no apunta a nada) al llamar a **`getCliente()`**, muchas veces lejos de donde nació el **`null`**.

Esa lectura tiene respaldo oficial, aunque posterior: desde Java 9, el javadoc de **`Optional`** dice que está pensado principalmente como tipo de retorno de un método, cuando hace falta representar "sin resultado" y usar **`null`** probablemente causaría errores.

ANTES · JAVA 7

Antes.java

```
static Pedido buscarPedido(List<Pedido>
pedidos, String id) {
    for (Pedido pedido : pedidos) {
        if (pedido.getId().equals(id)) {
            return pedido;
        }
    }
    return null;
}

static String clienteDe(List<Pedido>
pedidos, String id) {
    Pedido pedido = buscarPedido(pedidos,
id);
    if (pedido == null) {
        return "sin cliente";
    }
    return pedido.getCliente();
}
```

DESPUÉS · JAVA 8

Despues.java

```
static Optional<Pedido>
buscarPedido(List<Pedido> pedidos, String
id) {
    return pedidos.stream()
            .filter(pedido ->
pedido.getId().equals(id))
            .findFirst();
}

static String clienteDe(List<Pedido>
pedidos, String id) {
    return buscarPedido(pedidos, id)
            .map(Pedido::getCliente)
            .orElse("sin cliente");
}
```

SALIDA

```
Cliente de P-2: Luis
Cliente de P-9: sin cliente
```

Fíjate en que el "antes" es correcto: comprueba el **`null`**. La diferencia es que en el "después" no puedes olvidarte, porque el tipo de retorno te obliga a decidir qué pasa cuando no hay pedido.

Qué es. Un `Optional<Pedido>` es una caja que contiene un pedido o está vacía. Se crea con `Optional.of(valor)`, `Optional.empty()` u `Optional.ofNullable(valor)` si el valor puede ser `null`, y algunas APIs de Java 8 ya lo devuelven, como el `findFirst()` de los Streams. Para usar el contenido sin sacarlo a mano:

- `map(Pedido::getCliente)` transforma el valor si existe; si la caja está vacía, devuelve otra caja vacía.
- `orElse("sin cliente")` entrega el valor o, si no hay, el que le pasas.

Por qué es mejor. La posibilidad de que falte está en la firma, y el caso "no hay pedido" se resuelve en la misma cadena que el caso normal, sin un `if` aparte.

Cuándo no usarlo. `Optional` es para valores de retorno. Fuera de ahí, estorba:

- **Como campo de una clase.** No implementa `Serializable` (su declaración no lo incluye) y cada valor presente va envuelto en un objeto más. Dentro de tu clase, un campo que puede ser `null` es un detalle privado que controlas tú.
- **Como parámetro.** Obliga a quien llama a envolver sus argumentos, y nada le impide pasarte `null` en vez de un `Optional`. Dos versiones del método, con y sin ese parámetro, son más claras.
- **Dentro de una colección**, como `List<Optional<Pedido>>`. Una lista ya sabe expresar "no hay": basta con que no incluya el elemento.

Y no llames a `get()` sin comprobar antes: con la caja vacía lanza `NoSuchElementException`, y habrás cambiado una excepción por otra.

> – Para profundizar
>
> `orElse(x)` recibe `x` ya calculado, haya valor o no, porque Java evalúa los argumentos antes de llamar al método. Si el valor por defecto es caro de obtener (una consulta, por ejemplo), `orElseGet(() -> ...)` solo invoca su función cuando la caja está vacía. Y `Optional.of(null)` lanza una NPE: si el valor puede ser `null`, usa `ofNullable`.

java.time: fechas que no cambian a tus espaldas

java.time · API · definitiva en Java 8 · primera LTS: Java 8 · JEP 150

Un pedido se hace el 27 de febrero de 2024, se entrega tres días después y el cliente tiene un mes desde la entrega para devolverlo.

Por qué nace. El JEP 150 lo resume en una línea: las clases de fecha y hora que había eran "pobres, mutables y de rendimiento impredecible", y existía un deseo de larga data por una API mejor basada en el proyecto Joda-Time. En el día a día se notaba así:

- **Son mutables**, es decir, su valor puede cambiar después de creadas. **`Calendar.add`** modifica el mismo objeto, y un **`Date`** se puede cambiar con **`setTime`**. Si guardas un **`Date`** en un pedido y otro código modifica ese mismo objeto, tu pedido cambia de fecha sin que nadie haya tocado el pedido.
- **Los meses empiezan en 0.** Lo dicen los javadoc de **`Date`** y de **`Calendar`**: enero es 0 y diciembre es 11. Si escribes **`new GregorianCalendar(2024, 2, 27)`** pensando en febrero, obtienes el 27 de marzo.
- **`SimpleDateFormat` no está sincronizado.** Su javadoc recomienda una instancia por hilo, o sincronizar el acceso desde fuera. Una constante compartida sin más, lo primero que se te ocurre, no cumple ninguna de las dos.

ANTES · JAVA 7

Antes.java

```
Calendar calendario = new
GregorianCalendar(2024,
Calendar.FEBRUARY, 27);
Date fechaPedido = calendario.getTime();
calendario.add(Calendar.DAY_OF_MONTH, 3);
Date fechaEntrega = calendario.getTime();
calendario.add(Calendar.MONTH, 1);
Date devolucionHasta =
calendario.getTime();

SimpleDateFormat formato = new
SimpleDateFormat("dd/MM/yyyy");
System.out.println("Pedido: " +
formato.format(fechaPedido));
System.out.println("Entrega: " +
formato.format(fechaEntrega));
System.out.println("Devolución hasta: " +
formato.format(devolucionHasta));
```

DESPUÉS · JAVA 8

Despues.java

```
LocalDate fechaPedido =
LocalDate.of(2024, 2, 27);
LocalDate fechaEntrega =
fechaPedido.plusDays(3);
LocalDate devolucionHasta =
fechaEntrega.plus(Period.ofMonths(1));

DateTimeFormatter formato =
DateTimeFormatter.ofPattern("dd/MM/yyyy")
;
System.out.println("Pedido: " +
fechaPedido.format(formato));
System.out.println("Entrega: " +
fechaEntrega.format(formato));
System.out.println("Devolución hasta: " +
devolucionHasta.format(formato));
```

SALIDA

```
Pedido: 27/02/2024
Entrega: 01/03/2024
Devolución hasta: 01/04/2024
```

El "antes" es correcto, pero mira el orden: **`calendario`** es un solo objeto que cambia con cada **`add`**, así que cada fecha hay que pedirla (**`getTime()`**) en el momento justo. Usé **`Calendar.FEBRUARY`** en lugar de un número para no caer en el mes 0; el programa completo lo confirma al imprimir **`Calendar.FEBRUARY vale 1`**, mientras que en **`java.time`** febrero es el mes 2.

Qué es. `java.time` es el paquete nuevo de fechas y horas, con una clase para cada concepto:

- **`LocalDate`**: una fecha sin hora ni zona horaria, como el 27 de febrero de 2024.

- **`LocalDateTime`** y **`ZonedDateTime`**: fecha con hora, sin zona o con ella. **`Instant`**: un punto exacto en la línea del tiempo.
- **`Period`**: una cantidad de tiempo en años, meses y días, como "un mes". **`Duration`** es su equivalente en horas, minutos y segundos.
- **`DateTimeFormatter`**: el formateador.

Todas sus clases son **inmutables y seguras entre hilos** (lo dice la documentación del paquete): **`plusDays(3)`** no cambia **`fechaPedido`**, devuelve una fecha nueva.

Por qué es mejor. Cada fecha tiene su variable y ninguna cambia después, así que el orden de las líneas deja de importar. Los meses van de 1 a 12, el formateador se puede guardar en una constante y el tipo dice qué representa el valor: un **`LocalDate`** no puede esconder una hora.

Cuándo no usarlo. En código nuevo casi no hay razón para volver a **`Date`**. El límite está en las bibliotecas que todavía piden un **`Date`**, y ni ahí hace falta abandonar **`java.time`**: Java 8 agregó **`Date.from(instant)`** y **`date.toInstant()`** para convertir solo en ese punto. Y escoge bien la clase: **`LocalDate`** no sabe de zonas horarias, así que no sirve para registrar el momento exacto de un pago; para eso está **`Instant`**.

Ejercicio. ¿Cuánto margen hay entre el día del pedido y el último día para devolverlo? Pista: **`Period`** sabe calcularlo entre dos fechas.

<details>
<summary>Solución</summary>

`Period.between` cuenta los meses completos y después los días que sobran:

Ejercicio.java

```
LocalDate fechaPedido = LocalDate.of(2024, 2, 27);
LocalDate devolucionHasta = LocalDate.of(2024, 4, 1);
Period margen = Period.between(fechaPedido, devolucionHasta);
```

Imprime **`Margen: 1 mes y 5 días (P1M5D)`**: del 27 de febrero al 27 de marzo hay un mes, y de ahí al 1 de abril, cinco días. **`P1M5D`** es como **`Period`** se escribe a sí mismo, en el formato ISO 8601.
</details>

Métodos default: agregar métodos a una interfaz sin romper a nadie

Métodos default · lenguaje · definitiva en Java 8 · primera LTS: Java 8 · JEP 126

Tu equipo publicó hace tiempo una interfaz **`Pedidos`** con un solo método, **`todos()`**, y otros equipos ya la implementan. Ahora quieres que cualquier implementación sepa calcular el importe total.

Por qué nace. Antes de Java 8, una interfaz solo podía declarar métodos abstractos, sin cuerpo. Agregarle uno rompía a todas las clases que ya la implementaban: dejaban de compilar hasta que alguien escribiera

el método nuevo. Con una interfaz tuya y dos implementaciones, es una tarde de trabajo. Con **`java.util.Collection`**, implementada por código que no es tuyo, es imposible.

Y ese era el problema de Java 8: los Streams necesitaban que cada colección tuviera un método **`stream()`**. El JEP 126 lo plantea así: los *virtual extension methods* (el nombre de trabajo de los métodos default) resuelven la limitación histórica de no poder añadir métodos a interfaces muy usadas, y dejan evolucionar las interfaces sin romper la compatibilidad de código fuente ni binaria. Por eso **`importes.stream()`** funciona con cualquier lista: **`stream()`** es un método default de **`Collection`** desde Java 8, igual que **`forEach`** lo es de **`Iterable`** (javadoc de ambas).

Sin métodos default, la salida era una clase de utilidades aparte, igual que **`Collections`** acompaña a **`Collection`**:

ANTES · JAVA 7

Antes.java

```
interface Pedidos {
    List<Pedido> todos();
}

// Ya existía y no se toca.
static class PedidosEnMemoria implements
Pedidos {
    private final List<Pedido> pedidos;

    PedidosEnMemoria(List<Pedido>
pedidos) {
        this.pedidos = pedidos;
    }

    @Override
    public List<Pedido> todos() {
        return pedidos;
    }
}

// La operación nueva no cabe en la
interfaz sin romper a quien la
implementa.
static final class PedidosUtil {
    private PedidosUtil() {
    }

    static int importeTotal(Pedidos
pedidos) {
        int total = 0;
        for (Pedido pedido :
pedidos.todos()) {
```

DESPUÉS · JAVA 8

Despues.java

```
interface Pedidos {
    List<Pedido> todos();

    default int importeTotal() {
        int total = 0;
        for (Pedido pedido : todos()) {
            total += pedido.getImporte();
        }
        return total;
    }
}

// Ya existía y no se toca.
static class PedidosEnMemoria implements
Pedidos {
    private final List<Pedido> pedidos;

    PedidosEnMemoria(List<Pedido>
pedidos) {
        this.pedidos = pedidos;
    }

    @Override
    public List<Pedido> todos() {
        return pedidos;
    }
}
```

```
            total += pedido.getImporte();
        }
        return total;
    }
}
```

SALIDA

```
Importe total: 340
```

En las dos versiones, **`PedidosEnMemoria`** es idéntica: es la clase que ya existía. Lo único que cambia es dónde vive **`importeTotal`**.

Qué es. Un **método default** es un método de interfaz con cuerpo, marcado con la palabra **`default`**. Las clases que implementan la interfaz lo heredan sin hacer nada, y cualquiera puede sobrescribirlo si tiene una forma mejor: una implementación sobre una base de datos podría calcular el total con una consulta. Su cuerpo solo puede apoyarse en otros métodos de la interfaz (aquí, **`todos()`**), porque una interfaz no tiene campos de instancia.

Por qué es mejor. La operación vive donde la buscas: **`tienda.importeTotal()`** aparece al autocompletar, en vez de esconderse en una clase **`PedidosUtil`** que hay que conocer. Y **`PedidosEnMemoria`** no cambió ni una línea.

Cuándo no usarlo. Un método default sirve para hacer evolucionar una interfaz, no para convertirla en una clase con herencia múltiple. Si necesitas guardar estado, lo que buscas es una clase abstracta: una interfaz no puede tenerlo. Y ten cuidado al agregarlo a una interfaz pública: si una implementación ya tenía un método con la misma firma, el de la clase gana (JLS 8, §8.4.8) y el tuyo nunca se ejecuta para ella.

> – Para profundizar
>
> si una clase implementa dos interfaces que traen el mismo método default, el compilador la rechaza (JLS 8, §8.4.8.4); javac 8 lo informa como **`class C inherits unrelated defaults for nombre() from types A and B`**. Se resuelve sobrescribiendo el método en la clase, y desde ahí puedes llamar a la versión que prefieras con **`A.super.nombre()`**. Si en cambio la superclase ya tiene un método concreto con esa firma, ese gana y no hay conflicto. Las tres cosas las comprobé compilando con javac 8.

También en Java 8. **`CompletableFuture`**, un **`Future`** (el resultado de una tarea que todavía puede estar en curso) que se puede completar de forma explícita y que, como **`CompletionStage`**, permite encadenar funciones y acciones que se disparan cuando se completa (javadoc de **`CompletableFuture`**, Java 8).

> **Lo que debes llevarte:** las piezas de Java 8 se apoyan entre sí. Las lambdas y las referencias a métodos te dejan pasar comportamiento como un valor, siempre a través de una interfaz funcional. Los Streams usan eso para que describas *qué* hacer con una colección y dejes el recorrido a la biblioteca: cada elemento pasa por toda la cadena, el orden se conserva y nada corre hasta la operación terminal. Los métodos default permitieron agregarles `stream()` a las colecciones que ya existían sin romper a nadie. `Optional` deja que un método diga en su tipo de retorno que el resultado puede faltar, y `java.time` trae fechas inmutables con meses del 1 al 12. Nada de esto es obligatorio: un bucle o un `null` bien comprobado siguen siendo la mejor opción cuando aclaran el código.

Java 11: menos ceremonia, más JDK

Java 11 salió en septiembre de 2018 y es la primera LTS después de Java 8. También es la primera del calendario actual: el JEP 322 fijó una versión cada seis meses, y lo explica con un ejemplo, la de marzo de 2018 es JDK 10 y la de septiembre de 2018, JDK 11. Java 9 y 10 no fueron LTS: cada una quedó reemplazada por la siguiente en cuanto salió (Oracle Java SE Support Roadmap).

Por eso este capítulo reúne lo que nació en 9, 10 y 11. Si tu equipo saltó de 8 a 11, todo esto le llegó de golpe. Ninguna de estas novedades cambia la forma de pensar un programa como lo hicieron los Streams: casi todas quitan trámites que escribías sin pensar.

`var`: deja de escribir el tipo que ya se ve

`var` · lenguaje · definitiva en Java 10 · primera LTS: Java 11 · JEP 286

Imagina que quieres agrupar los importes de los pedidos por cliente e imprimirlos en orden alfabético.

Por qué nace. En Java 8, cada variable local lleva su tipo escrito delante, aunque la misma línea ya lo diga. Con genéricos (los tipos con parámetros entre `< >`, como `List<Integer>`) el tipo se repite y crece: `Map<String, List<Integer>>` aparece en la declaración, y recorrer el mapa pide escribir `Map.Entry<String, List<Integer>>`, un tipo que nadie quiere escribir ni leer.

El JEP 286 parte de esa queja: los desarrolladores protestan por el código repetitivo, y con buenos nombres el tipo de cada variable local suele sobrar. Agrega una observación menos obvia: tener que escribir cada tipo empuja a encadenar expresiones largas en vez de partirlas en variables con nombre, porque cada variable intermedia cuesta una línea de tipos.

El mismo agrupamiento, antes y después:

ANTES · JAVA 8

```
Antes.java
```

DESPUÉS · JAVA 11

```
Despues.java
```

```
static void
imprimirPorCliente(List<Pedido> pedidos)
{
    Map<String, List<Integer>>
importesPorCliente = new TreeMap<>();
    for (Pedido pedido : pedidos) {
        importesPorCliente

.computeIfAbsent(pedido.cliente, cliente
-> new ArrayList<>())
                .add(pedido.importe);
    }
    for (Map.Entry<String, List<Integer>>
entrada : importesPorCliente.entrySet())
{

System.out.println(entrada.getKey() + ":
" + entrada.getValue());
    }
}
```

```
static void
imprimirPorCliente(List<Pedido> pedidos)
{
    var importesPorCliente = new
TreeMap<String, List<Integer>>();
    for (var pedido : pedidos) {
        importesPorCliente

.computeIfAbsent(pedido.cliente, cliente
-> new ArrayList<>())
                .add(pedido.importe);
    }
    for (var entrada :
importesPorCliente.entrySet()) {

System.out.println(entrada.getKey() + ":
" + entrada.getValue());
    }
}
```

SALIDA

```
Ana: [120, 90]
Luis: [40, 180]
```

Uso un **`TreeMap`**, un mapa que mantiene sus claves ordenadas, para que los clientes salgan siempre en el mismo orden.

Qué es. **`var`** le pide al compilador que **deduzca** el tipo de una variable local a partir de su inicializador, la expresión a la derecha del **`=`**. A eso se le llama *inferencia de tipos*. La variable sigue teniendo un tipo fijo, decidido al compilar: **`importesPorCliente`** es un **`TreeMap<String, List<Integer>>`** y no puede recibir después otra cosa. Java no se vuelve un lenguaje dinámico: solo dejas de escribir lo que el compilador ya sabe.

Se usa en variables locales con inicializador y en los bucles **`for`**, el clásico y el mejorado. No sirve para atributos, parámetros de métodos ni tipos de retorno.

Por qué es mejor. El tipo aparece una sola vez, donde se crea el objeto, y en el bucle **`var entrada`** reemplaza a **`Map.Entry<String, List<Integer>> entrada`**. Lo que queda a la vista son los nombres, que es lo que lees primero.

Cuándo no usarla. Cuando el tipo no se deduce leyendo la línea. **`var total = calcular(pedidos);`** te obliga a ir a buscar qué devuelve **`calcular`**: ¿un **`int`**, un **`long`**, un **`BigDecimal`**? Con dinero, esa diferencia importa. Mi criterio: **`var`** cuando el lado derecho dice el tipo (un **`new`**, un literal, un método cuyo nombre no

deja dudas); el tipo escrito cuando no. Además, el compilador necesita algo de lo que deducir: no hay **`var`** sin inicializador ni con **`null`**.

> **– Para profundizar**
>
> **`var`** no es una palabra clave sino un *nombre de tipo reservado* (JEP 286): el código que ya lo usaba como nombre de variable sigue compilando. Infiere el tipo concreto: **`new TreeMap<...>()`** da un **`TreeMap`**, no un **`Map`**; si quieres programar contra la interfaz, escríbela. Desde Java 11 sirve también en los parámetros de una lambda (JEP 323), para poder anotarlos sin escribir sus tipos: **`(@Nonnull var x, @Nullable var y) -> x.process(y)`**. Es todo o nada: o todos los parámetros llevan **`var`**, o ninguno.

Ejercicio. ¿Qué tipo tiene **`estados`** en **`var estados = new ArrayList<>();`**? ¿Compila agregarle un **`"PAGADO"`** y después un **`42`**?

> **– Solución**
>
> compila, y ese es el problema. El diamante **`<>`** deduce los parámetros de tipo a partir del contexto, casi siempre el tipo de la variable que recibe el objeto. Con **`var`** ese tipo todavía no existe, así que queda **`ArrayList<Object>`**.
>
> Ejercicio.java
>
> ```
> var estados = new ArrayList<>();   // el diamante no tiene de dónde deducir: ArrayList<Object>
> estados.add("PAGADO");
> estados.add(42);                   // compila: también es un Object
> // String primero = estados.get(0); // no compilaría: get devuelve Object
> ```
>
> Imprime **`Elementos: [PAGADO, 42]`** y **`Tipo del primero al leerlo: Object`**: una sobrecarga que el compilador escoge según el tipo de **`estados.get(0)`** confirma que es **`Object`**. Con **`var`**, escribe el tipo dentro del diamante: **`new ArrayList<String>()`**.

Colecciones pequeñas e inmodificables en una sola expresión

`List.of`, **`Set.of`**, **`Map.of`** · API · definitiva en Java 9 · primera LTS: Java 11 · JEP 269

La tienda tiene datos que no deberían cambiar nunca: los estados posibles de un pedido y el costo de envío por zona.

Por qué nace. En Java 8 no había forma de escribir una colección pequeña e inmodificable en una sola expresión. Para una lista, lo más corto era envolver **`Arrays.asList`** con **`Collections.unmodifiableList`**. Para un mapa, ni eso: había que crear un **`HashMap`**, llamar a **`put`** una vez por entrada y envolverlo. Como eso no cabe en la declaración de una constante, terminaba en un bloque **`static`**, separado del campo que llena.

El JEP 269 lo dice así: Java se critica por verboso, y las colecciones estáticas tienen que poblarse en bloques de inicialización en vez de en la declaración del campo. Agrega un detalle que importa: los atajos de entonces no creaban colecciones inmodificables por sí mismos. **`Collections.unmodifiableList`** devuelve una *vista* de solo lectura de otra lista (javadoc de **`Collections`**): si alguien conserva la original y la cambia, el cambio se ve a través de la vista.

Las mismas constantes, antes y después:

ANTES · JAVA 8

Antes.java

```
static final List<String> ESTADOS =

Collections.unmodifiableList(Arrays.asLis
t("PENDIENTE", "PAGADO", "ENVIADO"));

static final Map<String, Integer>
COSTO_ENVIO;
static {
    Map<String, Integer> costos = new
HashMap<>();
    costos.put("NACIONAL", 5);
    costos.put("INTERNACIONAL", 20);
    COSTO_ENVIO =
Collections.unmodifiableMap(costos);
}
```

DESPUÉS · JAVA 11

Despues.java

```
static final List<String> ESTADOS =
List.of("PENDIENTE", "PAGADO",
"ENVIADO");

static final Map<String, Integer>
COSTO_ENVIO = Map.of("NACIONAL", 5,
"INTERNACIONAL", 20);
```

SALIDA

```
Estados: [PENDIENTE, PAGADO, ENVIADO]
Envío internacional: 20
Agregar un estado: UnsupportedOperationException
```

Las dos versiones imprimen lo mismo, y en las dos intentar agregar un estado falla.

Qué es. **`List.of`**, **`Set.of`** y **`Map.of`** son *métodos fábrica*: métodos estáticos que construyen el objeto por ti y te lo devuelven. Reciben los elementos (en **`Map.of`**, clave y valor alternados, hasta diez pares; para más está **`Map.ofEntries`**) y devuelven una colección **inmodificable**: no se puede agregar, quitar ni reemplazar

nada, y cualquier intento lanza **`UnsupportedOperationException`**, la excepción que usa Java cuando una colección no admite una operación (javadoc de **`List`** y **`Set`**).

Por qué es mejor. La constante se declara y se llena en el mismo lugar, y no existe una colección original por ahí que alguien pueda seguir modificando. Lo que lees es exactamente lo que contiene.

Cuándo no usarla. Tres límites concretos:

- Cuando necesitas modificar la colección. Para partir de unos valores y después agregar, copia: **`new ArrayList<>(List.of(...))`**.
- Cuando puede haber **`null`**. No los admiten: crear una con un **`null`** lanza **`NullPointerException`** (javadoc de **`List`**, **`Set`** y **`Map`**).
- Cuando te importa el orden de **`Set.of`** o **`Map.of`**. Su orden de iteración no está especificado y puede cambiar (javadoc de **`Set`** y **`Map`**). **`List.of`** sí conserva el orden de sus argumentos. Por eso mi ejemplo imprime la lista entera, pero del mapa solo consulta una clave. Si necesitas orden, usa un **`LinkedHashMap`** o un **`TreeMap`**.

> **– Para profundizar**
>
> **`Set.of`** y **`Map.of`** rechazan los duplicados al crearse, con **`IllegalArgumentException`**: **`Map.of("NACIONAL", 5, "NACIONAL", 7)`** compila, pero falla en cuanto se ejecuta. Las instancias son *value-based*: el javadoc pide no fiarse de su identidad (**`==`**, **`synchronized`**). Y lo inmodificable es la colección, no sus elementos: si guardas objetos mutables, su contenido puede cambiar.

Ejercicio. **`Arrays.asList("PAGADO", null)`** guarda el **`null`** sin quejarse. ¿Qué pasa con **`List.of("PAGADO", null)`**? ¿Y si a una lista creada con **`List.of`** le preguntas **`contains(null)`**?

> **– Solución**
>
> las dos cosas lanzan **`NullPointerException`**.

Ejercicio.java

```
List<String> conArrays = Arrays.asList("PAGADO", null);
System.out.println("Arrays.asList con null: " + conArrays);

try {
    List.of("PAGADO", null);
} catch (NullPointerException e) {
    System.out.println("List.of con null: NullPointerException");
}

List<String> estados = List.of("PENDIENTE", "PAGADO");
```

```
try {
    estados.contains(null);
} catch (NullPointerException e) {
    System.out.println("contains(null): NullPointerException");
}
```

Imprime **`Arrays.asList con null: [PAGADO, null]`**, **`List.of con null: NullPointerException`** y **`contains(null): NullPointerException`**. La primera excepción la garantiza el javadoc de **`List`**. La segunda sorprende más: el javadoc de **`Collection`** permite, como opcional, que **`contains`** la lance en colecciones que no admiten **`null`**, y la implementación del JDK lo hace.

Un cliente HTTP moderno dentro del JDK

`HttpClient` · API · definitiva en Java 11 · primera LTS: Java 11 · JEP 321 · incubadora en Java 9 (JEP 110) y 10

La tienda le pregunta a otro servicio, por HTTP, en qué estado está el pedido 42.

Por qué nace. Hasta Java 10, la API estándar del JDK para hablar HTTP era **`HttpURLConnection`**, y el JEP 110 enumera sus problemas: se diseñó para varios protocolos que hoy ya no se usan (como ftp y gopher), es anterior a HTTP/1.1 y demasiado abstracta, es difícil de usar, tiene comportamientos sin documentar y es difícil de mantener. Y solo funciona en modo *bloqueante*: el hilo que hace la petición se queda parado hasta que llega la respuesta, un hilo por cada petición.

En el código se nota: hay que convertir la conexión con un cast, fijar el método con un **`String`**, leer el cuerpo como un flujo de bytes, cerrarlo y desconectar. El JEP 110 entregó un cliente nuevo en Java 9 como *incubadora* (una API que viene con el JDK marcada como no definitiva, para probarla antes de fijarla); se actualizó en 10, y el JEP 321 lo estandarizó en Java 11, en el paquete **`java.net.http`**.

La misma consulta, antes y después:

ANTES · JAVA 8

Antes.java

```
static void consultarEstado(String url)
throws IOException {
    HttpURLConnection conexion =
(HttpURLConnection) new
URL(url).openConnection();
    try {
        conexion.setRequestMethod("GET");
        int status =
conexion.getResponseCode();
        String cuerpo;
        try (BufferedReader lector = new
BufferedReader(
```

DESPUÉS · JAVA 11

Despues.java

```
static void consultarEstado(HttpClient
cliente, String url)
        throws IOException,
InterruptedException {
    HttpRequest peticion =
HttpRequest.newBuilder(URI.create(url)).G
ET().build();
    HttpResponse<String> respuesta =
            cliente.send(peticion,
HttpResponse.BodyHandlers.ofString());
    System.out.println("Status: " +
respuesta.statusCode());
```

```
                new
InputStreamReader(conexion.getInputStream
(), StandardCharsets.UTF_8))) {
            cuerpo =
lector.lines().collect(Collectors.joining
("\n"));
        }
        System.out.println("Status: " +
status);
        System.out.println("Cuerpo: " +
cuerpo);
    } finally {
        conexion.disconnect();
    }
}
```

```
    System.out.println("Cuerpo: " +
respuesta.body());
}
```

SALIDA

```
Status: 200
Cuerpo: ENVIADO
```

Para que el ejemplo no dependa de la red, el programa completo levanta su propio servidor, un **`com.sun.net.httpserver.HttpServer`** en un puerto libre de la máquina local, y lo apaga al terminar. Es igual en las dos versiones, así que no lo muestro en el comparador.

Qué es. La API tiene tres piezas. **`HttpClient`** guarda la configuración y los recursos que comparten todas las peticiones; una vez creado no cambia, y sirve para enviar muchas (javadoc de **`HttpClient`**). **`HttpRequest`** describe lo que pides: la dirección, el método y las cabeceras. **`HttpResponse`** trae lo que vuelve: el código de estado, las cabeceras y el cuerpo. El *body handler*, aquí **`BodyHandlers.ofString()`**, dice en qué convertir el cuerpo.

Hay dos formas de enviar: **`send`** bloquea hasta tener la respuesta, como en el ejemplo, y **`sendAsync`** vuelve en el acto con un **`CompletableFuture`** que se completa cuando llega la respuesta.

Por qué es mejor. No hay cast ni flujos que cerrar a mano: el cuerpo llega convertido. Cada pieza tiene un nombre que dice qué es, y la forma asíncrona viene incluida en vez de obligarte a montar tus propios hilos.

Cuándo no usarlo. Si tu proyecto ya usa el cliente HTTP de un framework, con reintentos, métricas y configuración centralizada, cambiarlo solo por usar el del JDK no te da nada; mi criterio es usarlo en código nuevo sin ese ecosistema. Y conoce sus valores por defecto: no sigue redirecciones (la política es **`NEVER`**), así que un 301 llega tal cual a menos que configures **`followRedirects`**.

> – Para profundizar

> el cliente prefiere HTTP/2 por defecto, y su javadoc aclara que algunas condiciones, como ciertos proxys, pueden hacer que use HTTP/1.1. El servidor de mi ejemplo solo habla HTTP/1.1, y la petición funciona sin configurar nada. **`send`** declara **`InterruptedException`** porque bloquea el hilo: si alguien interrumpe ese hilo mientras espera, te enteras.

Métodos de `String` para lo que hacías a mano

`isBlank`, **`lines`**, **`strip`**, **`repeat`** · API · definitiva en Java 11 · primera LTS: Java 11 · sin JEP (javadoc de **`String`**, **`@since 11`**)

El cliente puede escribir notas en su pedido, en varias líneas, con espacios de sobra y alguna línea vacía. Quieres imprimirlas limpias en el ticket, entre dos separadores.

Por qué nace. Estos métodos no tienen JEP, así que no hay un texto oficial con su motivación: lo que sigue es mi lectura. Eran operaciones que casi todo proyecto necesitaba y que en Java 8 se armaban con piezas pensadas para otra cosa: partir en líneas con una expresión regular (**`split("\\R")`**), saber si un texto está en blanco con **`trim().isEmpty()`**, que recorta primero y pregunta después, o repetir un guion con **`String.join`** y **`Collections.nCopies`**, que hay que leer dos veces.

Y **`trim`** tiene una definición de "espacio" muy suya: cualquier carácter con código menor o igual a U+0020 (javadoc de **`String`**). Eso deja fuera los espacios de Unicode, que llegan, por ejemplo, en texto copiado de otro programa.

Las mismas notas, antes y después:

ANTES · JAVA 8

Antes.java

```
static List<String> notasUtiles(String
notas) {
    return
Arrays.stream(notas.split("\\R"))
            .map(String::trim)
            .filter(linea ->
!linea.isEmpty())

.collect(Collectors.toList());
}

static String separador(int ancho) {
    return String.join("",
Collections.nCopies(ancho, "-"));
}
```

DESPUÉS · JAVA 11

Despues.java

```
static List<String> notasUtiles(String
notas) {
    return notas.lines()
            .filter(linea ->
!linea.isBlank())
            .map(String::strip)

.collect(Collectors.toList());
}

static String separador(int ancho) {
    return "-".repeat(ancho);
}
```

SALIDA

```
--------------------
Dejar en portería
Llamar antes de subir
--------------------
```

Para estas notas, las dos versiones dan lo mismo. En general no son idénticas, y el ejercicio de abajo muestra por qué.

Qué es. Cuatro métodos, todos definitivos en Java 11:

- **`isBlank()`** devuelve **`true`** si el texto está vacío o solo tiene espacios en blanco.
- **`lines()`** devuelve un Stream con las líneas del texto, sin el salto de línea. Reconoce **`\n`**, **`\r`** y **`\r\n`**, y un salto al final no produce una línea vacía extra.
- **`strip()`** quita los espacios en blanco del principio y del final; **`stripLeading()`** y **`stripTrailing()`**, solo de un lado.
- **`repeat(n)`** concatena el texto **`n`** veces. Con **`0`** devuelve **`""`**; con un número negativo lanza **`IllegalArgumentException`**.

"Espacio en blanco", en los tres primeros, es lo que dice **`Character.isWhitespace`**, y ahí está la diferencia con **`trim`**. **`trim`** quita todo carácter con código hasta U+0020: el espacio, el tabulador, los saltos de línea y también controles como U+0000. **`strip`** quita el espacio, el tabulador, los saltos y los separadores de Unicode, como el espacio eme (U+2003), pero no la mayoría de los controles, como U+0000, ni los espacios de no separación (U+00A0, U+2007, U+202F) (javadoc de **`Character`**). Ninguno de los dos contiene al otro.

Por qué es mejor. Cada método dice su intención: **`isBlank`** pregunta, **`strip`** limpia, **`repeat`** repite. **`lines`** evita la expresión regular, y su javadoc agrega que rinde mejor que **`split("\\R")`**, porque entrega las líneas a medida que se piden y busca los saltos más rápido.

Cuándo no usarlos. **`strip`** no es "limpiar todo". Si tus datos vienen de HTML, el **` `** es U+00A0 y **`strip`** lo conserva: tienes que reemplazarlo tú. Tampoco cambies **`trim`** por **`strip`** en un sistema que ya funciona sin revisar qué datos procesa: quitan cosas distintas. Y **`lines`** reconoce menos saltos que **`\R`**, que también acepta U+0085, U+2028 y U+2029, entre otros (javadoc de **`Pattern`**).

Ejercicio. El texto **`"\u2003PAGADO\u2003"`** tiene un espacio eme a cada lado. ¿Cuántos caracteres quedan después de **`trim()`**? ¿Y después de **`strip()`**?

> – Solución
>
> **`trim()`** no lo toca, porque U+2003 está por encima de U+0020; **`strip()`** sí lo quita.
>
> Ejercicio.java

```
String estado = "\u2003PAGADO\u2003";   // U+2003: espacio eme, un espacio Unicode
System.out.println("trim():  " + estado.trim().length() + " caracteres");
System.out.println("strip(): " + estado.strip().length() + " caracteres");
System.out.println("isBlank() de un espacio eme: " + "\u2003".isBlank());

String conNoSeparable = "\u00A0PAGADO";  // U+00A0: espacio de no separación
System.out.println("strip() con no separable: " + conNoSeparable.strip().length() +
" caracteres");
```

Imprime **`trim(): 8 caracteres`**, **`strip(): 6 caracteres`**, **`isBlank() de un espacio eme: true`** y **`strip() con no separable: 7 caracteres`**. La última línea es la trampa: el espacio de no separación sobrevive a **`strip`**.

Ejecutar un archivo `.java` sin compilarlo antes

`java Hola.java` · herramienta · definitiva en Java 11 · primera LTS: Java 11 · JEP 330

Quieres una utilidad rápida que sume los importes de unos pedidos que le pasas por la línea de comandos.

Por qué nace. Hasta Java 10, ejecutar un programa siempre tomaba dos pasos: **`javac`** compila el archivo fuente y deja un **`.class`** en el disco, y después **`java`** ejecuta la clase, por su nombre y no por el del archivo. El JEP 330 parte de que los programas de un solo archivo son comunes cuando aprendes Java y en utilidades pequeñas, y en ese contexto compilar antes de ejecutar es "pura ceremonia". Un archivo, además, puede generar varios **`.class`**.

El paso extra también confunde al que empieza. En Java 8, **`java Hola.java`** responde **`Error: Could not find or load main class Hola.java`**: busca una clase llamada así y no la encuentra. Esa es la salida real que obtuve con el JDK 8 de esta guía.

El programa es un solo archivo:

Hola.java

```
public class Hola {
    public static void main(String[] args) {
        int total = 0;
        for (String importe : args) {
            total += Integer.parseInt(importe);
        }
        System.out.println("Pedidos: " + args.length + ", total: " + total);
    }
}
```

Y así se ejecuta, antes y después:

ANTES · JAVA 8

Terminal

```
$ javac Hola.java
$ java Hola 120 80 40
```

DESPUÉS · JAVA 11

Terminal

```
$ java Hola.java 120 80 40
```

SALIDA

```
Pedidos: 3, total: 240
```

Estos dos bloques no son código Java sino comandos de terminal. Los ejecuté en los JDK 8 y 11 de esta guía, cada uno en un contenedor sin red, y la salida es la que obtuve en los dos.

Qué es. Desde Java 11, el lanzador **`java`** tiene un modo de *archivo fuente*: si le pasas un **`.java`**, lo compila en memoria y ejecuta la primera clase de nivel superior del archivo, que tiene que tener el método **`public static void main(String[])`** de siempre (JEP 330). No escribe ningún **`.class`** en el disco. Lo que va después del nombre del archivo llega a **`main`** como argumentos.

Por qué es mejor. Un paso en vez de dos, y nada que limpiar después. Para aprender, probar una API o escribir una utilidad de un rato, el ciclo entre escribir y ver el resultado se acorta. En sistemas tipo Unix, además, el archivo puede empezar con una línea *shebang* (**`#!/ruta/a/java --source 11`**) y ejecutarse como un script; en ese caso el archivo no lleva la extensión **`.java`**.

Cuándo no usarlo. Cuando el programa crece más allá de un archivo. En Java 11, este modo no busca ni compila ningún otro archivo fuente: si tu clase usa otra que está en otro **`.java`**, no la encuentra. En cuanto hay varios archivos o dependencias, vuelves a **`javac`** o a una herramienta de construcción como Maven o Gradle. Y como no guarda nada, compila de nuevo en cada ejecución.

> – Para profundizar
>
> la opción **`--source`** fija la versión del lenguaje con la que se compila el archivo, y es obligatoria cuando el archivo no termina en **`.java`**, como en los scripts con shebang. El límite de un solo archivo lo levantó el JEP 458, en Java 22, que permite programas de varios archivos fuente. Y Java 25 lleva la idea de quitar ceremonia todavía más lejos.

También en Java 11. Tres cambios que no tienen ficha propia, pero que conviene conocer. El **sistema de módulos** (JPMS, JEP 261, sobre el diseño del JDK modular del JEP 200), definitivo en Java 9, permite declarar en un **`module-info.java`** qué paquetes exporta un módulo y de qué otros depende; el JDK mismo se partió en módulos. Tu código puede ignorarlo y seguir en el classpath como siempre, pero el JDK empezó a encapsular sus APIs internas, un camino que se endurece en Java 17. **JShell** (JEP 222, Java 9) es una consola interactiva para probar Java línea a línea sin escribir una clase; según su JEP, la razón número

uno que daban las escuelas para dejar Java como lenguaje de enseñanza era que otros lenguajes tenían una consola así y un "Hola, mundo" mucho más simple. Y el JEP 320 **retiró del JDK** en Java 11 los módulos de Java EE y CORBA, que estaban marcados para eliminación desde Java 9: JAX-WS, JAXB, JAF, Common Annotations, CORBA y **`java.transaction`**, la parte de JTA para interoperar con CORBA. Java EE no desapareció: esas tecnologías tienen versiones independientes, por ejemplo en Maven Central. Al migrar de 8 a 11 lo vas a notar así: un **`import javax.xml.bind.JAXBContext`** que compila en Java 8 falla en Java 11 con **`package javax.xml.bind does not exist`**, hasta que agregas la dependencia.

> **Lo que debes llevarte:** Java 11 quita trámites. **`var`** deja que el compilador escriba el tipo que ya se ve en la línea, sin volver a Java dinámico. **`List.of`**, **`Set.of`** y **`Map.of`** crean colecciones inmodificables en una expresión, sin **`null`** y, en **`Set`** y **`Map`**, sin un orden garantizado. **`HttpClient`** reemplaza a **`HttpURLConnection`** con una API clara, síncrona o asíncrona. Los métodos nuevos de **`String`** dicen su intención, pero **`strip`** no es **`trim`**. Y **`java Hola.java`** ejecuta un programa de un archivo sin compilarlo antes.

Java 17: datos y decisiones sin ceremonia

Java 17 salió en septiembre de 2021, tres años después de Java 11. En medio hubo cinco versiones, de la 12 a la 16, y casi todo lo que verás en este capítulo llegó primero como preview: había que activarlo a mano, y fue cambiando con lo que opinaban quienes lo usaban. Java 17 es la primera LTS que lo trae en forma definitiva.

Las cinco novedades de este capítulo tienen un hilo común: menos ceremonia para dos cosas que haces todos los días. Describir datos (text blocks, records) y decidir qué hacer según el dato que tienes (switch, **`instanceof`**, sealed classes). Las tres que sirven para decidir son, además, las piezas que Java 21 junta en una sola.

Switch expressions: decidir un valor sin `break` olvidados

Switch expressions · lenguaje · preview en Java 12 y 13 · definitiva en Java 14 · primera LTS: Java 17 · JEP 361

Cada pedido tiene un estado (pendiente, pagado, enviado, entregado o cancelado), y la tienda le muestra al cliente un mensaje distinto según ese estado.

Por qué nace. El **`switch`** de Java viene de C y hereda su regla más traicionera: cada **`case`** es solo un punto de entrada. La ejecución entra en uno y sigue de largo por los de abajo hasta encontrar un **`break`**. A eso se le llama *fall-through* (caer al siguiente). A veces se usa a propósito, para que dos estados compartan mensaje, pero olvidar un **`break`** compila sin una sola queja.

Además, **`switch`** era solo una sentencia: no producía un valor. Para obtener el mensaje hay que declarar la variable antes, asignarla en cada rama y agregar un **`default`** aunque los cinco estados estén cubiertos,

porque sin él el compilador no acepta que la variable tenga valor. El JEP 361 suma un tercer problema: todo el bloque es un solo ámbito, así que una variable declarada en un **case** se ve en los demás.

ANTES · JAVA 11

Antes.java

```
static String
mensajeParaCliente(EstadoPedido estado) {
    String mensaje;
    switch (estado) {
        case PENDIENTE:
        case PAGADO:
            mensaje = "Todavía puedes
cancelarlo";
            break;
        case ENVIADO:
            mensaje = "Va en camino";
            break;
        case ENTREGADO:
            mensaje = "Ya lo recibiste";
            break;
        case CANCELADO:
            mensaje = "Se canceló";
            break;
        default:
            throw new
IllegalStateException("Estado
desconocido: " + estado);
    }
    return mensaje;
}
```

DESPUÉS · JAVA 17

Despues.java

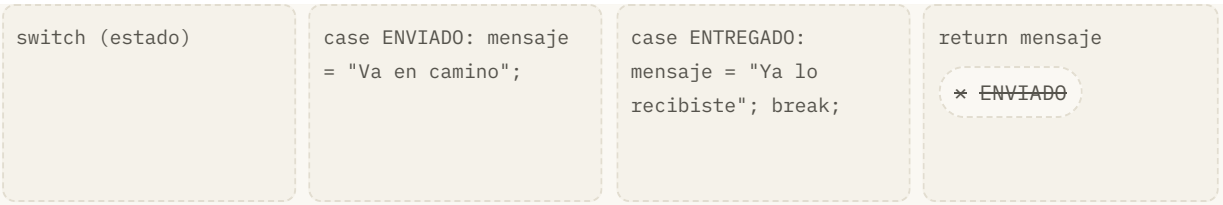
```
static String
mensajeParaCliente(EstadoPedido estado) {
    return switch (estado) {
        case PENDIENTE, PAGADO ->
"Todavía puedes cancelarlo";
        case ENVIADO -> "Va en camino";
        case ENTREGADO -> "Ya lo
recibiste";
        case CANCELADO -> "Se canceló";
    };
}
```

SALIDA

```
PENDIENTE: Todavía puedes cancelarlo
PAGADO: Todavía puedes cancelarlo
ENVIADO: Va en camino
ENTREGADO: Ya lo recibiste
CANCELADO: Se canceló
```

Para ver el riesgo, borré el **break** que sigue a **`"Va en camino"`** en el "antes" y lo volví a ejecutar en Java 11. Compiló sin avisos, y el pedido enviado pasó a decir "Ya lo recibiste". Recórrelo paso a paso:

Un break olvidado en el switch de Java 11



Llega un pedido en estado ENVIADO. El switch salta directamente a la etiqueta que coincide.

Se asigna "Va en camino". Falta el break, así que la ejecución no sale del switch: sigue en la línea siguiente, aunque pertenezca a otro case.

Se asigna "Ya lo recibiste" encima del mensaje anterior. Solo este break saca la ejecución del switch.

El método devuelve "Ya lo recibiste" para un pedido que todavía va en camino. Nada falló: el programa hizo exactamente lo que dice el código.

Qué es. Una **switch expression** es un **`switch`** que produce un valor: se puede poner después de un **`return`** o a la derecha de un **`=`**. Con las etiquetas nuevas **`case X ->`** solo se ejecuta lo que está a la derecha de la flecha: no hay fall-through y no hace falta **`break`**. Varias constantes que comparten resultado van en el mismo **`case`**, separadas por comas.

Además, una switch expression tiene que ser **exhaustiva**: cubrir todos los valores posibles. Con un **`enum`**, si nombras todas sus constantes, no necesitas **`default`**.

Por qué es mejor. Se lee como una tabla: estado, flecha, mensaje. Y la exhaustividad se vuelve una red de seguridad. Agregué un estado **`DEVUELTO`** al **`enum`** sin tocar los **`switch`**. El "antes" compiló y solo falló en ejecución, al llegar el primer pedido devuelto (**`IllegalStateException: Estado desconocido: DEVUELTO`**). El "después" ni siquiera compiló:

```
Despues.java:8: error: the switch expression does not cover all possible input values
        return switch (estado) {
               ^
```

Cuándo no usarla. Cuando cada rama hace algo (guardar, enviar un correo) en vez de producir un valor, lo que necesitas es una sentencia, y ahí también puedes usar las flechas. Y en Java 17 **`switch`** solo compara contra constantes: enteros **`int`** o más pequeños, caracteres, **`String`** o un **`enum`**. Para decidir por rangos o por tipos, todavía necesitas **`if`**.

> – Para profundizar
>
> si una rama necesita varias líneas, va en un bloque **`{ ... }`** que termina con **`yield valor;`**. En el preview de Java 12 era **`break valor`**, y el JEP 354 lo cambió en Java 13 porque confundía. El JEP 361 advierte que agregar un **`default`** "por si acaso" en un **`switch`** sobre un **`enum`** apaga la revisión: el **`default`** cubre en silencio la constante nueva. Para el código viejo, **`javac -Xlint:fallthrough`** avisa de cada caída entre **`case`**; lo comprobé con el **`break`** borrado.

Text blocks: el texto de varias líneas, tal como se lee

Text blocks · lenguaje · preview en Java 13 y 14 · definitivos en Java 15 · primera LTS: Java 17 · JEP 378

La tienda le avisa al almacén de cada pedido nuevo con un JSON: número, cliente e importe.

Por qué nace. Un literal de texto de Java (**`"..."`**) no puede ocupar más de una línea. Para escribir uno de varias, había que cortarlo en pedazos, unirlos con **`+`**, cerrar cada uno con **`\n`** y escapar cada comilla como **`\"`**. El JSON queda enterrado entre símbolos que no son parte de él, y un **`\n`** de menos no da error: junta dos líneas en silencio. El JEP 378 lo plantea con HTML, XML, SQL y JSON: textos que casi todo programa incrusta y que así quedan difíciles de leer y de mantener.

ANTES · JAVA 11

Antes.java

```
static String pedidoComoJson(int numero,
String cliente, int importe) {
    return "{\n"
            + "  \"pedido\": " + numero +
",\n"
            + "  \"cliente\": \"" +
cliente + "\",\n"
            + "  \"importe\": " + importe
+ "\n"
            + "}\n";
}
```

DESPUÉS · JAVA 17

Despues.java

```
static String pedidoComoJson(int numero,
String cliente, int importe) {
    return """
            {
              "pedido": %d,
              "cliente": "%s",
              "importe": %d
            }
            """.formatted(numero,
cliente, importe);
}
```

SALIDA

```
{
  "pedido": 1001,
  "cliente": "Ana",
  "importe": 120
}
Caracteres: 59
Empieza sin sangría: true
Termina en salto de línea: true
```

Las dos versiones producen el mismo texto, carácter por carácter: los dos programas cuentan 59.

Qué es. Un **text block** es un literal de texto que abre con **`"""`** seguido de un salto de línea y cierra con **`"""`**. Entre los dos delimitadores va el contenido: los saltos de línea son reales y las comillas van sin escapar. El resultado es un **`String`** como cualquier otro. Aquí uso **`formatted`** (Java 15), que equivale a **`String.format`** con el texto como plantilla, para poner los valores en los **`%d`** y **`%s`**.

Fíjate en la sangría. En el código, las líneas del JSON están corridas a la derecha para alinearse con el `return`, pero la salida empieza en el margen. El compilador separa dos tipos de espacios:

- **Incidentales**: los que pusiste para que el texto se alinee con tu código. Los quita: le resta a cada línea la misma cantidad, hasta que la línea menos sangrada toca el margen.
- **Esenciales**: los que forman parte del texto. Los dos espacios antes de `"pedido"` sobreviven porque están *además* de la sangría común.

El `"""` de cierre también cuenta. Como lo dejé solo en su línea, el texto termina en un salto de línea, igual que el `"}\n"` del "antes". Si lo pones pegado a la `}`, el texto termina sin salto. Y si lo corres a la izquierda, se conserva más sangría.

Por qué es mejor. Lees el JSON tal como sale, y puedes pegarlo desde un archivo o una documentación sin escapar nada.

Cuándo no usarlo. Un text block no inserta valores ni los escapa. Si el nombre del cliente trae una comilla, el JSON se rompe en las dos versiones. Para generar JSON a partir de datos, usa una biblioteca de JSON; los text blocks brillan con texto fijo: consultas SQL con parámetros `?`, datos de prueba, mensajes largos.

> – **Para profundizar**
>
> el compilador normaliza los saltos de línea a LF (`\n`) aunque el archivo se haya editado en Windows, y borra los espacios al final de cada línea. El segundo preview (Java 14) agregó dos escapes para controlar eso: `\s` (un espacio que no se borra) y `\` al final de una línea (la une con la siguiente). El algoritmo de la sangría está disponible como `String.stripIndent()`.

Records: una clase de datos en una línea

Records · lenguaje · preview en Java 14 y 15 · definitivos en Java 16 · primera LTS: Java 17 · JEP 395

Un pedido, como dato, es un número, un cliente y un importe. Nada más.

Por qué nace. Escribir bien esa clase en Java 11 exige campos `private final`, un constructor, un *getter* por campo y tres métodos más: `equals` (cuándo dos pedidos son iguales), `hashCode` (un número que debe coincidir cuando `equals` dice que sí; de él dependen `HashSet` y `HashMap`) y `toString` (cómo se imprime). Cinco lugares que repiten los mismos tres nombres. El JEP 395 lo llama código repetitivo, de poco valor y propenso a errores: si mañana agregas un campo `fecha` y no lo sumas a `equals`, dos pedidos distintos pasan a ser "iguales" y nadie te avisa. El IDE lo genera, pero alguien lo tiene que leer y mantener.

ANTES · JAVA 11

Antes.java

DESPUÉS · JAVA 17

Despues.java

```
static final class Pedido {
    private final int numero;
    private final String cliente;
    private final int importe;

    Pedido(int numero, String cliente,
int importe) {
        this.numero = numero;
        this.cliente = cliente;
        this.importe = importe;
    }

    int getNumero() {
        return numero;
    }

    String getCliente() {
        return cliente;
    }

    int getImporte() {
        return importe;
    }

    @Override
    public boolean equals(Object otro) {
        if (this == otro) {
            return true;
        }
        if (!(otro instanceof Pedido)) {
            return false;
        }
        Pedido pedido = (Pedido) otro;
        return numero == pedido.numero
                && importe ==
pedido.importe
                &&
Objects.equals(cliente, pedido.cliente);
    }

    @Override
    public int hashCode() {
        return Objects.hash(numero,
cliente, importe);
    }

    @Override
    public String toString() {
```

```
record Pedido(int numero, String cliente,
int importe) { }
```

```
        return "Pedido[numero=" + numero
+ ", cliente=" + cliente + ", importe=" +
importe + "]";
    }
}
```

SALIDA

```
Pedido[numero=1001, cliente=Ana, importe=120]
Cliente: Ana
Mismos datos, equals: true
Mismos datos, ==: false
Pedidos distintos en un Set: 1
```

Al **`toString`** del "antes" le di a propósito el formato que genera el record, para que las salidas coincidan.

Qué es. Un **record** es una clase que declara sus datos en la cabecera: **`record Pedido(int numero, String cliente, int importe)`**. Cada dato es un *componente*, y con ellos el compilador genera:

1. Un campo **`private final`** por componente.
2. Un constructor que los recibe en ese orden.
3. Un método de acceso con el mismo nombre: **`cliente()`**, no **`getCliente()`**.
4. **`equals`**, **`hashCode`** y **`toString`** basados en todos los componentes.

La salida lo muestra: **`equals`** compara datos, así que dos pedidos con los mismos valores son iguales aunque sean objetos distintos (**`==`** da **`false`**), y en un **`HashSet`** cuentan como uno. Puedes agregarle métodos, campos **`static`** y validaciones en un *constructor compacto*, sin lista de parámetros, como el del ejercicio.

Por qué es mejor. La declaración dice qué es un pedido, y el resto no puede desincronizarse: si agregas un componente, **`equals`**, **`hashCode`** y **`toString`** lo incluyen solos.

Cuándo no usarlo. Un record tiene límites a propósito:

- Es inmutable **solo en la superficie**: sus campos son **`final`**, pero una lista guardada en él puede cambiar.
- No puede extender otra clase (todos extienden **`java.lang.Record`**) y es **`final`**. Sí puede implementar interfaces.
- No admite más campos de instancia que sus componentes. Si el objeto cambia de estado con el tiempo, o su API no debe exponer cómo guarda los datos, usa una clase normal: según el JEP, un record renuncia a separar su API de su representación interna.

> – Para profundizar

convertir en record una clase que no sobrescribía **`equals`** cambia su comportamiento: dos objetos con los mismos datos pasan de distintos (se comparaba la identidad) a iguales, y un **`Set`** que guardaba dos pedidos así pasa a guardar uno. Antes de migrar, revisa dónde se comparan o se usan como clave. Y no parsees el **`toString`**: el javadoc de **`Record`** dice que su formato puede cambiar.

Ejercicio. El pedido ahora guarda sus líneas: **`record Pedido(int numero, List<String> lineas)`**. Si después de crearlo alguien agrega un elemento a la lista que le pasaste, ¿cambia el pedido? ¿Cómo lo evitas?

– Solución

sí cambia: **`final`** impide apuntar a otra lista, no modificar esta. La solución es copiarla en un constructor compacto, que corre antes de asignar los campos:

Ejercicio.java

```
record PedidoProtegido(int numero, List<String> lineas) {
    PedidoProtegido {
        lineas = List.copyOf(lineas);
    }
}
```

`List.copyOf` (Java 10) devuelve una copia inmodificable, y rechaza los elementos **`null`**. Tras agregar **`"Ratón"`** a la lista original, el programa imprime:

```
Pedido: [Teclado, Ratón]
PedidoProtegido: [Teclado]
PedidoProtegido no deja agregar líneas: UnsupportedOperationException
```

Pattern matching de `instanceof`: comprobar y convertir en un solo paso

Pattern matching de `instanceof` · lenguaje · preview en Java 14 y 15 · definitivo en Java 16 · primera LTS: Java 17 · JEP 394

Un pedido se paga con tarjeta, con transferencia o en efectivo, y el recibo describe el medio de pago que se usó.

Por qué nace. **`instanceof`** responde si un objeto es de un tipo, pero no te lo entrega como ese tipo. Después de preguntar, hay que hacer un *cast* (una conversión explícita, **`(Tarjeta) pago`**) y guardarlo en una variable nueva. Son tres pasos (comprobar, convertir, declarar) y el mismo tipo escrito tres veces. El JEP 394 lo llama tedioso, y apunta a lo más grave: esa repetición deja que los errores se cuelen sin que

nadie los note. Lo comprobé: si copias una rama, cambias el **instanceof** a **Transferencia** y te olvidas del cast a **Tarjeta**, Java 11 lo compila sin quejarse y falla en ejecución con una **ClassCastException**.

ANTES · JAVA 11

Antes.java

```
static String describir(MedioDePago pago)
{
    if (pago instanceof Tarjeta) {
        Tarjeta tarjeta = (Tarjeta) pago;
        return "Tarjeta terminada en " +
tarjeta.getUltimos4();
    }
    if (pago instanceof Transferencia) {
        Transferencia transferencia =
(Transferencia) pago;
        return "Transferencia con
referencia " +
transferencia.getReferencia();
    }
    return "Otro medio de pago";
}
```

DESPUÉS · JAVA 17

Despues.java

```
static String describir(MedioDePago pago)
{
    if (pago instanceof Tarjeta tarjeta)
{
        return "Tarjeta terminada en " +
tarjeta.ultimos4();
    }
    if (pago instanceof Transferencia
transferencia) {
        return "Transferencia con
referencia " +
transferencia.referencia();
    }
    return "Otro medio de pago";
}
```

SALIDA

```
Tarjeta terminada en 4242
Transferencia con referencia TRF-881
Otro medio de pago
¿El primero es la tarjeta 4242? true
¿El segundo es la tarjeta 4242? false
```

En el "después", **Tarjeta** y **Transferencia** son records: por eso se lee **ultimos4()** en vez de **getUltimos4()**. Las dos últimas líneas de la salida las produce el método **terminaEn**, que verás más abajo.

Qué es. `pago instanceof Tarjeta tarjeta` es un **patrón de tipo**: si **pago** es una **Tarjeta**, se convierte y queda guardado en la variable nueva **tarjeta**. Si no lo es, la condición da **false** y la variable no existe. Esa variable vive solo donde el compilador puede asegurar que el patrón coincidió: dentro del **if** y, en una condición con **&&**, a la derecha del **&&**:

Despues.java

```
static boolean terminaEn(MedioDePago pago, String digitos) {
    return pago instanceof Tarjeta tarjeta && tarjeta.ultimos4().equals(digitos);
}
```

Con `||` no compila, porque a la derecha no hay garantía de que **`pago`** fuera una tarjeta. A esta regla se le llama *alcance por flujo*: la variable existe donde el flujo del programa garantiza que tiene valor. También funciona al revés: tras un **`if (!(pago instanceof Tarjeta tarjeta)) return ...;`**, la variable se puede usar en el resto del método.

Por qué es mejor. Escribes el tipo una sola vez, y el cast equivocado deja de ser posible, porque ya no hay cast. Los métodos **`equals`** hechos a mano, como el del "antes" de records, se reducen a una expresión.

Cuándo no usarlo. Una cadena larga de **`instanceof`** suele ser un síntoma. Si preguntas por el tipo de un objeto en muchos lugares del programa, quizá ese comportamiento debería vivir dentro de cada clase, como un método **`describir()`** que cada medio de pago implementa a su manera. Y cuando eliges entre muchos tipos, el **`switch`** con patrones de Java 21 lo expresa mejor.

> – **Para profundizar**
>
> en la versión definitiva (Java 16) cambiaron dos cosas respecto de los previews: la variable del patrón dejó de ser implícitamente **`final`** (se le puede reasignar, aunque rara vez conviene), y pasó a ser un error de compilación usar un patrón que siempre coincide, por ejemplo cuando la expresión ya es de un subtipo del tipo del patrón. Es el primer paso de una serie: en Java 21 los patrones llegan al **`switch`** y aprenden a desarmar records.

Sealed classes: un conjunto cerrado que el compilador conoce

Sealed classes · lenguaje · preview en Java 15 y 16 · definitivas en Java 17 · primera LTS: Java 17 · JEP 409

La tienda acepta exactamente tres medios de pago: tarjeta, transferencia y efectivo al recibir el pedido.

Por qué nace. En la sección anterior, **`MedioDePago`** era una interfaz abierta: cualquier clase, en cualquier parte, podía implementarla. El código asume que hay tres, pero ese dato vive en tu cabeza y en el "Otro medio de pago" del final. Un **`enum`** no alcanza, porque cada variante trae datos distintos: la tarjeta sus cuatro dígitos, la transferencia su referencia. El JEP 409 lo resume así: un **`enum`** modela un conjunto fijo de *valores*; aquí hace falta un conjunto fijo de *tipos* de valores.

Java 11 ofrecía dos herramientas: **`final`**, que no deja ninguna subclase, o un constructor que no sea **`public`**, que limita quién hereda: al paquete o, si es privado, a las clases anidadas en la misma clase. Ese segundo truco es el del "antes". El JEP 409 lo llama frágil, y señala que con interfaces no funciona: una interfaz no tiene constructor.

ANTES · JAVA 11

Antes.java

```
abstract static class MedioDePago {
```

DESPUÉS · JAVA 17

Despues.java

```
sealed interface MedioDePago permits
Tarjeta, Transferencia, Efectivo { }
```

```
    private MedioDePago() { } // solo las
clases anidadas pueden heredar

    static final class Tarjeta extends
MedioDePago {
        final String ultimos4;

        Tarjeta(String ultimos4) {
            this.ultimos4 = ultimos4;
        }
    }

    static final class Transferencia
extends MedioDePago {
        final String referencia;

        Transferencia(String referencia)
{
            this.referencia = referencia;
        }
    }

    static final class Efectivo extends
MedioDePago { }
}
```

```
record Tarjeta(String ultimos4)
implements MedioDePago { }

record Transferencia(String referencia)
implements MedioDePago { }

record Efectivo() implements MedioDePago
{ }
```

SALIDA

```
Tarjeta terminada en 4242
Transferencia con referencia TRF-881
Efectivo al recibir el pedido
```

Qué es. Una clase o interfaz **sellada** (**`sealed`**) declara con **`permits`** la lista completa de sus subtipos directos. Cada subtipo permitido tiene que decir cómo sigue la jerarquía: **`final`** (no se extiende más; los records ya lo son), **`sealed`** (con su propia lista) o **`non-sealed`** (esa rama vuelve a quedar abierta). Todos tienen que estar en el mismo módulo o, si no usas módulos, en el mismo paquete. Si están en el mismo archivo, puedes omitir **`permits`** y el compilador los deduce.

Si alguien agrega un medio de pago sin sumarlo a la lista, como **`record Cupon(String codigo) implements MedioDePago`**, Java 17 no lo compila:

```
error: class is not allowed to extend sealed class: MedioDePago (as it is not listed in
its permits clause)
```

Por qué es mejor. El conjunto cerrado queda escrito en el propio tipo: lo lee quien abre el archivo y lo hace cumplir el compilador. Funciona con interfaces y con records, sin clases anidadas ni

constructores privados.

Lo que Java 17 todavía **no** hace es usar ese conocimiento al revisar tus decisiones. Mira cómo se describe un pago:

Despues.java

```
static String describir(MedioDePago pago) {
    if (pago instanceof Tarjeta tarjeta) {
        return "Tarjeta terminada en " + tarjeta.ultimos4();
    }
    if (pago instanceof Transferencia transferencia) {
        return "Transferencia con referencia " + transferencia.referencia();
    }
    if (pago instanceof Efectivo) {
        return "Efectivo al recibir el pedido";
    }
    throw new IllegalArgumentException("Medio de pago desconocido: " + pago);
}
```

El **`throw`** del final no se puede alcanzar, pero el compilador no lo sabe y, como reconoce el propio JEP 409, si borras una de las tres ramas no te avisa. La pieza que falta es el pattern matching de **`switch`**: tuvo su primer preview en este mismo Java 17 (JEP 406), quedó definitivo en Java 21 y, con una jerarquía sellada, comprueba que cubras todos los casos sin **`default`**. Lo verás en el capítulo de Java 21.

Cuándo no usarlas. Cuando otros tienen que poder agregar variantes (un sistema de plugins, una biblioteca que sus usuarios extienden), sellar impide justo eso. Y si las variantes no llevan datos propios, un **`enum`** es más simple.

> – **Para profundizar**
>
> el sellado no lo controla solo **`javac`**: la clase compilada guarda la lista de permitidas, y la JVM rechaza en ejecución una subclase que no esté en ella. Records y sealed se pensaron juntos: el JEP 409 los llama *tipos de datos algebraicos*. Un record es un dato hecho de varias partes; una jerarquía sellada, un dato que es una de varias formas.

También en Java 17. Tres cambios que no necesitan una sección entera. Las **NullPointerException explicativas** (JEP 358) dicen qué expresión era **`null`**, por ejemplo **`Cannot read field "c" because "a.b" is null`**: llegaron en Java 14 desactivadas por defecto y se activan solas desde Java 15. La **encapsulación fuerte de los internos del JDK** (JEP 403) cierra un endurecimiento que empezó en Java 9: de la 9 a la 15, el acceso por reflexión a los internos que ya existían en Java 8 se permitía por defecto, con una advertencia; Java 16 pasó a negarlo por defecto (JEP 396), y Java 17 dejó sin efecto la opción que lo relajaba (**`--illegal-access`**). Ahora solo **`--add-opens`** abre un paquete, y **`sun.misc.Unsafe`** sigue disponible. Si una

biblioteca vieja falla al migrar a 17 con un error de acceso por reflexión, es lo primero que conviene revisar. Y **`Stream.toList()`** (Java 16), que viste en el recuadro de Streams, devuelve una lista **inmodificable**.

> **Lo que debes llevarte:** Java 17 te deja escribir datos y decisiones con menos ceremonia. Los text blocks muestran el texto tal como es; los records declaran un dato y el compilador escribe el resto, aunque su inmutabilidad es solo superficial. Las switch expressions eliminan el fall-through y avisan si falta un caso; el pattern matching de **`instanceof`** junta la comprobación y el cast; las sealed classes cierran un conjunto de tipos. Estas tres últimas son piezas sueltas en Java 17: Java 21 las junta en el **`switch`** con patrones.

Java 21: concurrencia barata y datos que se desarman

Java 21 salió en septiembre de 2023 y reúne lo que maduró entre Java 18 y 21. Trae dos historias. La primera es de concurrencia: atender miles de tareas que esperan sin necesitar miles de hilos del sistema operativo. La segunda cierra lo que empezó en Java 17: allí aprendiste a modelar datos con records y sealed classes; aquí el lenguaje te deja preguntar por su forma y desarmarlos en una sola línea.

Hilos virtuales: miles de esperas sin miles de hilos

Hilos virtuales (virtual threads) · JVM · definitiva en Java 21 · primera LTS: Java 21 · JEP 444 · preview en Java 19 y 20

La tienda tiene que cobrar 1000 pedidos, y cada cobro consulta por red al servicio de pagos. En el ejemplo, **`consultarPago`** simula esa llamada: espera 20 ms con **`Thread.sleep`** y devuelve un importe que depende solo del id del pedido.

Por qué nace. Un **hilo** es una línea de ejecución independiente. El estilo más simple para un servidor es "un hilo por petición": cada pedido corre en su hilo de principio a fin, con código secuencial, fácil de leer y de depurar.

El problema es el costo: hasta Java 21, todo hilo de Java envolvía un hilo del sistema operativo, y esos son caros. El JEP 444 lo explica con la ley de Little: atender más peticiones por segundo exige más peticiones en curso a la vez y, con un hilo por cada una, más hilos, que se agotan mucho antes que la CPU o la red. Un pool no lo arregla, porque no aumenta cuántos hay. La salida era el estilo asíncrono (**`CompletableFuture`**, frameworks reactivos), que escala a costa de partir la lógica en callbacks.

ANTES · JAVA 17

Antes.java

```
static int cobrarPedidos(List<Integer>
ids) throws Exception {
```

DESPUÉS · JAVA 21

Despues.java

```
static int cobrarPedidos(List<Integer>
ids) throws Exception {
```

```
    ExecutorService ejecutor =
Executors.newFixedThreadPool(10);
    try {
        List<Future<Integer>> respuestas
= new ArrayList<>();
        for (int id : ids) {

respuestas.add(ejecutor.submit(() ->
consultarPago(id)));
        }
        int total = 0;
        for (Future<Integer> respuesta :
respuestas) {
            total += respuesta.get();
        }
        return total;
    } finally {
        ejecutor.shutdown();
    }
}
```

```
    try (ExecutorService ejecutor =
Executors.newVirtualThreadPerTaskExecutor
()) {
        List<Future<Integer>> respuestas
= new ArrayList<>();
        for (int id : ids) {

respuestas.add(ejecutor.submit(() ->
consultarPago(id)));
        }
        int total = 0;
        for (Future<Integer> respuesta :
respuestas) {
            total += respuesta.get();
        }
        return total;
    }
}
```

SALIDA

```
Pedidos consultados: 1000
Importe cobrado: 124500
```

Solo cambia cómo se crea y se cierra el ejecutor: desde Java 19, **`ExecutorService`** se cierra con **`try`**, y el cierre espera a que terminen sus tareas. No imprimo tiempos, pero la cuenta es simple: con 10 hilos, las 1000 esperas de 20 ms van de 10 en 10, así que el antes no baja de 2 segundos. Con hilos virtuales pueden ocurrir todas a la vez.

Qué es. Un **hilo virtual** es un **`Thread`** que gestiona la JVM, no el sistema operativo. Corre montado sobre un hilo de plataforma (el de siempre), que hace de **portador**. Cuando se bloquea en E/S, la JVM lo desmonta, guarda su pila en memoria y libera el portador; cuando llega la respuesta, lo vuelve a montar, quizá en otro. Por defecto hay tantos portadores como núcleos.

Cuatro pedidos que esperan la red: hilos de plataforma frente a hilos virtuales





Primero, un pool fijo de dos hilos de plataforma. Cada uno envuelve un hilo del sistema operativo. Hay cuatro pedidos y cada uno tiene que consultar el pago por red.

Los pedidos 1 y 2 toman los dos hilos y lanzan su consulta. Mientras esperan la respuesta siguen ocupando su hilo: el hilo del sistema queda bloqueado, sin hacer nada.

Los pedidos 3 y 4 no pueden empezar: no queda ningún hilo libre, aunque la CPU está desocupada. Un hilo de plataforma no sabe esperar sin ocupar su hilo, así que el carril de esperar sin hilo queda vacío.

Llegan las respuestas de 1 y 2. Terminan y devuelven sus hilos al pool.

Recién ahora 3 y 4 toman los hilos, y esperan igual que los anteriores. Con dos hilos, solo dos consultas pueden estar en curso a la vez.

Terminan 3 y 4. Hicieron falta dos rondas de espera, una detrás de la otra.

Ahora con hilos virtuales: un hilo virtual por pedido. Los dos hilos de plataforma pasan a ser portadores; en la realidad hay, por defecto, tantos como núcleos.

El pedido 1 se monta en el portador 1 y el 2 en el portador 2. Ejecutan su código hasta llamar a la red.

Al bloquearse en la E/S, cada hilo virtual se desmonta: la JVM guarda su pila en memoria y el portador queda libre.

Los portadores libres montan los pedidos 3 y 4, que también llegan hasta su consulta.

También se desmontan. Las cuatro consultas están en curso a la vez y ningún hilo del sistema está bloqueado.

Llega la respuesta del pedido 2. El planificador lo vuelve a montar en un portador libre, que no tiene por qué ser el mismo de antes, y termina su trabajo.

Llegan las demás respuestas: cada pedido se monta, termina y libera el portador. Una sola ronda de espera para los cuatro. Cada consulta tardó lo mismo; lo que sube es cuántos pedidos se atienden por segundo.

Por qué es mejor. El código sigue siendo secuencial, pero el límite ya no lo pone cuántos hilos del sistema aguanta la máquina. Como son baratos, no se reutilizan: el JEP 444 pide crear uno por tarea, nunca un pool.

Cuándo no usarlos.

- Cuando la tarea calcula (ordenar, comprimir, cifrar) en lugar de esperar. No son hilos más rápidos: aumentan el *throughput* (tareas terminadas por segundo) cuando las tareas esperan, y con trabajo de CPU, más hilos que núcleos no ayuda.
- Cuando el recurso del otro lado es limitado. Una base de datos con 20 conexiones no gana conexiones porque tengas 1000 hilos: el límite se pone explícito con un `Semaphore`, como recomienda el JEP (lo practicas en el ejercicio).

> – **Para profundizar**

> en Java 21, un hilo virtual dentro de un bloque o método **`synchronized`** no se puede desmontar: queda **anclado** (*pinned*) a su portador, y si espera E/S ahí, bloquea también el hilo del sistema. No rompe el programa, pero le quita escala. El JEP 444 sugiere **`ReentrantLock`** para los **`synchronized`** frecuentes que protegen E/S larga, y **`-Djdk.tracePinnedThreads=full`** para encontrarlos. Java 24 eliminó ese anclaje (JEP 491); lo explico en el capítulo de Java 25.

Ejercicio. La base de datos acepta 20 conexiones. Sin volver a un pool, haz que nunca haya más de 20 consultas a la vez. Pista: **`java.util.concurrent`** tiene algo que reparte permisos.

<details>
<summary>Solución</summary>

un **`Semaphore`** con 20 permisos: cada hilo pide uno antes de consultar y lo devuelve en el **`finally`**.

Ejercicio.java

```
static final Semaphore conexiones = new Semaphore(20);

static int consultarConLimite(int idPedido) throws InterruptedException {
    conexiones.acquire();
    try {
        return consultarBase(idPedido);
    } finally {
        conexiones.release();
    }
}
```

Imprime **`Importe cobrado: 124500`** y **`Nunca más de 20 consultas a la vez: true`**. Los hilos que no consiguen permiso esperan en **`acquire()`**, y esa espera también libera al portador.
</details>

Pattern matching en switch: preguntar por la forma con el compilador vigilando

Pattern matching en switch · lenguaje · definitiva en Java 21 · primera LTS: Java 21 · JEP 441 · preview en Java 17, 18, 19 y 20

Vuelves a los medios de pago sellados del capítulo de Java 17: tarjeta, transferencia y efectivo al recibir el pedido. Quieres el texto que ve el cliente.

Por qué nace. En Java 17 viste las sealed classes, que cierran la lista de subtipos de un tipo, y el pattern matching de **`instanceof`**, que prueba el tipo y declara la variable a la vez. Faltaba comparar un valor contra varias alternativas. **`switch`** solo admitía enteros (salvo **`long`**), sus envoltorios, enums y **`String`**, y solo comparaba contra constantes (JEP 441), así que quedaba la cadena de **`if/else`**.

El JEP le ve dos problemas a esa cadena: nada obliga a que cada rama haga lo que debe, y el compilador no puede comprobar que cubriste todos los casos. Por eso el antes, que es el mismo código con el que cerraba el capítulo de Java 17, termina con un **`throw`** que nunca se alcanza.

ANTES · JAVA 17

Antes.java

```
static String describir(MedioDePago pago)
{
    if (pago instanceof Tarjeta tarjeta)
{
        return "Tarjeta terminada en " +
tarjeta.ultimos4();
    }
    if (pago instanceof Transferencia
transferencia) {
        return "Transferencia con
referencia " +
transferencia.referencia();
    }
    if (pago instanceof Efectivo) {
        return "Efectivo al recibir el
pedido";
    }
    throw new
IllegalArgumentException("Medio de pago
desconocido: " + pago);
}
```

DESPUÉS · JAVA 21

Despues.java

```
static String describir(MedioDePago pago)
{
    return switch (pago) {
        case Tarjeta tarjeta -> "Tarjeta
terminada en " + tarjeta.ultimos4();
        case Transferencia transferencia
-> "Transferencia con referencia " +
transferencia.referencia();
        case Efectivo efectivo ->
"Efectivo al recibir el pedido";
    };
}
```

SALIDA

```
Tarjeta terminada en 4242
Transferencia con referencia TRF-881
Efectivo al recibir el pedido
```

Qué es. Un **`switch`** cuyos **`case`** llevan un **patrón de tipo** (**`case Tarjeta tarjeta`**) en lugar de una constante. Si el valor es de ese tipo, **`tarjeta`** queda declarada y ya convertida, solo dentro de esa rama. Como las switch expressions de Java 17, devuelve un valor.

La clave es la **exhaustividad**: un switch con patrones tiene que cubrir todos los valores posibles. Como **`MedioDePago`** es sealed y sus tres subtipos tienen su **`case`**, el compilador sabe que no falta ninguno y no hace falta **`default`**. Hay dos herramientas más: **`case null`** atiende el null (sin él, el switch lanza **`NullPointerException`**, como siempre), y la **guarda `when`** agrega una condición a un patrón; la verás en la sección siguiente.

Por qué es mejor. La diferencia aparece el día que alguien suma un cuarto medio de pago, **`Cupon`**, a los **`permits`**. Lo comprobé en el JDK 21: la cadena de **`if`** sigue compilando y falla en ejecución con **`IllegalArgumentException: Medio de pago desconocido`**, quizá en producción. El switch, en cambio, deja de compilar: **`javac`** responde **`the switch expression does not cover all possible input values`** y señala justo el switch que hay que actualizar.

Cuándo no usarlo.

- Con un **`default`** puesto por costumbre. En una jerarquía sealed, el **`default`** tapa la exhaustividad y el compilador ya no te avisa del medio de pago nuevo. El JEP 441 lo dice: un switch exhaustivo sin **`default`** es mejor que uno con **`default`**, cuando se puede.
- Cuando los tipos crecen seguido y las operaciones son pocas. Mi criterio: ahí, un método en la interfaz que cada tipo implementa envejece mejor que el mismo switch repetido en diez lugares. El switch brilla con tipos fijos y muchas operaciones.

> – Para profundizar
>
> el orden de los **`case`** importa. Un **`case`** que nunca puede ganar porque otro anterior lo cubre (por ejemplo, **`case Tarjeta t`** antes de un **`case Tarjeta t when ...`**) es un error de compilación: el JEP lo llama *dominancia*. Si llega en ejecución un subtipo que el switch no conocía al compilarse, el **`default`** sintético que insertó el compilador lanza **`MatchException`**. Y **`default`** no atrapa el null: para eso se escribe **`case null, default`**. La variable sin usar de **`case Efectivo efectivo`** puede escribirse _ desde Java 22 (Java 25).

Record patterns: desarmar los datos en el mismo case

Record patterns · lenguaje · definitiva en Java 21 · primera LTS: Java 21 · JEP 440 · preview en Java 19 y 20

El costo de envío depende de cómo se entrega el pedido: el retiro en tienda es gratis; a domicilio en Costa Rica cuesta 5, o nada si el pedido llega a 100; fuera del país, 20.

Por qué nace. Un record (capítulo de Java 17) es un portador transparente de datos: sus componentes se leen con accesores como **`direccion()`** o **`pais()`**. El pattern matching de **`instanceof`** ya ahorraba el cast, pero la variable del patrón servía casi solo para llamar a esos accesores, uno por uno. El JEP 440 propone que el patrón, además de probar el tipo, extraiga los componentes directamente. Y cuando los records se anidan (un envío con una dirección que tiene un país), la navegación se alarga: un **`instanceof`**, una variable, una cadena de accesores y otro **`if`**.

ANTES · JAVA 17

Antes.java

```
static int costoDeEnvio(Pedido pedido) {
```

DESPUÉS · JAVA 21

Despues.java

```
static int costoDeEnvio(Pedido pedido) {
```

```
    Envio envio = pedido.envio();
    if (envio instanceof Retiro) {
        return 0;
    } else if (envio instanceof Domicilio
d) {
        String pais =
d.direccion().pais();
        if (pais.equals("Costa Rica")) {
            return pedido.importe() >=
100 ? 0 : 5;
        }
        return 20;
    }
    throw new
IllegalStateException("Envío desconocido:
" + envio);
}
```

```
    return switch (pedido.envio()) {
        case Retiro r -> 0;
        case Domicilio(Direccion(String
ciudad, String pais)) when
pais.equals("Costa Rica") ->
                pedido.importe() >= 100 ?
0 : 5;
        case Domicilio d -> 20;
    };
}
```

SALIDA

```
Envío de Ana: 0
Envío de Luis: 0
Envío de Marta: 5
Envío de John: 20
```

Qué es. Un **record pattern** tiene la forma del constructor del record: **`Domicilio(Direccion(String ciudad, String pais))`**. Se lee al revés que un **`new`**: en lugar de armar un **`Domicilio`** con una **`Direccion`** adentro, lo desarma. Si el valor es un **`Domicilio`**, Java llama a los accesores por ti y declara **`ciudad`** y **`pais`**. Los patrones se anidan, así que en un solo **`case`** bajas dos niveles. En lugar del tipo puedes escribir **`var`**, y la misma forma funciona en un **`instanceof`**.

El después usa además una guarda: **`when pais.equals("Costa Rica")`** solo deja entrar a ese **`case`** si la condición se cumple. Los envíos a domicilio que no la cumplen siguen bajando hasta **`case Domicilio d`**. Por eso el orden de los **`case`** importa.

Por qué es mejor. El código que desarma se parece al que arma. Las tres reglas de negocio quedan en tres **`case`**, uno por regla, y el switch sigue siendo exhaustivo porque **`Envio`** es sealed: si mañana se agrega otra forma de entrega, el compilador te avisa. La cadena **`d.direccion().pais()`** desaparece; **`pais`** nace dentro del patrón, justo donde se usa.

Cuándo no usarlos.

- Con records de muchos componentes, cuando solo te interesa uno. En Java 21 hay que nombrarlos todos: aquí **`ciudad`** se declara y no se usa. Java 22 permite escribir **`_`** (capítulo de Java 25); hasta entonces, un accesor puede leerse mejor.
- Fuera de los records: un record pattern solo desarma records, no clases comunes.

- Cuando el orden de los componentes cambia a menudo. El patrón es posicional: si alguien invierte dos componentes del mismo tipo, el patrón sigue compilando y asigna los valores cruzados. Lo comprobé: con **`Direccion(String pais, String ciudad)`**, el patrón de arriba dejó "Costa Rica" en **`ciudad`**.

> – Para profundizar
>
> un record pattern nunca coincide con null (JEP 440). Por eso los dos códigos difieren en un caso borde: con un **`Domicilio`** cuya dirección es null, el antes lanza **`NullPointerException`** en **`d.direccion().pais()`**, y el después no entra al primer **`case`** de domicilio y cae en **`case Domicilio d`**, que cobra 20. Lo comprobé en el JDK 21. Y si un accesor lanza una excepción durante el desarme, el switch termina con **`MatchException`**.

Colecciones secuenciadas: primero, último y al revés, igual en todas

Colecciones secuenciadas (sequenced collections) · API · definitiva en Java 21 · primera LTS: Java 21 · JEP 431

La tienda guarda los pedidos en el orden en que llegan, en una lista, y los clientes en el orden de su primera compra, sin repetir, en un **`LinkedHashSet`**. Quieres el primero, el último y la lista con los más recientes primero.

Por qué nace. Muchas colecciones tienen un **orden de encuentro**: un primer elemento, un segundo, y así hasta el último. Lo tienen una lista, una cola doble (**`Deque`**) y un **`LinkedHashSet`**. Pero hasta Java 21 no había un tipo común que lo dijera, y cada una pedía el primero y el último a su manera. El JEP 431 lo muestra con una tabla: en una **`List`**, el último es **`list.get(list.size() - 1)`**; en un **`Deque`**, **`getLast()`**; en un **`SortedSet`**, **`last()`**; y en un **`LinkedHashSet`** no había método: la única forma de obtener el último era recorrer el conjunto entero. Recorrer al revés también cambiaba en cada caso, y un **`LinkedHashSet`** no lo permitía sin copiarlo a otra colección.

ANTES · JAVA 17

Antes.java

```
String primero = pedidos.get(0);
String ultimo =
pedidos.get(pedidos.size() - 1);
List<String> recientesPrimero = new
ArrayList<>(pedidos);
Collections.reverse(recientesPrimero);

String primerCliente =
clientes.iterator().next();
String ultimoCliente = null;
for (String cliente : clientes) {
```

DESPUÉS · JAVA 21

Despues.java

```
String primero = pedidos.getFirst();
String ultimo = pedidos.getLast();
List<String> recientesPrimero =
pedidos.reversed();

String primerCliente =
clientes.getFirst();
String ultimoCliente =
clientes.getLast();
```

```
    ultimoCliente = cliente;
}
```

SALIDA

```
Primer pedido: P-101
Último pedido: P-104
Más recientes primero: [P-104, P-103, P-102, P-101]
Primer cliente: Ana
Último cliente: Marta
```

Qué es. Java 21 agrega tres interfaces a las colecciones: **`SequencedCollection`**, **`SequencedSet`** y **`SequencedMap`**. **`List`** y **`Deque`** ahora heredan de **`SequencedCollection`**; **`LinkedHashSet`** y **`SortedSet`** son **`SequencedSet`**; **`LinkedHashMap`** y **`SortedMap`** son **`SequencedMap`**. Todas comparten los mismos métodos: **`getFirst`**, **`getLast`**, **`addFirst`**, **`addLast`**, **`removeFirst`**, **`removeLast`** y **`reversed`**, que devuelve la colección en orden inverso.

Por qué es mejor. El código dice lo que quieres: **`getLast()`** en lugar de una cuenta con **`size() - 1`**, y una línea en lugar de un bucle que recorre el conjunto para quedarse con el último. Los mismos nombres sirven en una lista y en un conjunto, y un método puede pedir un parámetro **`SequencedCollection`**: "cualquier colección con orden". Antes eso no se podía expresar, porque **`Collection`** es demasiado general y **`List`** deja fuera a los conjuntos ordenados (JEP 431).

Cuándo no usarlas. Más que casos para evitarlas, tienen tres límites:

- El tipo declarado manda. Con **`Set<String> clientes`**, **`getFirst()`** no compila aunque el objeto sea un **`LinkedHashSet`** (lo comprobé: **`cannot find symbol`**). Por eso el después declara **`SequencedSet<String>`**. Un **`HashSet`** no tiene orden, así que no es secuenciado.
- **`reversed()`** devuelve una **vista**, no una copia. En el antes, **`recientesPrimero`** es una lista independiente; en el después, si la colección admite cambios, lo que modifiques en la vista se escribe en la original. Lo comprobé: un **`addFirst`** sobre la vista agregó el elemento al final de la lista original. Si necesitas una copia, envuélvela en un **`new ArrayList<>(...)`**.
- En una colección vacía, **`getFirst()`** lanza **`NoSuchElementException`**; no devuelve null.

> – Para profundizar
>
> agregar métodos default tan arriba en la jerarquía tiene un costo de compatibilidad. Las notas de versión de Oracle de JDK 21 advierten de posibles incompatibilidades de código fuente y binarias: una colección propia que ya tenía un **`getFirst()`** o un **`reversed()`** puede chocar con los nuevos. Si tu proyecto tiene colecciones propias, revísalas al migrar. Para mapas, **`SequencedMap`** trae **`firstEntry()`**, **`lastEntry()`**, **`pollFirstEntry()`** y **`putFirst()`**.

También en Java 21. Dos cambios de Java 18 llegan a una LTS por primera vez. **UTF-8 por defecto** (JEP 400): las APIs que leen o escriben texto sin recibir un charset usaban el del sistema y el idioma de la máquina, así que un archivo escrito con `FileWriter` en macOS podía leerse corrupto en Windows. Desde Java 18 usan UTF-8 en todas partes, salvo la consola; antes de migrar, el JEP recomienda probar con `-Dfile.encoding=UTF-8`. Y el **servidor web simple** (JEP 408): `jwebserver` sirve los archivos estáticos de la carpeta actual en `127.0.0.1`, puerto 8000, solo con GET y HEAD y sin HTTPS. Es para prototipos y pruebas, no para producción.

> **Cuidado:** si encuentras un tutorial con `STR."Hola \{nombre}"`, no es Java estándar. Los String Templates fueron preview en Java 21 (JEP 430) y en Java 22 (JEP 459), y se retiraron en Java 23. Según las notas de versión de Oracle de JDK 23, concluyeron que la función no servía en su forma actual y, sin consenso sobre un diseño mejor, la retiraron "por ahora". Hasta Java 27, ninguna versión la volvió a incluir.

> **Lo que debes llevarte:** los hilos virtuales hacen barato el estilo "un hilo por petición": un hilo que espera E/S ya no bloquea un hilo del sistema, así que aumentan cuántas tareas terminas por segundo cuando las tareas esperan, no la velocidad de cada una. El pattern matching de `switch` y los record patterns convierten los records y las sealed classes de Java 17 en datos que puedes preguntar por su forma y desarmar, con el compilador avisándote si falta un caso. Y las colecciones secuenciadas dan a todas las colecciones con orden los mismos `getFirst`, `getLast` y `reversed`.

Java 25: menos ruido, más intención

Java 25 salió en septiembre de 2025, y es la LTS que recoge cuatro versiones de trabajo: todo lo que quedó definitivo entre Java 22 y Java 25. No trae un cambio de paradigma como Java 8 ni una pieza tan grande como los hilos virtuales de Java 21. Lo que trae es más fino: quita ruido. Variables que había que nombrar sin usarlas, bucles que había que escribir porque a un Stream le faltaba una operación, una validación escondida en un método auxiliar del constructor, la clase y el `static` que un programa de diez líneas no necesita.

Casi todas estas novedades pasaron por varias previews antes de quedar definitivas, y algunas cambiaron de nombre o de API en el camino. En cada una te muestro su forma final, que es la que vas a escribir. Todos los "antes" de este capítulo compilan y corren en Java 21, la LTS anterior.

Variables sin nombre: decir en el código lo que no vas a usar

Variables y patrones sin nombre · lenguaje · preview en Java 21 · definitiva en Java 22 · primera LTS: Java 25 · JEP 456

Imagina que lees el importe de un pedido desde un texto y, si el texto no es un número, el importe vale 0. O que de un pedido solo te interesa el cliente.

Por qué nace. Hay variables que existen solo porque el lenguaje exige un nombre. El parámetro de un **`catch`** es el caso típico: si lo único que haces al capturar la excepción es devolver 0, la **`e`** sobra, pero sin ella el código no compila. Con los *record patterns* de Java 21 (un patrón que desarma un record en sus componentes) pasa lo mismo: para sacar el cliente de un **`Pedido`** tienes que nombrar también el importe y el estado, aunque no los toques.

El JEP 456 señala el riesgo que eso deja: si la intención de no usar una variable no queda escrita, alguien que mantenga el código después puede usarla por accidente. Y quien lee tiene que revisar todo el cuerpo para saber que **`importe`** y **`estado`** no se usan.

El mismo código en Java 21 y en Java 25:

ANTES · JAVA 21

Antes.java

```
static int leerImporte(String texto) {
    try {
        return Integer.parseInt(texto);
    } catch (NumberFormatException e) {
// e no se usa
        return 0;
    }
}

static String clienteDe(Object dato) {
    if (dato instanceof Pedido(String
cliente, int importe, String estado)) {
        return cliente; // importe y
estado no se usan
    }
    return "sin cliente";
}
```

DESPUÉS · JAVA 25

Despues.java

```
static int leerImporte(String texto) {
    try {
        return Integer.parseInt(texto);
    } catch (NumberFormatException _) {
        return 0;
    }
}

static String clienteDe(Object dato) {
    if (dato instanceof Pedido(String
cliente, _, _)) {
        return cliente;
    }
    return "sin cliente";
}
```

SALIDA

```
Importe de "120": 120
Importe de "ciento veinte": 0
Cliente: ana
Cliente: sin cliente
```

Qué es. El guion bajo **_** declara una variable o un patrón **sin nombre**: algo que tiene que estar ahí, pero que no vas a usar. Como no pone ningún nombre a tu alcance, después no se puede leer ni escribir; si lo intentas, el código no compila. Sirve en un **`catch`**, en los parámetros de una lambda, en una variable local,

en un **`for`**, en un try-with-resources y dentro de un patrón. En **`Pedido(String cliente, _, _)`**, cada **`_`** ocupa el lugar de un componente sin declarar su tipo ni su nombre.

Por qué es mejor. La intención queda escrita y la comprueba el compilador. Al leer **`Pedido(String cliente, _, _)`** sabes de inmediato que solo importa el cliente, sin buscar en el cuerpo. Y quien necesite el importe tendrá que nombrarlo a propósito, no usar por descuido una variable que estaba ahí por obligación.

Cuándo no usarlo. **`_`** deja a la vista que algo se ignora, pero no vuelve buena la decisión de ignorarlo. Un **`catch (NumberFormatException _)`** que devuelve 0 está bien cuando 0 es una respuesta válida; si estás tragándote un error que alguien debería ver, el problema no es el nombre de la variable sino que nadie registra el fallo. Y si un nombre ayuda a entender qué hay en esa posición, aunque no lo uses, consérvalo.

> – **Para profundizar**
>
> Java 8 ya avisaba al usar **`_`** como nombre de variable, y desde Java 9 es un error (JEP 213): por eso pudo reaprovecharse sin romper código de Java 9 en adelante. Como un **`_`** no declara nada, puedes usar varios en el mismo ámbito, como en el ejemplo. Un **`_`** suelto no puede ser el patrón de primer nivel de un **`instanceof`** ni de un **`case`**, pero un patrón de tipo sin nombre sí: **`case Pedido _ ->`** es válido. La preview de Java 21 (JEP 443) quedó definitiva en Java 22 sin cambios.

Stream Gatherers: las operaciones que le faltaban a un Stream

Stream Gatherers · API · preview en Java 22 y 23 · definitiva en Java 24 · primera LTS: Java 25 · JEP 485

Imagina que los pedidos pagados se despachan en lotes de tres: el almacén recibe tres ids juntos, después los tres siguientes, y el último lote lleva los que sobren.

Por qué nace. En el capítulo de Java 8 viste que un Stream tiene operaciones intermedias (**`filter`**, **`map`**, **`sorted`**…) y una terminal. El problema es que las intermedias son un conjunto **fijo**: puedes combinar las que hay, pero no agregar una propia. La única pieza extensible era la terminal, **`collect`**. El JEP 485 lo dice así: hay tareas que no caben en un pipeline porque falta la operación intermedia que necesitan, y agrupar de a tres es uno de sus ejemplos.

En la práctica, eso te obligaba a salir del Stream: terminarlo en una lista y seguir con un bucle sobre índices. Funciona, pero parte la lectura en dos, crea una lista intermedia y te deja a mano la aritmética del borde (el **`Math.min`** del último lote).

ANTES · JAVA 21

Antes.java

```
static List<List<String>>
lotesDePagados(List<Pedido> pedidos) {
```

DESPUÉS · JAVA 25

Despues.java

```
static List<List<String>>
lotesDePagados(List<Pedido> pedidos) {
```

```
    List<String> pagados =
pedidos.stream()
            .filter(pedido ->
pedido.estado().equals("PAGADO"))
            .map(Pedido::id)
            .toList(); // el Stream
termina aquí

    List<List<String>> lotes = new
ArrayList<>();
    for (int i = 0; i < pagados.size(); i
+= 3) {
        lotes.add(pagados.subList(i,
Math.min(i + 3, pagados.size())));
    }
    return lotes;
}
```

```
    return pedidos.stream()
            .filter(pedido ->
pedido.estado().equals("PAGADO"))
            .map(Pedido::id)

.gather(Gatherers.windowFixed(3))
            .toList();
}
```

SALIDA

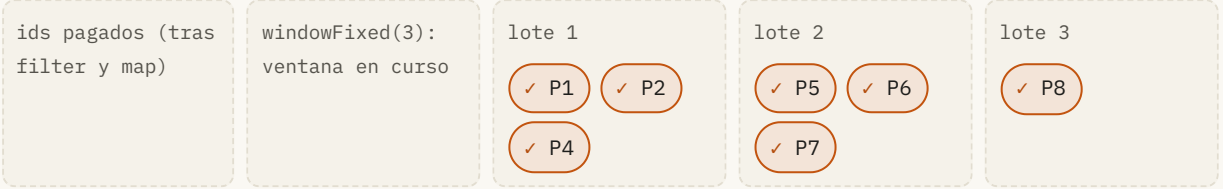
```
Lotes de 3: [[P1, P2, P4], [P5, P6, P7], [P8]]
Sin pedidos: []
```

P3 está pendiente: **`filter`** lo deja fuera antes de agrupar.

Qué es. Un **gatherer** es una operación intermedia que puedes definir tú, y **`gather(...)`** es el método del Stream que la aplica. La clase **`Gatherers`** trae cinco hechas, entre ellas **`windowFixed`** (grupos de tamaño fijo) y **`windowSliding`** (grupos que se solapan). **`windowFixed(3)`** recibe los elementos de a uno y los entrega agrupados en listas de tres. Si al final sobran menos, el último grupo es más corto, y un Stream vacío no produce ningún grupo.

Para eso el gatherer tiene que **recordar** lo que ya vio, y esa memoria no se ve en el código. Recórrelo paso a paso:

Cómo arma los lotes windowFixed(3)



filter y map ya hicieron su parte: P3 estaba pendiente y no llegó. Quedan siete ids, que entran al gatherer de a uno.

Entra P1. windowFixed no lo deja seguir: lo guarda en su ventana, porque un lote necesita tres.

Entra P2 y espera junto a P1. Todavía no ha salido nada hacia toList.

Entra P4 y la ventana se llena: el gatherer entrega [P1, P2, P4] como un solo elemento, una lista, y empieza una ventana vacía.

Entran P5 y P6, y esperan en la ventana nueva.

Con P7 se completa el segundo lote: [P5, P6, P7].

Entra P8, el último. Queda solo en la ventana: no hay más elementos que la completen.

La fuente se terminó. Solo entonces el gatherer entrega lo que quedó en la ventana como un último lote, más corto: [P8]. Resultado: [[P1, P2, P4], [P5, P6, P7], [P8]].

Por qué es mejor. La cadena vuelve a leerse como una frase: de los pedidos, los pagados, sus ids, en lotes de tres. Desaparecen la lista intermedia y los índices, y con ellos el error típico del último lote: un **`subList`** que se pasa del tamaño. Y como sigue siendo un Stream, el agrupado convive con las operaciones que cortan el recorrido: con un **`limit(1)`** detrás, el Stream deja de pedir elementos en cuanto tiene el primer lote, aunque la fuente sea infinita (lo probé con **`Stream.iterate`**).

Cuándo no usarlo. Si lo que necesitas ya existe como operación o como colector (**`distinct`**, **`groupingBy`**), úsalo: es lo que el siguiente lector espera encontrar. Un gatherer propio exige entender sus cuatro piezas (las del recuadro de abajo) y solo compensa si lo vas a reutilizar. Y si los datos ya están en una lista sin más cadena alrededor, el bucle sigue siendo honesto.

> – **Para profundizar**
>
> cada ventana que entrega **`windowFixed`** es una lista **inmodificable**: un **`add`** lanza **`UnsupportedOperationException`** (lo dice su javadoc). En el "antes" también lo son, por otra razón: **`subList`** es una vista sobre la lista que devuelve **`toList()`**, que ya era inmodificable. Un gatherer se arma con hasta cuatro funciones: un inicializador (el estado privado), un integrador (recibe cada elemento y decide qué empujar), un combinador opcional (para evaluarlo en paralelo) y un finalizador, que corre cuando no quedan elementos y es lo que le permite a **`windowFixed`** entregar el último lote incompleto. Tras las previews de Java 22 (JEP 461) y 23 (JEP 473), quedó definitivo en Java 24 sin cambios.

Scoped values: un dato de contexto que no se queda pegado al hilo

Scoped values · API · incubadora en Java 20 · preview de Java 21 a 24 · definitiva en Java 25 · primera LTS: Java 25 · JEP 506

Imagina que cada petición a la tienda la hace un cliente, y que la auditoría, dos llamadas más abajo, tiene que anotar quién fue.

Por qué nace. Lo natural es pasar el cliente como parámetro. Pero si en medio hay código que no es tuyo (un framework) o diez métodos que no lo necesitan, agregarlo a todos es impracticable. La salida de

siempre fue **`ThreadLocal`**: una variable con una copia por hilo, que se guarda al empezar la petición y se lee donde haga falta.

El JEP 506 le encuentra tres defectos de diseño. Es **mutable sin restricciones**: cualquier código que la alcance puede llamar a **`set`**, así que cuesta saber quién la cambió. Su **vida no tiene límite**: el valor queda en el hilo hasta que alguien llama a **`remove`**, y si se olvida, en un pool de hilos puede filtrarse a la petición siguiente. Y su **herencia es cara**: un hilo hijo que hereda los valores del padre tiene que reservar espacio para cada uno. Con los hilos virtuales, que multiplican la cantidad de hilos, el JEP dice que estos problemas pesan más.

ANTES · JAVA 21

Antes.java

```
static final ThreadLocal<String> CLIENTE
= new ThreadLocal<>();

static void atender(String cliente, int
importe) {
    CLIENTE.set(cliente);
    try {
        registrarPedido(importe);
    } finally {
        CLIENTE.remove(); // si se
olvida, el valor queda pegado al hilo
    }
}

static void registrarPedido(int importe)
{
    auditar("pedido registrado por " +
importe);
}

static void auditar(String mensaje) {
    System.out.println("[" +
CLIENTE.get() + "] " + mensaje);
}
```

DESPUÉS · JAVA 25

Despues.java

```
static final ScopedValue<String> CLIENTE
= ScopedValue.newInstance();

static void atender(String cliente, int
importe) {
    ScopedValue.where(CLIENTE, cliente)
            .run(() ->
registrarPedido(importe));
}

static void registrarPedido(int importe)
{
    auditar("pedido registrado por " +
importe);
}

static void auditar(String mensaje) {
    System.out.println("[" +
CLIENTE.get() + "] " + mensaje);
}
```

SALIDA

```
[ana] pedido registrado por 120
[luis] pedido registrado por 180
¿Queda un cliente fuera de la petición? false
```

Qué es. Un **scoped value** es un valor que se **liga** a una clave mientras se ejecuta un bloque de código, y solo durante ese bloque. **`ScopedValue.where(CLIENTE, cliente)`** prepara la ligadura y **`.run(...)`** ejecuta

la lambda con ella activa: cualquier método que se llame desde ahí, a cualquier profundidad, lee el valor con `CLIENTE.get()`. Cuando `run` termina, la ligadura desaparece, también si terminó con una excepción. Para devolver un resultado, `.call(...)` hace lo mismo y retorna el de la lambda.

Por qué es mejor. Desaparecen el `try`/`finally` y el `remove`, y con ellos el olvido que filtra datos entre peticiones: la vida del valor la marca la forma del código. No hay `set`: dentro del bloque el valor no se puede reasignar. Si una parte necesita otro valor, un `where` anidado lo cambia solo ahí dentro; al salir, vuelve el de afuera. Y fuera de cualquier ligadura, `get()` no devuelve `null` en silencio: lanza `NoSuchElementException`, y `isBound()` te deja preguntarlo antes.

Cuándo no usarlo. Si el dato cabe en un parámetro, pásalo así: es más explícito y fácil de probar. Tampoco sirve cuando un método de más abajo tiene que **cambiar** el valor para que lo vea quien lo llamó; ahí `ThreadLocal` sigue siendo la herramienta, y el JEP aclara que no busca retirarlo. Y un hilo que creas tú dentro del bloque no ve la ligadura (lo comprobé con un hilo virtual): heredarla es tarea de la concurrencia estructurada, que en Java 25 sigue en preview.

> – Para profundizar
>
> la API cambió entre previews. Hasta Java 23 había atajos como `ScopedValue.runWhere(...)` y `callWhere(...)`; el JEP 487 los quitó en Java 24 y dejó un solo camino, `where(...).run(...)` o `where(...).call(...)`, que es el que compila en Java 25 (lo comprobé: `runWhere` ya no existe). La versión definitiva solo cambió un detalle respecto de la última preview: `orElse` ya no acepta `null`.

Archivos compactos: un programa chico sin ceremonia

Archivos fuente compactos y métodos main de instancia · lenguaje · preview de Java 21 a 24, con otros nombres · definitiva en Java 25 · primera LTS: Java 25 · JEP 512

Imagina que quieres un programa de diez líneas que sume los importes del día y lo imprima. Nada más.

Por qué nace. Para eso, Java pedía una clase pública, un método `public static void main(String[] args)`, un `import` y `System.out.println`. Cada palabra tiene sentido en un sistema grande (`public` controla quién ve qué, `static` separa la clase de sus objetos, `String[] args` recibe los argumentos), pero en diez líneas no aporta nada, y a quien empieza le exige aceptar varios conceptos antes de imprimir una línea.

El JEP 512 lo plantea así: Java brilla en la programación "en grande", pero también quiere ser un buen primer lenguaje, con una rampa de entrada que crezca sin saltos hasta el lenguaje completo.

ANTES · JAVA 21

Antes.java

```
import java.util.List;
```

DESPUÉS · JAVA 25

Despues.java

```
void main() {
```

```java
public class Antes {
    public static void main(String[] args) {
        List<Integer> importes = List.of(40, 120, 90, 180);
        int total = 0;
        for (int importe : importes) {
            total += importe;
        }
        System.out.println("Pedidos: " + importes.size() + ", total: " + total);
    }
}
```

```java
    List<Integer> importes = List.of(40, 120, 90, 180);
    int total = 0;
    for (int importe : importes) {
        total += importe;
    }
    IO.println("Pedidos: " + importes.size() + ", total: " + total);
}
```

SALIDA

```
Pedidos: 4, total: 430
```

Fíjate en que el "después" no tiene clase, ni **`static`**, ni **`import`**.

Qué es. Un **archivo fuente compacto** es un **`.java`** con métodos y campos sueltos, sin una clase que los envuelva: el compilador crea por ti una clase implícita que los contiene. Su **`main`** puede ser un **método de instancia**, sin **`static`**, sin **`public`** y, si no usas los argumentos, sin **`String[] args`**. **`IO.println`** es la forma corta de imprimir en consola: **`IO`** es una clase nueva de **`java.lang`**, que se importa siempre.

El **`import`** de **`List`** falta por otra novedad de Java 25, las **importaciones de módulo** (JEP 511, preview en Java 23 y 24). **`import module java.base;`** importa de golpe las clases públicas de todos los paquetes que exporta ese módulo (**`List`**, **`Map`**, **`Stream`**, **`Path`**...). Un archivo compacto la trae implícita; en una clase normal la escribes tú:

Modulos.java

```java
import module java.base;

public class Modulos {
    public static void main(String[] args) {
        List<Integer> importes = List.of(40, 120, 90, 180);
        Map<Boolean, Long> porTamano = importes.stream()
                .collect(Collectors.partitioningBy(importe -> importe >= 100,
                        Collectors.counting()));
        IO.println("De 100 o más: " + porTamano.get(true) + ", menores: " + porTamano.get(false));
    }
}
```

Imprime **`De 100 o más: 2, menores: 2`**.

Por qué es mejor. Lo que queda en el archivo es el programa. Quien empieza aprende primero variables, bucles y métodos, y deja clases y paquetes para después. Y crecer es corto: cuando el programa merece una clase, lo envuelves en ella.

Cuándo no usarlo. En código de producción con paquetes y varias clases. La clase implícita no tiene paquete y su nombre no se debe usar desde otro código: es un punto de entrada, no una pieza de un sistema. Con **`import module`**, cuidado con las ambigüedades: si importas dos módulos con una clase del mismo nombre (**`Date`** está en **`java.util`** y en **`java.sql`**), usarla no compila hasta que importes esa clase concreta, que manda sobre los de módulo.

> – **Para profundizar**
>
> el nombre cambió en cada preview: *unnamed classes* en Java 21 (JEP 445), *implicitly declared classes* en 22 y 23, *simple source files* en 24 y, en la forma final, *compact source files*. En la última preview **`IO`** vivía en **`java.io`** y sus métodos se importaban solos (bastaba **`println(...)`**); en Java 25 pasó a **`java.lang`**, ya no se importan y se escribe **`IO.println(...)`**. El nombre de la clase implícita lo decide la implementación; con el javac del JDK es el del archivo, y por eso este ejemplo se ejecuta con **`java Despues`**. Y **`java Despues.java`** lo compila en memoria y lo ejecuta sin pasar por **`javac`** (lo probé en Java 25).

Constructores flexibles: validar antes de super(...)

Cuerpos de constructor flexibles · lenguaje · preview de Java 22 a 24 · definitiva en Java 25 · primera LTS: Java 25 · JEP 513

Imagina un **`PedidoUrgente`** que hereda de **`Pedido`** y que tiene que rechazar un importe de 0 o negativo antes de construirse.

Por qué nace. Cuando una clase hereda de otra, su constructor llama al del padre con **`super(...)`**. En Java 21, esa llamada tenía que ser **la primera sentencia** del constructor: el lenguaje lo exigía para garantizar que la parte heredada del objeto estuviera lista antes de que el resto del código la tocara. La regla tiene sentido, pero también prohibía cosas que no tocan el objeto en absoluto, como comprobar un argumento.

Validar *después* de **`super(...)`** funciona, pero tarde: el constructor del padre ya corrió con un valor inválido. El JEP 513 describe la salida que quedaba: llamar a un método estático auxiliar dentro de los propios argumentos de **`super(...)`**. Es lo que ves en el "antes": la validación queda lejos del constructor, metida en una expresión.

ANTES · JAVA 21

Antes.java

DESPUÉS · JAVA 25

Despues.java

```
static class PedidoUrgente extends Pedido
{

    PedidoUrgente(int importe) {
        super(validar(importe));
    }

    private static int validar(int
importe) {
        if (importe <= 0) {
            throw new
IllegalArgumentException("el importe debe
ser positivo: " + importe);
        }
        return importe;
    }
}
```

```
static class PedidoUrgente extends Pedido
{

    PedidoUrgente(int importe) {
        if (importe <= 0) {
            throw new
IllegalArgumentException("el importe debe
ser positivo: " + importe);
        }
        super(importe);
    }
}
```

SALIDA

```
Pedido urgente de 120
Rechazado: el importe debe ser positivo: 0
```

Qué es. Desde Java 25, un constructor puede tener sentencias **antes** de **`super(...)`** o de **`this(...)`**. Esa parte se llama **prólogo**. En el prólogo el objeto todavía no está construido, así que no puedes usar **`this`**: ni llamar a sus métodos ni leer sus campos, y el compilador lo rechaza si lo intentas. Lo que sí puedes es trabajar con los argumentos: validarlos, lanzar una excepción, calcular los valores que le pasas al padre.

Por qué es mejor. El código queda en el orden en que lo piensas: primero compruebo, después construyo. La validación vive donde se lee, sin un método auxiliar que solo existe para esquivar una regla, y el padre ni siquiera empieza a construirse si el argumento no sirve.

Cuándo no usarlo. El prólogo no es lugar para lógica pesada: si preparar los argumentos crece, un método con buen nombre es más claro. Si la validación vale para todas las subclases, su lugar natural sigue siendo el constructor del padre. Y los records y los enums mantienen sus reglas: el constructor canónico de un record no puede llamar ni a **`super(...)`** ni a **`this(...)`**, y el de un enum no puede llamar a **`super(...)`**.

> – Para profundizar
>
> el prólogo tiene una excepción a la regla de no usar **`this`**: puede **asignar** campos de la propia clase que no tengan inicializador. Eso resuelve un caso clásico: el constructor del padre llama a un método que la subclase sobrescribe, y ese método lee un campo de la subclase. Con **`this.cliente =`**

`cliente` después de **`super()`**, como obligaba Java 21, el campo todavía no tiene valor cuando el padre lo pide, y se imprime **`pedido urgente de null`** (lo comprobé en Java 21). Asignado antes de **`super()`**, ya está:

Prologo.java

```
static class PedidoUrgente extends Pedido {
    private final String cliente;

    PedidoUrgente(String cliente) {
        this.cliente = cliente; // se asigna antes de super(...)
        super();
    }

    @Override
    String describir() {
        return "pedido urgente de " + cliente;
    }
}
```

Imprime **`Creando: pedido urgente de ana`**. La primera preview (JEP 447, Java 22) se llamaba *Statements before super(...)*; las de Java 23 y 24 ya usaban el nombre actual.

También en Java 25. Hay novedades que quedaron definitivas en este tramo y que no vas a ver en el código de todos los días, pero conviene ubicar. La API de funciones y memoria foráneas (FFM, en **`java.lang.foreign`**, definitiva en Java 22, JEP 454) permite llamar a bibliotecas nativas y manejar memoria fuera del heap sin la fragilidad de JNI. La API de archivos de clase (**`java.lang.classfile`**, definitiva en Java 24, JEP 484) es una forma estándar de leer, generar y transformar archivos **`.class`**, para herramientas que hasta ahora dependían de bibliotecas externas como ASM. Desde Java 23, los comentarios de documentación se pueden escribir en Markdown, con líneas que empiezan por **`///`** (JEP 467). Y una que Java 25 no estrena, sino que hereda de Java 24: un hilo virtual que se bloquea dentro de un bloque **`synchronized`** ya no queda anclado (*pinned*) a su hilo portador (JEP 491). Es la limitación de la que habla la nota sobre hilos virtuales del capítulo de Java 21: la recomendación de cambiar **`synchronized`** por **`ReentrantLock`** para esquivarla ya no hace falta. Siguen existiendo casos de anclaje, como el código nativo que vuelve a llamar a Java y se bloquea, pero son la excepción.

Nota: varias funciones vienen dentro de Java 25 pero siguen en preview (o en incubadora, o experimentales): hay que activarlas a propósito y pueden cambiar antes de quedar definitivas, así que no les asigno LTS. Son la concurrencia estructurada (**`StructuredTaskScope`**, quinta preview, JEP 505), los tipos primitivos en patrones, **`instanceof`** y **`switch`** (tercera preview, JEP 507), Stable Values (preview, JEP 502; desde Java 26 se llama Lazy Constants), las codificaciones PEM de claves y certificados (preview, JEP 470), la Vector API (décima incubadora, JEP 508) y el perfilado de tiempo de CPU en JFR (experimental, JEP 509).

> **Lo que debes llevarte:** Java 25 no cambia la forma de pensar en Java: le quita ruido. `_` dice qué no vas a usar, los gatherers le dan al Stream las operaciones intermedias que le faltaban, los scoped values pasan un dato de contexto que no se puede reasignar y desaparece solo, los archivos compactos permiten un programa chico sin clase ni `static`, y un constructor puede validar antes de `super(...)`. Todas pasaron por previews y algunas cambiaron de nombre o de API en el camino: lo que compila en Java 25 es la forma final, y es la que aprendiste aquí.

Apéndice: todas las novedades en una tabla

Para repasar de un vistazo. La columna **Aparece** dice en qué versión se vio por primera vez y en qué estado; **Definitiva** es desde cuándo se usa sin activar nada, y **Capítulo** es la primera LTS que la trae así.

Novedad	Aparece	Definitiva	Capítulo
Streams	Java 8 · definitiva	Java 8	Java 8
Lambdas e interfaces funcionales	Java 8 · definitiva	Java 8	Java 8
Optional	Java 8 · definitiva	Java 8	Java 8
java.time	Java 8 · definitiva	Java 8	Java 8
Métodos default en interfaces	Java 8 · definitiva	Java 8	Java 8
CompletableFuture	Java 8 · definitiva	Java 8	Java 8
Métodos fábrica de colecciones (List.of, Set.of, Map.of)	Java 9 · definitiva	Java 9	Java 11
Inferencia de tipos de variables locales (var)	Java 10 · definitiva	Java 10	Java 11
Cliente HTTP (java.net.http)	Java 9 · incubadora	Java 11	Java 11
Métodos nuevos de String (isBlank, lines, strip, repeat)	Java 11 · definitiva	Java 11	Java 11
Ejecutar un programa de un solo archivo fuente (java Hola.java)	Java 11 · definitiva	Java 11	Java 11
Switch expressions	Java 12 · preview	Java 14	Java 17
NullPointerException explicativas	Java 14 · definitiva	Java 14	Java 17
Text blocks	Java 13 · preview	Java 15	Java 17
Records	Java 14 · preview	Java 16	Java 17
Pattern matching de instanceof	Java 14 · preview	Java 16	Java 17
Stream.toList()	Java 16 · definitiva	Java 16	Java 17
Sealed classes	Java 15 · preview	Java 17	Java 17

Novedad	Aparece	Definitiva	Capítulo
Encapsulación fuerte de los internos del JDK	Java 9 · definitiva	Java 17	Java 17
UTF-8 por defecto	Java 18 · definitiva	Java 18	Java 21
Servidor web simple (jwebserver)	Java 18 · definitiva	Java 18	Java 21
Hilos virtuales	Java 19 · preview	Java 21	Java 21
Pattern matching en switch	Java 17 · preview	Java 21	Java 21
Record patterns	Java 19 · preview	Java 21	Java 21
Colecciones secuenciadas	Java 21 · definitiva	Java 21	Java 21
String Templates (preview retirado)	Java 21 · preview	— (retirada en Java 23)	Java 21
Variables y patrones sin nombre (_)	Java 21 · preview	Java 22	Java 25
API de funciones y memoria foráneas (FFM)	Java 14 · incubadora	Java 22	Java 25
Comentarios de documentación en Markdown	Java 23 · definitiva	Java 23	Java 25
Stream Gatherers	Java 22 · preview	Java 24	Java 25
synchronized sin anclar hilos virtuales	Java 24 · definitiva	Java 24	Java 25
API de archivos de clase	Java 22 · preview	Java 24	Java 25
Scoped values	Java 20 · incubadora	Java 25	Java 25
Archivos fuente compactos y métodos main de instancia	Java 21 · preview	Java 25	Java 25
Importación de módulos (import module)	Java 23 · preview	Java 25	Java 25
Cuerpos de constructor flexibles	Java 22 · preview	Java 25	Java 25

Para cerrar

Si volvemos a la reunión del principio, la pregunta "¿qué cambia en el código que escribo todos los días?" tiene ahora una respuesta por versión:

1. **Java 8** cambió la forma de escribir: comportamiento como valor (lambdas), colecciones que se describen en vez de recorrerse (Streams) y fechas que no cambian a tus espaldas.
2. **Java 11** quitó ceremonia: `var`, colecciones inmodificables en una línea, un cliente HTTP en el JDK y programas que se ejecutan sin compilar a mano.

3. **Java 17** hizo de los datos ciudadanos de primera: records, text blocks, switch que devuelve un valor y jerarquías cerradas que el compilador conoce.
4. **Java 21** abarató la concurrencia con hilos virtuales y dejó desarmar los datos en un **`switch`** que el compilador vigila.
5. **Java 25** pulió lo que quedaba: `_` para lo que no usas, Gatherers para las operaciones que le faltaban a un Stream, scoped values y programas chicos sin clase.

Ninguna de estas novedades obliga a reescribir código que funciona. Lo que sí cambia es lo que puedes pedirle al lenguaje la próxima vez que abras un archivo, y lo que vas a encontrar cuando leas el código de otros.

¿Y en tu equipo? ¿En qué LTS están hoy, y cuál de estas novedades fue la que más les cambió (o les cambiaría) el código del día a día? Me interesa leerte en los comentarios.

Referencias

- Oracle, Java SE Support Roadmap.
- OpenJDK, JEP 126: Lambda Expressions & Virtual Extension Methods.
- OpenJDK, JEP 107: Bulk Data Operations for Collections.
- Oracle, Java SE 8 API: Collectors.
- OpenJDK, JEP 150: Date & Time API.
- Oracle, The Java Tutorials: Lambda Expressions.
- Oracle, Java Language Specification, Java SE 8: chapter 15 (15.8.3, 15.27).
- Oracle, Java Language Specification, Java SE 8: 8.4.8 Inheritance, Overriding, and Hiding.
- Oracle, Java SE 8 API: FunctionalInterface.
- Oracle, Java SE 8 API: java.util.function.
- Oracle, Java SE 8 API: Comparator.
- Oracle, Java SE 8 API: Optional.
- Oracle, Java SE 9 API: Optional.
- Oracle, Java SE 8 API: Date.
- Oracle, Java SE 8 API: Calendar.
- Oracle, Java SE 8 API: SimpleDateFormat.
- Oracle, Java SE 8 API: java.time.
- Oracle, Java SE 8 API: Period.
- Oracle, Java SE 8 API: Collection.
- Oracle, Java SE 8 API: Iterable.

- Oracle, Java SE 8 API: CompletableFuture.
- OpenJDK, JEP 322: Time-Based Release Versioning.
- OpenJDK, JEP 286: Local-Variable Type Inference.
- OpenJDK, JEP 323: Local-Variable Syntax for Lambda Parameters.
- OpenJDK, JEP 269: Convenience Factory Methods for Collections.
- Oracle, Java SE 11 API: List, Set, Map y Collection.
- Oracle, Java SE 8 API: Collections.
- OpenJDK, JEP 110: HTTP/2 Client (Incubator).
- OpenJDK, JEP 321: HTTP Client API.
- Oracle, Java SE 11 API: HttpClient.
- Oracle, Java SE 11 API: String, Character y Pattern.
- OpenJDK, JEP 330: Launch Single-File Source-Code Programs.
- OpenJDK, JEP 458: Launch Multi-File Source-Code Programs.
- OpenJDK, JEP 200: The Modular JDK.
- OpenJDK, JEP 261: Module System.
- OpenJDK, JEP 222: jshell: The Java Shell (Read-Eval-Print Loop).
- OpenJDK, JEP 320: Remove the Java EE and CORBA Modules.
- OpenJDK, JEP 325: Switch Expressions (Preview).
- OpenJDK, JEP 354: Switch Expressions (Second Preview).
- OpenJDK, JEP 361: Switch Expressions.
- OpenJDK, JEP 355: Text Blocks (Preview).
- OpenJDK, JEP 368: Text Blocks (Second Preview).
- OpenJDK, JEP 378: Text Blocks.
- OpenJDK, JEP 359: Records (Preview).
- OpenJDK, JEP 384: Records (Second Preview).
- OpenJDK, JEP 395: Records.
- OpenJDK, JEP 305: Pattern Matching for instanceof (Preview).
- OpenJDK, JEP 375: Pattern Matching for instanceof (Second Preview).
- OpenJDK, JEP 394: Pattern Matching for instanceof.
- OpenJDK, JEP 360: Sealed Classes (Preview).
- OpenJDK, JEP 397: Sealed Classes (Second Preview).
- OpenJDK, JEP 409: Sealed Classes.
- OpenJDK, JEP 406: Pattern Matching for switch (Preview).
- OpenJDK, JEP 358: Helpful NullPointerExceptions.

- OpenJDK, JEP 260: Encapsulate Most Internal APIs.
- OpenJDK, JEP 396: Strongly Encapsulate JDK Internals by Default.
- OpenJDK, JEP 403: Strongly Encapsulate JDK Internals.
- Oracle, Java SE 17 API: String.
- Oracle, Java SE 17 API: Record.
- Oracle, Java SE 17 API: List.
- Oracle, Java SE 16 API: Stream.
- Oracle, JDK 15 Release Notes.
- Oracle, JDK 17 Release Notes.
- OpenJDK, JEP 444: Virtual Threads.
- OpenJDK, JEP 425: Virtual Threads (Preview).
- OpenJDK, JEP 436: Virtual Threads (Second Preview).
- OpenJDK, JEP 491: Synchronize Virtual Threads without Pinning.
- Oracle, Java SE 21 API: ExecutorService.
- Oracle, Java SE 21 API: Executors.
- OpenJDK, JEP 441: Pattern Matching for switch.
- OpenJDK, JEP 420: Pattern Matching for switch (Second Preview).
- OpenJDK, JEP 427: Pattern Matching for switch (Third Preview).
- OpenJDK, JEP 433: Pattern Matching for switch (Fourth Preview).
- OpenJDK, JEP 440: Record Patterns.
- OpenJDK, JEP 405: Record Patterns (Preview).
- OpenJDK, JEP 432: Record Patterns (Second Preview).
- OpenJDK, JEP 431: Sequenced Collections.
- Oracle, Java SE 21 API: SequencedCollection.
- Oracle, JDK 21 Release Notes.
- OpenJDK, JEP 400: UTF-8 by Default.
- OpenJDK, JEP 408: Simple Web Server.
- OpenJDK, JEP 430: String Templates (Preview).
- OpenJDK, JEP 459: String Templates (Second Preview).
- OpenJDK, JEP 465: String Templates (Third Preview).
- Oracle, JDK 23 Release Notes.
- OpenJDK, JEP 456: Unnamed Variables & Patterns.
- OpenJDK, JEP 443: Unnamed Patterns and Variables (Preview).
- OpenJDK, JEP 485: Stream Gatherers.

- Oracle, Java SE 25 API: Gatherers.
- OpenJDK, JEP 506: Scoped Values.
- OpenJDK, JEP 487: Scoped Values (Fourth Preview).
- Oracle, Java SE 25 API: ScopedValue.
- OpenJDK, JEP 512: Compact Source Files and Instance Main Methods.
- OpenJDK, JEP 511: Module Import Declarations.
- OpenJDK, JEP 513: Flexible Constructor Bodies.
- OpenJDK, JEP 454: Foreign Function & Memory API.
- OpenJDK, JEP 484: Class-File API.
- OpenJDK, JEP 467: Markdown Documentation Comments.
- Oracle, JDK 25 Release Notes.