Java moderno

El ecosistema Java ha evolucionado significativamente en los últimos años incorporando características modernas que facilitan el desarrollo de aplicaciones empresariales.

por Ivan Ruiz Rube

Agenda de este tema

Fundamentos y Evolución

Ediciones y evolución histórica de Java.

Características Modernas

Lambdas, Streams, Optionals y Records para código conciso.

Mejoras Recientes

Text Blocks, Pattern Matching, Switch expressions y Virtual Threads.

Herramientas Interactivas

JShell: REPL para experimentación rápida con Java.

Ediciones de Java: SE, Jakarta EE y ME

Java SE (Standard Edition)

La base fundamental del ecosistema Java.

- Incluye las bibliotecas core
- Ideal para aplicaciones de escritorio
- Componentes esenciales: colecciones, concurrencia, I/O
- Cimiento para otras ediciones de Java

Jakarta EE (Enterprise Edition)

Enfocada en aplicaciones empresariales robustas.

- Antes conocida como Java EE
- Gestión de transacciones, seguridad, web, etc.
- Especificaciones: Servlet, CDI, JPA, JAX-RS
- Servidores: WildFly, Payara, OpenLiberty

Java ME (Micro Edition)

Optimizada para dispositivos con recursos limitados.

- Subconjunto de Java SE
- Perfiles para distintos dispositivos
- Ideal para IoT y sistemas embebidos
- Eficiente en entornos de baja memoria/CPU

Evolución de Java

Oracle adoptó un nuevo modelo de lanzamiento en 2018, acelerando la evolución de Java.

Cada 6 Meses

Nuevas versiones de Java se publican en marzo y septiembre.

LTS vs No-LTS

Versiones LTS reciben soporte extendido. Las intermedias tienen vida corta.

Características Incrementales

Cada versión introduce mejoras pequeñas sin esperar grandes actualizaciones.

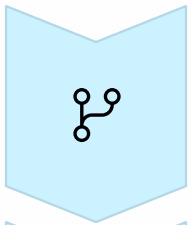
Adopción Flexible

Las organizaciones pueden actualizar rápidamente o esperar a versiones LTS.

Versiones principales de Java

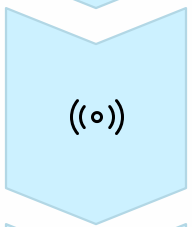
Versión	Año
JDK 1.0	1996
J2SE 1.2	1998
J2SE 5.0	2004
Java SE 7	2011
Java SE 8	2014
Java SE 9	2017
Java SE 11	2018
Java SE 17	2021
Java SE 21	2023
Java SE 25	2025*

Características Modernas de Java



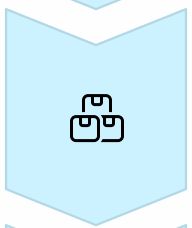
Expresiones lambda

Habilitan la programación funcional con sintaxis concisa y elegante



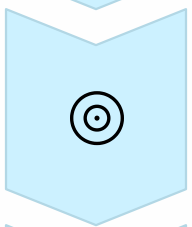
Streams

Facilitan el procesamiento de colecciones de forma declarativa y paralelizable



Optionals

Evitan NullPointerException y mejoran la legibilidad del código

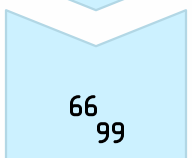


Records

Reducen el código repetitivo en clases de datos con implementación automática

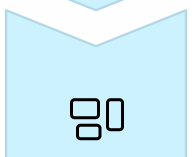
Pattern matching

Simplifica el código con verificación y conversión de tipos en una sola operación



Text blocks

Permiten definir cadenas multilínea con mejor legibilidad y sin secuencias de escape



Switch expressions

Transforma switch en expresiones que devuelven valores con sintaxis más compacta

Virtual threads

Implementan concurrencia ligera y eficiente para aplicaciones de alto rendimiento

JShell

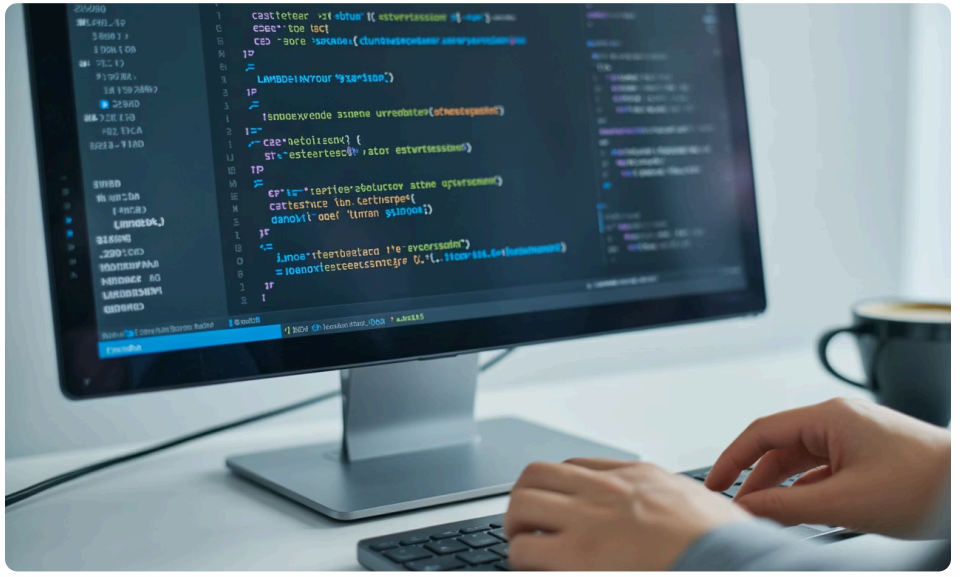
REPL interactivo que permite explorar y probar código Java sin necesidad de crear clases completas

Lambdas

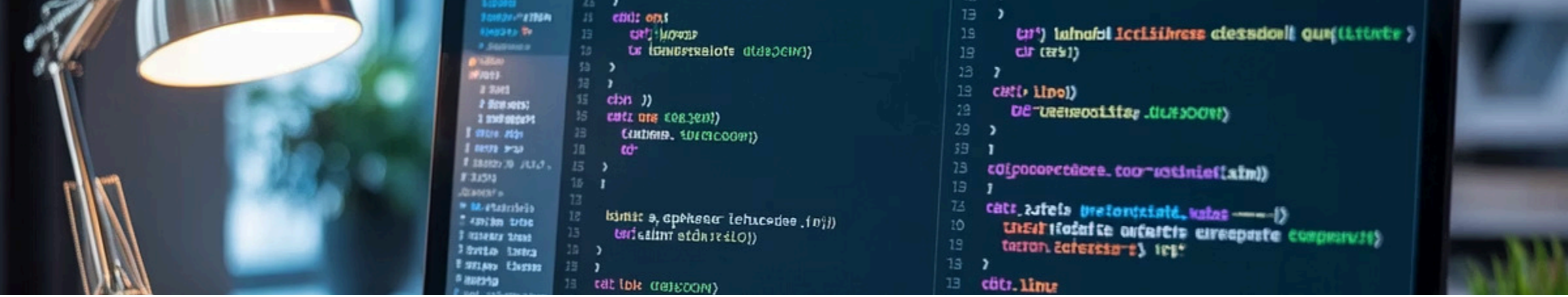
Introducción en Java 8

Las *expresiones lambda* permiten definir **funciones anónimas** o bloques de código concisos que se pueden tratar como datos (pasar como parámetro, asignar a una variable, etc.)

Representan un cambio significativo en el paradigma de programación de Java, acercándolo a la programación funcional



Las lambdas simplifican enormemente el código, especialmente cuando se trabaja con colecciones y operaciones funcionales



Lambdas: sintaxis

Reemplazan la necesidad de clases anónimas para implementar **interfaces funcionales** (interfaces con un solo método abstracto). Las lambdas reducen significativamente la verbosidad del código. El compilador infiere los tipos y el método a implementar.

Sintaxis General

```
(parámetros) -> { sentencias; }
```

Interfaces funcionales

Las expresiones lambda se aplican principalmente mediante interfaces funcionales predefinidas en Java.

Interfaz	Firma del método	Caso de uso principal
Function<T,R>	R apply(T t)	Transformar un valor.
Predicate<T>	boolean test(T t)	Filtrar o validar.
Consumer<T>	void accept(T t)	Realizar una acción con un valor.
Supplier<T>	T get()	Proveer o generar valores.
Comparator<T>	int compare(T o1, T o2)	Definir criterios de ordenación.
Runnable	void run()	Tareas sin retorno.

Estas interfaces estándar cubren la mayoría de patrones funcionales comunes en el desarrollo.

Interfaz Function

Uso

Transforma un valor de un tipo determinado en otro.

Definición

```
@FunctionalInterface
public interface Function<T, R> {
    R apply(T t);
}
```

Aplicaciones

- Convertir precios añadiendo IVA
- Formatear fechas
- Extraer propiedades de objetos

Implementación mediante clase anónima

```
Function longitud =
  new Function<String,Integer>() {
    public Integer apply(String s) {
        return s.length();
    }
  };
```

Implementación con Lambda

```
Function<String,Integer> longitud = s ->   s.length();
```

Invocando a la función

```
Integer resultado = longitud.apply("UCA");
// resultado=3
```

Interfaz Predicate

Uso

Evalúa una condición sobre un objeto y devuelve un valor booleano.

Definición

```
@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t);
}
```

Aplicaciones

- Filtrar colecciones
- Validar condiciones sobre objetos.

Implementación mediante clase anónima

```
Predicate esPar = new Predicate<>() {
    @Override
    public boolean test(Integer n) {
        return n % 2 == 0;
    }
};
```

Implementación con Lambda

```
Predicate esPar = n -> n % 2 == 0;
```

Invocando a la función

```
boolean resultado = esPar.test(4); // true
```

Interfaz Consumer

Uso

Procesar o consumir cada elemento de una colección, sin devolver nada.

Definición

```
@FunctionalInterface
public interface Consumer<T> {
    void accept(T t);
}
```

Aplicaciones

- Mostrar información en la interfaz de usuario
- Ejecutar acciones en cada elemento de una colección

Implementación mediante clase anónima

```
Consumer<String> imprimir = new Consumer<>() {
    @Override
    public void accept(String s) {
        System.out.println(s);
    }
};
```

Implementación con Lambda

```
Consumer<String> imprimir = ->
System.out.println(s);
```

Invocando a la función

```
imprimir.accept("Hola"); // Imprime: Hola
```

Interfaz Supplier

Uso

Representa un proveedor de resultados, no toma argumentos y devuelve un valor de tipo

Método principal: **T get()**

Ideal para operaciones del tipo:

- Retrasar la creación de objetos hasta que realmente se necesiten.
- Producir valores aleatorios o dinámicos cuando se solicitan.
- Proporciona conexiones a bases de datos o servicios externos.
- Genera identificadores únicos o credenciales temporales.

Implementación mediante clase anónima

```
Supplier<Double> generarAleatorio = new Supplier<>
() {
    @Override
    public Double get() {
        return Math.random();
    }
};
```

Implementación con Lambda

```
Supplier<Double> generarAleatorio =
    () -> Math.random();
```

Invocando a la función

```
Double valor = generarAleatorio.get();
// valor= 0.34
```

Interfaz Comparator

Uso

Compara dos objetos del mismo tipo para establecer un orden.

Definición

```
@FunctionalInterface
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```

Aplicaciones

- Ordenar listas de objetos

Implementación mediante clase anónima

```
Comparator<Persona> comparadorEdad =
  new Comparator<Persona>() {
   public int compare(Persona p1, Persona p2) {
     return p1.getEdad().compareTo(p2.getEdad());
   }
  };
```

Implementación con Lambda

```
Comparator<Persona> comparadorEdad =
 (p1, p2) -> p1.getEdad().compareTo(p2.getEdad());
```

Invocando a la función

```
Persona pA=new Persona("Antonio",30);
Persona pB=new Persona("Juan",10);

int resultado = comparadorEdad.compare(pA, pB);
    // resultado < 0

lista.sort(comparadorEdad);
    //[pB, pA]
```

Interfaz Runnable

Uso

Ejecutar una tarea que no recibe argumentos ni devuelve resultado, generalmente en un hilo separado.

Definición

```
@FunctionalInterface
public interface Runnable {
    void run();
}
```

Aplicaciones

- Ejecutar tareas en un hilo nuevo (Thread)
- Programación de temporizadores y cronómetros
- Actualizaciones periódicas en interfaces gráficas

Implementación mediante clase anónima

```
Runnable tarea = new Runnable() {
    @Override
    public void run() {
        System.out.println("Tarea en un hilo");
    }
};
```

Implementación con Lambda

```
Runnable tarea = () -> {
    System.out.println("Tarea en un hilo");
};
```

Invocando a la función

```
// Para iniciar el hilo
Thread hilo = new Thread(tarea);
hilo.start();
```

Lambdas: sintaxis flexible

Inferencia de tipos en parámetros

No es obligatorio especificar el tipo de los parámetros cuando el compilador puede deducirlo del contexto (target typing).

```
Function<String, Boolean> esVacia = s -> s.isEmpty();
```

Omisión de paréntesis en un único parámetro

Si solo hay un parámetro, puedes omitir los paréntesis alrededor de él

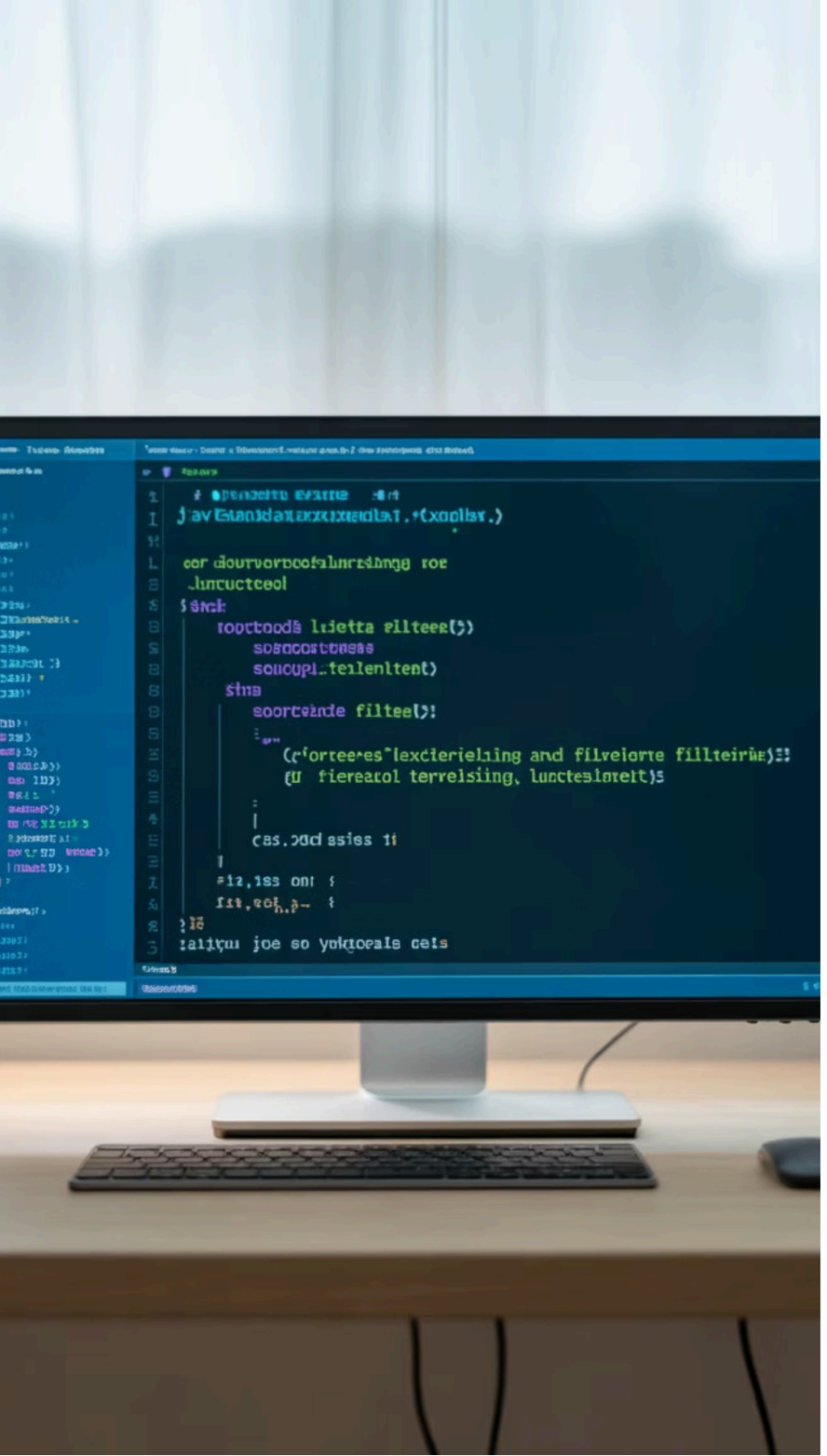
```
// Sin paréntesis
Consumer imprimir = mensaje -> System.out.println(mensaje);
```

Expresión única
Cuando el cuerpo consta de una única expresión (o llamada a método), puedes omitir las llaves `{}` y la palabra reservada `return` (si devuelve algo).

```
// Expresión única sin llaves ni return
Predicate esDeCadiCadi = p -> p.getLugarNacimiento().equals("Cádiz");
```

Bloque de código
Si quieres ejecutar varias líneas o necesitas más instrucciones (por ejemplo, validaciones, varias sentencias), debes usar llaves y, en su caso, retornar explícitamente

```
    int porcentajeIVA = 21;

    // Dentro de las funciones lambda se pueden leer variables del ámbito externo
    // pero siempre que sean finales o no se modifiquen tras su inicialización.
    // No se pueden modificar esas variables en la función lambda

    Function<Double, Double> calcularConIVA = precio -> {
        if (precio == null || precio < 0) {
            return 0.0;
        }
        // Aplica el IVA: precio * (1 + porcentajeIVA/100.0)
        return precio * (1 + porcentajeIVA / 100.0);
    };
```



Lambdas: usos habituales

Ordenación

list.sort((a,b) ->
a.compareTo(b))

Filtrado

stream.filter(x -> x > 10)

Transformación

stream.map(s ->
s.toUpperCase())

Iterar colecciones

list.forEach(nombre ->
System.out.println("Hola "
+ nombre))

Eventos

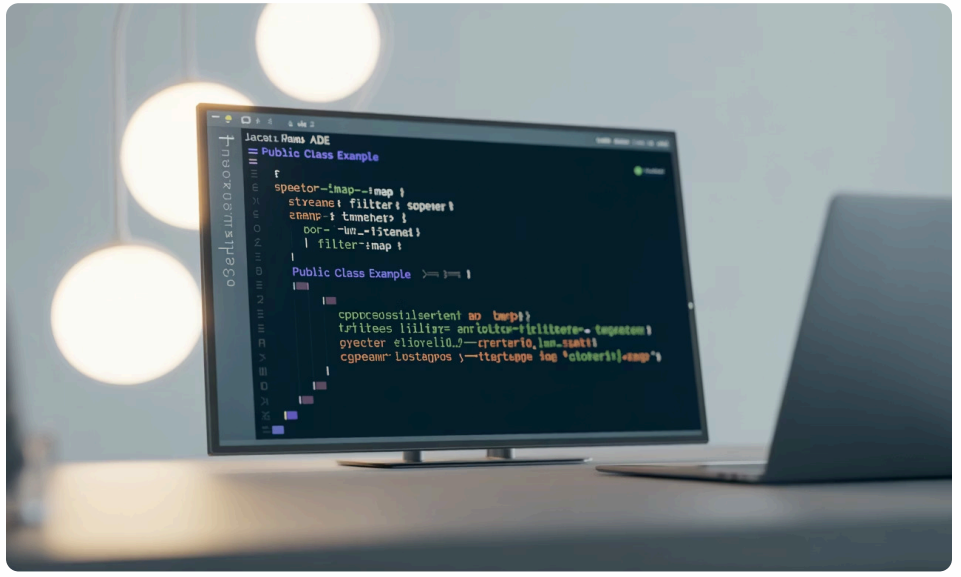
button.addActionListener(e ->
handleClick())

Streams

Introducción en Java 8

La API de *Streams* permite procesar conjuntos de datos de forma declarativa (estilo funcional), con operaciones como filter, map, reduce, etc.

Representa un cambio de paradigma hacia la programación funcional en Java



Los streams permiten expresar operaciones complejas de procesamiento de datos de manera concisa y legible

Streams: pipelines

Fuente

Colección, array u otra fuente de datos

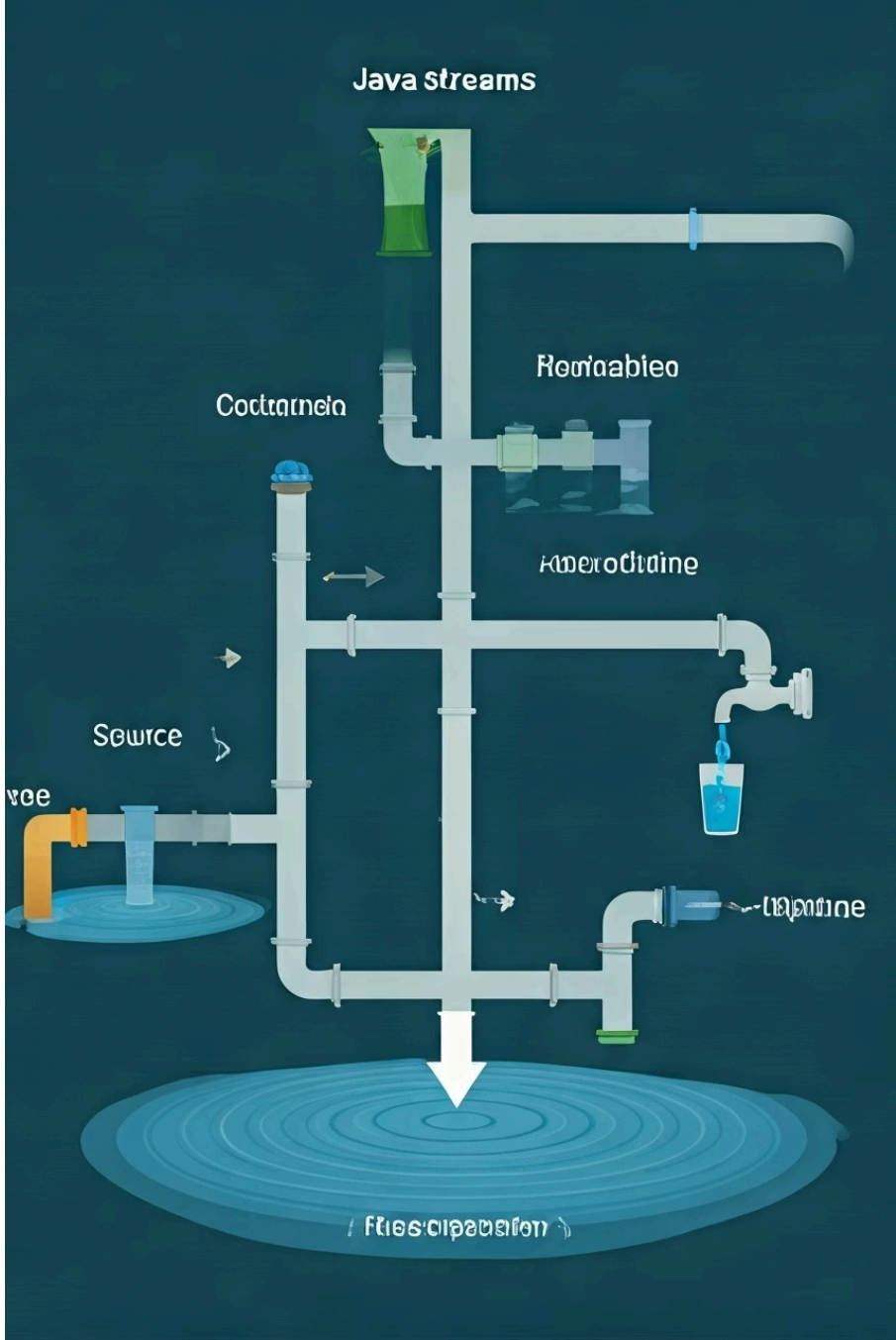
Operaciones intermedias

Transforman datos (filter, map, sorted, etc.)

Operación terminal

Produce resultado (forEach, reduce, max, collect, findFirst, anyMatch)

Los *streams* encadenan operaciones intermedias (que transforman datos) y una operación terminal (que produce un resultado o efecto colateral). No modifican la colección original, sino que generan resultados nuevos.



Streams: lazy evaluation

Operaciones Intermedias

Las transformaciones se registran pero no ejecutan inmediatamente. Solo se almacenan como pasos pendientes.

Operación Terminal

La ejecución real comienza cuando se invoca una operación terminal como collect() o forEach().

Optimizaciones Automáticas

El sistema procesa solo los elementos necesarios. Puede detener evaluaciones si se cumple una condición de terminación.

Ventajas

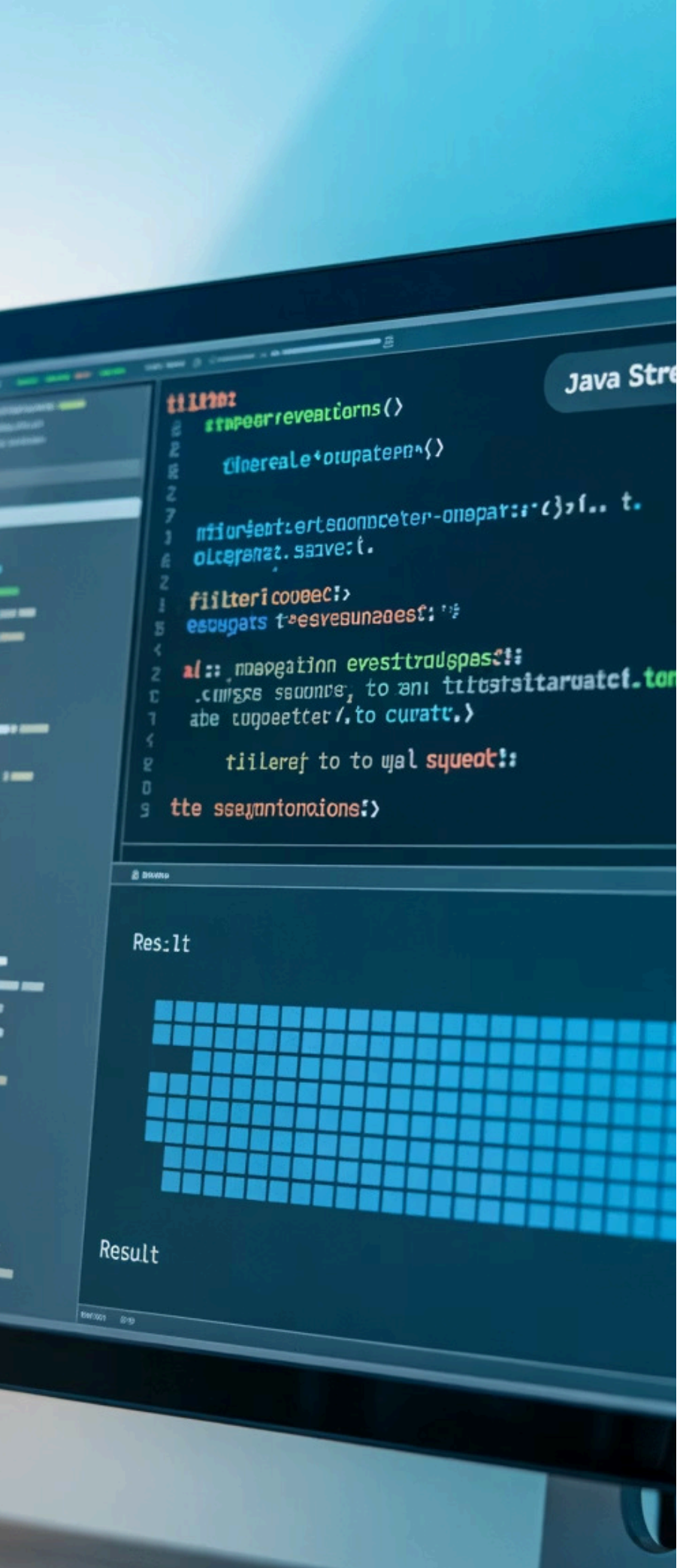
Mejora el rendimiento evitando trabajo innecesario. Reduce el consumo de recursos con procesamiento eficiente.

Esta estrategia de procesamiento bajo demanda permite a los streams de Java implementar operaciones eficientes sin evaluar elementos que no contribuyen al resultado final.

Streams: ejemplo

Con programación tradicional

```
// Recorrer una lista de números, seleccionar los números pares, elevarlos al cuadrado e imprimir solo los dos primeros resultados.

List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
int contadorResultados = 0;
int limite = 2;

// Recorremos la lista
for (Integer n : numeros) {
    // 1) Filtrar: solo quedarnos con números pares
    if (n % 2 != 0) {
        continue;
    }

    // 2) Mapear: elevar al cuadrado
    int cuadrado = n * n;

    // 3) Imprimir resultado
    System.out.println("Resultado final: " + cuadrado);
    contadorResultados++;

    // 4) Cortocircuito: detenernos cuando llevemos ya 'limite' resultados
    if (contadorResultados >= limite) {
        break;
    }
}
```



Streams: colectores

```
// Mismo ejemplo que antes
numeros.stream()
      .filter(n -> n % 2 == 0)
      .map(n -> n * n)
      .limit(2)
      .forEach(System.out::println);


// Convertertir en lista
List<Integer> cuadradosPares = numeros
    .stream()
    .filter(n -> (n % 2) == 0)
    .map(n -> n * n)
    .toList();

System.out.println(cuadradosPares);
// salida: [4, 16]


// Reducir (suma)
List<Integer> suma = numeros
    .stream()
    .filter(n -> (n % 2) == 0)
    .mapToInt(Integer::intValue)
    .sum();

System.out.println(suma);
// salida: 20
```

Optionals

Optional es un contenedor que encapsula valores que pueden existir o no, evitando el uso de null y los problemas asociados. Se introdujo en Java 8

Contenedor Seguro

Envuelve valores que podrían ser nulos en un contenedor tipo. Hace explícito en la firma del método que podría no haber valor.

Previene Excepciones

Elimina NullPointerException forzando a comprobar la presencia del valor. Proporciona métodos seguros para acceder al contenido.

API Fluida

Ofrece métodos como map(), filter() y orElse() para operaciones encadenadas. Funciona perfectamente con expresiones lambda y streams.

Optionals: métodos disponibles

Creación

Optional.of(valor) - No permite valores nulos

Optional.ofNullable(valor) - Acepta valores nulos

Optional.empty() - Crea un Optional vacío

Obtención

get() - Devuelve el valor

orElse(defecto) - Devuelve el valor o un valor por defecto

orElseGet(supplier) - Usa un proveedor para el valor por defecto

orElseThrow() - Lanza excepción si está vacío

Transformación

map(...) - Transforma el valor si existe

flatMap(...) - Evita Optional anidados (Optional<Optional>)

filter(...) - Descarta valores que no cumplen una condición

Optionals: recomendaciones de uso

Evitar como Campos de Clase

Nunca uses Optional como atributo en clases o entidades. Genera serialización complicada y confusión conceptual.

Precaución con get()

Usar get() sin consultar antes isPresent() puede causar NoSuchElementException. Usar orElse() u otros métodos seguros.

No en Colecciones

Evita List u otras colecciones de <Optional>. Filtra valores nulos antes de almacenarlos.

Ideal para Retornos

Úsalo principalmente como tipo de retorno en métodos que podrían no producir valor.

Optionals: ejemplo básico

```
// Creando Optionals
Optional nombreVacio = Optional.empty();
Optional nombrePresente = Optional.of("Juan");
Optional nombreNullable = Optional.ofNullable(getSomeName());

// Comprobando y obteniendo valores
if (nombrePresente.isPresent()) {
    System.out.println("Hola " + nombrePresente.get());
}

// Forma más elegante con expresiones lambda
nombrePresente.ifPresent(nombre -> System.out.println("Hola " + nombre));

// Valores por defecto
String resultado = nombreVacio.orElse("Invitado");
```

Creación flexible

Creamos Optionals vacíos, con valores garantizados o posiblemente nulos.

Verificación segura

Comprobamos la existencia antes de acceder al valor.

Uso funcional

Aprovechamos métodos como ifPresent() para código más limpio.

Optionals: ejemplo

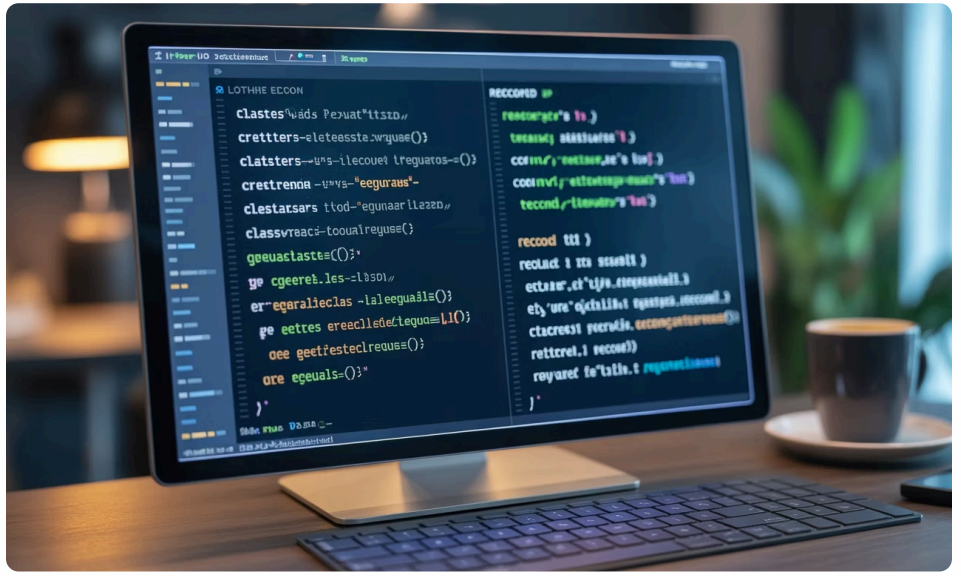
```
CursoDAO cursoDAO= new CursoDAO();

// Intentamos buscar curso con ID = 2 (existe)
String nombre = cursoDAO.findById(2)
    // Si existe, obtenemos el nombre en mayúsculas
    .map(Curso::getName)
    .map(String::toUpperCase)
    // Si no existe, usamos "DESCONOCIDO"
    .orElse("DESCONOCIDO");

System.out.println("Usuario 2: " + nombre);
```

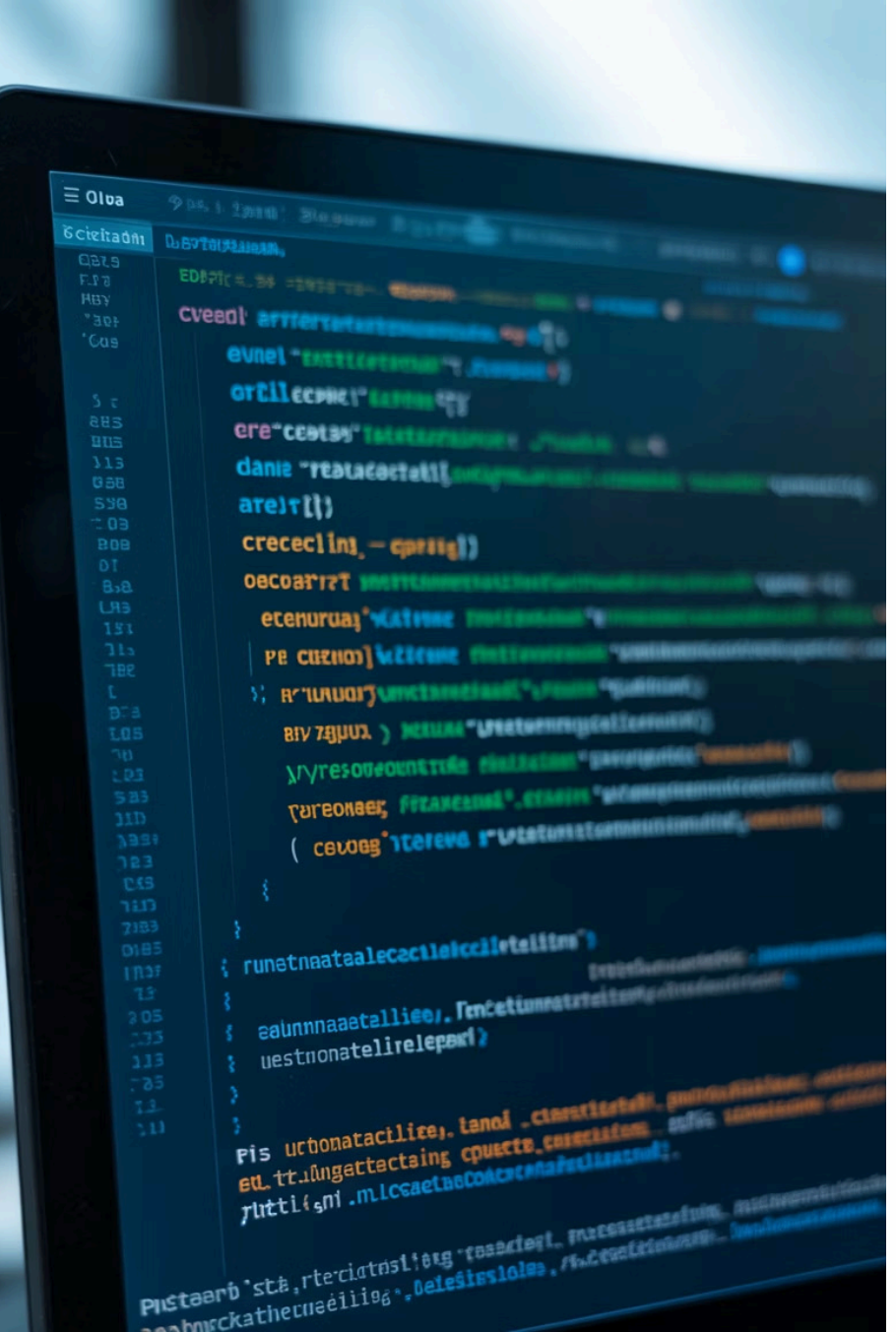
Records

Introducción

Los *records* (añadidos en Java 16) proporcionan una forma **compacta** de definir clases **inmutables** destinadas principalmente a portar datos (*data carriers*)

Reducen significativamente el código boilerplate necesario para clases de datos





Records: características

Código reducido

Reduce el código necesario para clases de datos

Constructor automático

Un record automáticamente genera el *constructor* para todos sus componentes

Getters automáticos

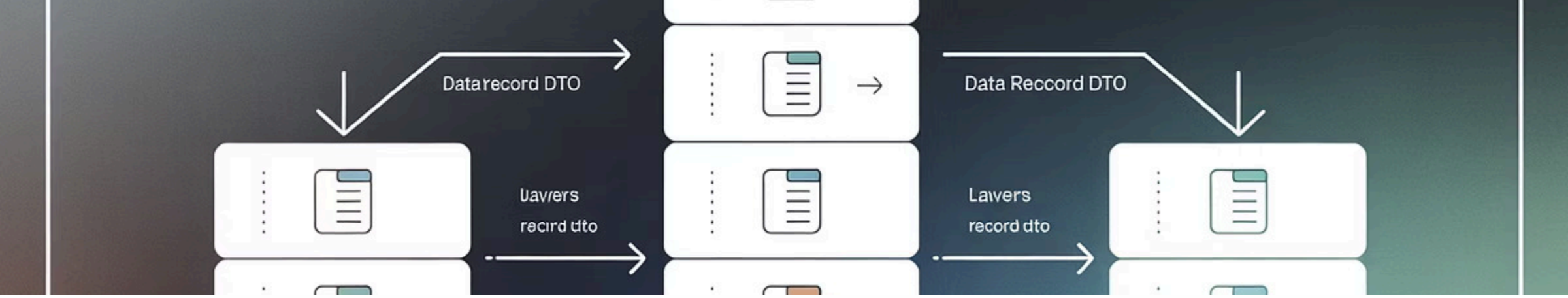
Genera *getters* (métodos accesores para cada campo)

Métodos de utilidad

Implementa automáticamente equals(), hashCode() y toString()

Ojo! Contrato entre `equals` y `hashCode`

- **Qué es la igualdad lógica**
  - `equals(Object o)` compara el contenido de dos objetos, no sus referencias en memoria.
  - Ejemplo: dos objetos `Persona("María", 30)` son iguales si tienen los mismos datos, aunque sean distintas instancias.
- **Reglas básicas**
  a. **Si dos objetos son iguales** (`a.equals(b)` **es** `true`**), deben tener el mismo código hash** (`a.hashCode() == b.hashCode()`**).**
  b. Si no son iguales, es mejor que tengan distintos códigos hash para mejorar el rendimiento.
- **Importancia en colecciones tipo Hash**
  - En `HashSet` o `HashMap`, primero se usa `hashCode()` para ubicar elementos.
  - Si objetos iguales tienen distinto hash, no se detectarán como duplicados.
  - Si objetos diferentes comparten el mismo hash, Java usa `equals()` para diferenciarlos, pero esto ralentiza el proceso.



Records: usos habituales

DTOs (Data Transfer Objects)

Ideal para objetos que transfieren datos entre capas de la aplicación

Respuestas de API

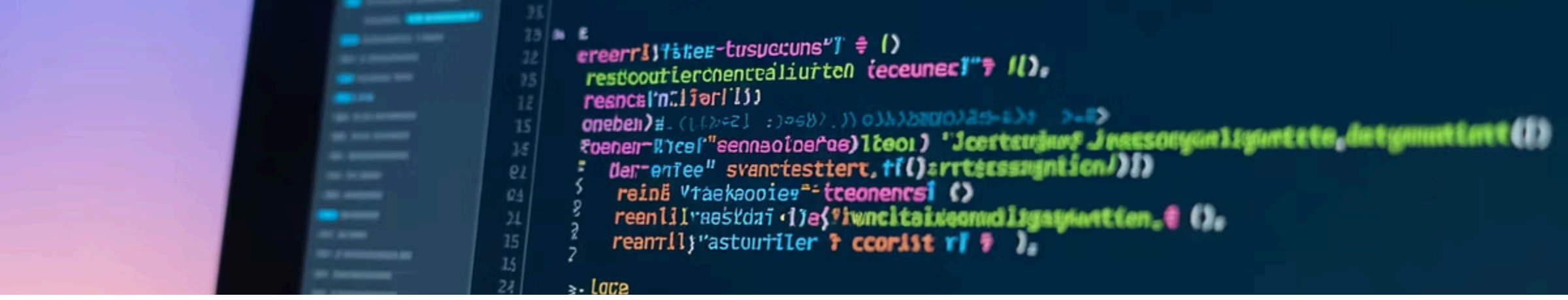
Estructuras de datos para respuestas de servicios web

Tuplas

Perfectos para representar grupos de valores relacionados

Objetos de valor inmutables

Para datos que no deben cambiar después de su creación



Records: ejemplo

Con programación tradicional

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

    public Persona(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }

    public String getNombre() {
        return nombre;
    }

    public int getEdad() {
        return edad;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Persona)) return false;
        Persona persona = (Persona) o;
        return edad == persona.edad &&
               (nombre != null ? nombre.equals(persona.nombre) : persona.nombre == null);
    }

    @Override
    public int hashCode() {
        return Objects.hash(nombre, edad);
    }

    @Override
    public String toString() {
        return "Persona[nombre=" + nombre + ", edad=" + edad + "]";
    }
}
```

Usando un record

```
public record Persona(String nombre, int edad) { }

Persona p = new Persona("Alice", 30);

System.out.println(p.nombre() + " tiene " + p.edad() + " años.");
// Imprime: Alice tiene 30 años.
```

Este ejemplo muestra la simplicidad de definir y usar un record. Con una sola línea se define una clase completa con constructor, getters, equals, hashCode y toString.

Records: diferencias con las clases

Inmutabilidad

Las clases permiten campos mutables o final con gestión manual de getters/setters.

Los records hacen que todos los campos sean implícitamente private final, garantizando completa inmutabilidad.

Generación de Código

Las clases requieren escritura manual de constructores, equals(), hashCode() y toString().

Los records generan automáticamente todos estos métodos sin código repetitivo.

Herencia

Las clases pueden extender otras clases y ser subclasificadas.

Los records no pueden extender ni ser extendidos; solo pueden implementar interfaces.

Propósito

Las clases son versátiles para lógica compleja, comportamiento mutable y patrones de diseño.

Los records están diseñados específicamente para transporte de datos inmutables con código mínimo.

Text Blocks

Sintaxis Moderna para Cadenas Multilínea

Sintaxis Simple

Se delimitan con tres comillas dobles (""") al inicio y al final del contenido.

Eliminación de Sangría

Java elimina automáticamente la sangría común en todas las líneas.

Preservación de Saltos

Mantiene los saltos de línea del código original (JSON, HTML, SQL, etc.) sin caracteres especiales.

Interpolación

Permite combinar con variables usando el método **formatted()**.

```
String nombre = "Juan";
int edad = 30;

String json = """
        {
           "nombre": "%s",
           "edad": %d
        }
        """;

System.out.println(json.formatted(nombre, edad));
```

Switch expressions

Las expresiones switch son una evolución moderna del switch tradicional en Java, permitiendo asignar directamente resultados a variables.

Sintaxis simplificada

Utiliza la flecha (->) en lugar de case/break, eliminando el riesgo de olvidar breaks.

Agrupación de casos

Permite agrupar múltiples casos en una sola línea para un código más compacto.

Retorno de valores

Facilita asignar el resultado a una variable, tratando el switch como una expresión.

Exhaustividad

El compilador verifica que todos los casos posibles estén cubiertos, mejorando la seguridad.

```
String resultado = switch (dia) {
    case LUNES, MARTES -> "Día laborable";
    case SABADO, DOMINGO -> "Fin de semana";
    default -> "Día intermedio";
};
```

Pattern Matching: instanceof

Java 16 introdujo pattern matching para instanceof, simplificando considerablemente la comprobación y conversión de tipos.

```
if (obj instanceof String s) {
    // aquí 's' ya es String, sin cast explícito
    System.out.println(s.toUpperCase());
}
```

Código más conciso, menos propenso a errores y más legible. La variable enlazada solo existe dentro del ámbito donde la condición es verdadera.

Pattern Matching: switch

Java 17 introduce pattern matching para switch, permitiendo comprobar tipos y desestructurar datos en una única expresión.

```
switch (obj) {
    case String s -> System.out.println("Texto: " + s.toUpperCase());
    case Integer i -> System.out.println("Número: " + (i * 2));
    case Record r -> System.out.println("Record: " + r.toString());
    case null -> System.out.println("Es null");
    default -> System.out.println("Otro tipo");
}
```

Sintaxis Concisa

Elimina código repetitivo y reduce la necesidad de castings manuales.

Exhaustividad

El compilador garantiza que se manejen todos los casos posibles.

Expresivo

Permite combinar comprobaciones de tipo con la desestructuración de datos.

Virtual Threads

Introducidos en Java 21, los virtual threads son hilos ligeros para ejecutar código de forma concurrente con mínimo consumo de recursos.

Ligeros

Consumen pocos recursos del sistema, permitiendo crear millones de hilos sin sobrecarga.

Orientados a E/S

Se suspenden automáticamente durante operaciones bloqueantes sin ocupar recursos.

3

API Familiar

Utilizan la misma API que los hilos tradicionales, facilitando su adopción.

Aplicaciones Escalables

Ideales para aplicaciones con alta concurrencia como servidores web.

Virtual threads: ejemplo

```
public class VirtualThreadsEjemplo {

    public static void main(String[] args) throws InterruptedException {
        // Creamos un Virtual Thread que ejecuta un simple mensaje
        Thread hiloVirtual = Thread.startVirtualThread(() -> {
            System.out.println("¡Hola desde un hilo virtual!");
        });

        // Esperamos a que el hilo virtual termine
        hiloVirtual.join();
    }

}
```

JShell: REPL Interactivo de Java

Read-Eval-Print Loop

Introducido en Java 9, JShell proporciona un shell interactivo para probar fragmentos de código instantáneamente. Perfecta para principiantes en Java, exploración de API y prototipado rápido de ideas.

Retroalimentación Inmediata

Los resultados se muestran automáticamente. Las variables persisten entre comandos para un desarrollo iterativo. No se necesitan puntos y comas.

```
jshell> int a=2+4
a ==> 6

jshell> System.out
Signatures:
System.out:java.io.PrintStream

<press tab again to see documentation>
jshell> System.out.p
print(     printf(    println(
jshell> System.out.println("La variable a vale " + a);
La variable a vale 6
```

Resumen

Java ha evolucionado enormemente desde su creación, incorporando características modernas que mejoran su productividad y expresividad.

Las nuevas funcionalidades como lambdas, streams, records y virtual threads transforman el desarrollo.

Programación Funcional

Lambdas e interfaces funcionales facilitan código más conciso y expresivo.

Manipulación de Datos

Streams y Optionals permiten operaciones fluidas y seguras.

Inmutabilidad y Concisión

Records, pattern matching y virtual threads mejoran el rendimiento y legibilidad.