Objetos en Java

(*Desde Java 8 en adelante*)

La **fundamentación teórica de la Programación Orientada a Objetos (POO) se mantiene intacta**, pero el **estilo y las prácticas** han evolucionado significativamente.

¿Cómo enfocar la teoría de Objetos en Java 8+?

No piense solo en *objetos como moldes (clases) y datos*. En Java moderno, debe pensar en **comportamientos y flujos de datos**.

1. **Los Objetos son Contenedores de Datos + Comportamiento (aún cierto)**: Una clase sigue teniendo atributos y métodos. Esto es la base.
2. **Los Objetos son la Fuente de Datos para *Streams***: En lugar de usar bucles `for` externos que modifican objetos, ahora extrae datos de colecciones de objetos y los procesas de forma declarativa (tal como lo hace SQL). Ejemplo: “Tengo una lista de objetos *Usuario*; y voy a crear un `stream` para filtrar, mapear y recolectar resultados sin modificar la lista original”.
3. **Inmutabilidad como Prioridad:** Java 8+ (y sobre todo 17+) fomenta que los objetos sean **inmutables** (usando ***record*** o clases con ***final***, sin *setters*). Esto hace que el código concurrente sea más seguro y predecible.
4. **Interfaces Funcionales y Lambdas:** Aquí hay un cambio de paradigma. Ahora los métodos pueden recibir *comportamiento* (una función) como parámetro. Esto reduce la necesidad de crear clases anónimas internas para una sola operación. Un objeto puede delegar parte de su comportamiento a través de lambdas.

Diferencias Notables en Objetos: Java 8 vs Java 5/6/7

Las diferencias en la **definición** de objetos son mínimas. Las diferencias en el **uso, creación y manejo** de objetos son **enormes**.

Característica	Java 7 y anteriores	Java 8+	Impacto en Objetos
Métodos por defecto en interfaces	**No existían**. Las interfaces solo tenían métodos abstractos.	**Sí**. Una interfaz puede tener métodos con implementación (`default` y `static`)	**Profundo** Ahora las interfaces pueden evolucionar sin romper implementaciones existentes. Un objeto puede heredar comportamiento de múltiples interfaces.
Interfaces funcionales y Lambdas	**No existían**. Necesitabas clases anónimas (verbosas)	**Sí**. Una interfaz con un solo método abstracto puede instanciarse con lambda.	**Profundo** Reduce la necesidad de crear *objetos de una sola vez* (como `Runnable`, `Comparator`). El objeto es reemplazado por la lambda (aunque en bytecode sigue siendo un objeto sintético)
Streams y API de Colecciones	Bucles `for`, `while`, iteradores manuales. Modificabas los objetos directamente.	Procesamiento declarativo: `filter()`, `map()`, `reduce()` sobre `streams`. Se fomenta la inmutabilidad.	**Notable** Ya no iteras sobre objetos *para modificarlos*; transformas flujos de datos. Los objetos pasan a ser datos que fluyen.
OptionalT	Usabas `null` y te arriesgabas a `NullPointerException`.	Contenedor que puede o no contener un objeto no nulo.	**Notable**. Expresa explícitamente que un objeto puede estar *vacío*. El código obliga a manejar la ausencia de objeto (`.orElse()`, `.ifPresent()`).
`java.time` **(nuevo API de fechas)**	`Date` y `Calendar` eran mutables, confusos e inseguros.	`LocalDate`, `Instant`, `ZonedDateTime` son objetos **inmutables**.	**Notable.** Los objetos de fecha ya no se modifican, crean nuevos objetos. Refuerza la inmutabilidad.
`Nashorn` **(JavaScript en JVM)**	Solo relevante para integración.	Motor JS que permite invocar objetos Java desde JS y viceversa.	Menor impacto para POO general.

Diferencias en versiones *posteriores* a Java 8 (9, 11, 17, 21)

Aquí sí hay **nuevas formas de definir objetos** que son MUY importantes:

Versión	Característica	Cambio en Objetos
Java 14/17	**Records**	**REVOLUCIONARIO.** Una forma compacta de definir objetos que son solo portadores de datos (inmutables). Una línea de código genera constructor, `getters`, `equals()`, `hashCode()`, `toString()`. Ej: `record Persona(String nombre, int edad) {}`
Java 14/17	**Pattern Matching for** `instanceof`	Simplifica la comprobación y conversión de tipos de objetos: `if (obj instanceof Persona p) p.nombre();`
Java 15/17	**Text Blocks**	Mejor para crear objetos que contienen cadenas multilínea (JSON, SQL, HTML)
Java 17	**Sealed Classes**	Restringe qué clases o interfaces pueden extender/implementar una clase/interfaz. Control más estricto de la jerarquía de objetos.
Java 21	**Pattern Matching for** `switch`	Puedes hacer `switch` sobre el **tipo** de objeto y extraer sus atributos directamente.

Consejos prácticos

- **Domina bien la POO clásica de Java 7** (clases, herencia, polimorfismo, encapsulación). Es la base.
- **Aprende Lambdas + Streams** como si fuera un nuevo idioma. Practica convertir bucles ‘for‘ en pipelines de streams.
- **Adopta ‘Optional‘** en lugar de devolver ‘null‘ en tus propios métodos.
- **Cuando uses Java 17+ (estándar hoy), prefiere ‘record‘** para objetos de datos (DTO, Value Objects, entidades simples).
- **Considera la inmutabilidad** como primera opción. Pregúntate: *¿Este objeto necesita cambiar realmente o puedo crear uno nuevo?*
- **Entre Java 7 y Java 8**: La diferencia NO está en *cómo se definen los objetos*, sino en cómo se usan (streams, lambdas, opcionales, inmutabilidad).
- **Entre Java 8 y versiones posteriores (17+)**: Aparecen nuevas formas sintácticas de definir objetos (**`record`**, **`sealed`**, **`pattern matching`**) que hacen el código mucho más conciso y expresivo.

Comparación entre Java 7, Java 8 y Java 21

El problema

Tenemos una lista de objetos **`Persona`** y queremos:

1. Filtrar mayores de edad ($\geq$ 18 años)
2. Obtener sus nombres en mayúsculas
3. Ordenarlos alfabéticamente
4. Mostrarlos en consola

Java 7, estilo tradicional

```
import java.util.*;

public class MainJava7 {
    public static void main(String[] args) {
        // Crear objetos manualmente
        List<Persona> personas = Arrays.asList(
            new Persona("Ana", 25),
            new Persona("Luis", 17),
            new Persona("Carlos", 30),
            new Persona("Beatriz", 20),
            new Persona("Zoe", 15)
        );

        // 1. Filtrar mayores de edad
        List<Persona> mayoresEdad = new ArrayList<>();
        for (Persona p : personas) {
            if (p.getEdad() >= 18) {
                mayoresEdad.add(p);
            }
        }
```

```
        // 2. Obtener nombres en mayúsculas
        List<String> nombresMayusculas = new ArrayList<>();
        for (Persona p : mayoresEdad) {
            nombresMayusculas.add(p.getNombre().toUpperCase());
        }

        // 3. Ordenar alfabéticamente
        Collections.sort(nombresMayusculas, new Comparator<String>() {
            @Override
            public int compare(String s1, String s2) {
                return s1.compareTo(s2);
            }
        });

        // 4. Mostrar resultados
        for (String nombre : nombresMayusculas) {
            System.out.println(nombre);
        }
        // Salida: ANA, BEATRIZ, CARLOS
    }
}

// Clase definida aparte
class Persona {
    private String nombre;
    private int edad;

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

    public String getNombre() { return nombre; }
    public int getEdad() { return edad; }
}
```

Problema: Código verboso, muchos bucles, mutabilidad, lógica imperativa paso a paso.

Java 8, Streams + Lambda + Optional

```
import java.util.*;
import java.util.stream.*;

public class MainJava8 {
    public static void main(String[] args) {
        List<Persona> personas = Arrays.asList(
            new Persona("Ana", 25),
            new Persona("Luis", 17),
            new Persona("Carlos", 30),
            new Persona("Beatriz", 20),
            new Persona("Zoe", 15)
        );

        // Todo el procesamiento en UNA cadena funcional
        personas.stream()
            .filter(p -> p.getEdad() >= 18)               // Filtro con lambda
            .map(p -> p.getNombre().toUpperCase())        // Transformación
            .sorted()                                     // Orden natural
            .forEach(System.out::println);                // Acción final

        // Salida: ANA, BEATRIZ, CARLOS
    }
}

// La clase Persona es igual que en Java 7
```

Ventajas:

- Código declarativo (qué hacer, no cómo)
- Sin bucles explícitos ni colecciones intermedias mutables
- Fácil paralelización (`.parallelStream()`)

Java 17+ (Records + Pattern Matching + Mejores prácticas)

```
import java.util.*;
import java.util.stream.*;
```

```
// RECORD: Una línea para definir objeto de datos inmutable
public record Persona(String nombre, int edad) {}

public class MainJava17 {
    public static void main(String[] args) {
        // Crear objetos sin 'new' explícito (records tienen constructor automático)
        var personas = List.of(
            new Persona("Ana", 25),
            new Persona("Luis", 17),
            new Persona("Carlos", 30),
            new Persona("Beatriz", 20),
            new Persona("Zoe", 15)
        );

        // Pattern Matching con switch (Java 17+ preview, 21 estable)
        String resultado = procesarPersonas(personas);
        System.out.println(resultado);

        // O también con el mismo stream de Java 8 (pero con records más limpios)
        personas.stream()
            .filter(Persona::mayorDeEdad)                  // Referencia a método
            .map(p -> p.nombre().toUpperCase())            // Acceso directo (sin getters)
            .sorted()
            .forEach(System.out::println);
    }

    // Método que demuestra Pattern Matching con switch (Java 21)
    static String procesarPersonas(List<Persona> lista) {
        return switch (lista.size()) {
            case 0 -> "Lista vacía";
            case 1 -> "Una persona: " + lista.get(0).nombre();
            default -> "Total: " + lista.size() + " personas";
        };
    }
}

// Método auxiliar dentro del record (¡se pueden agregar métodos!)
record Persona(String nombre, int edad) {
    public boolean mayorDeEdad() {  // Método personalizado
        return edad >= 18;
    }
}
```

Ventajas:

- **`record`**: Adiós a `getters`, constructores, `equals`/`hashCode`. Todo automático.
- Acceso directo: **`p.nombre()`** (no `p.getNombre()`)
- **`var`**: Inferencia de tipos local
- **`List.of()`**: Crea lista inmutable desde el inicio
- `Pattern Matching` en switch: El ejemplo del tamaño de lista

Tabla Comparativa de Características de Objetos

Aspecto	Java 7	Java 8	Java 17/21
Definir objeto simple	Clase con 20 líneas (campos, constructor, **`getters`**)	Igual que Java 7	**`record Persona(String nombre, int edad)`**
Mutabilidad	Por defecto mutable (con **`setters`**)	Recomienda inmutabilidad	**`Records`** son inmutables por diseño
Iterar objetos	Bucles for explícitos	Streams con lambdas	**`Streams`** + métodos de referencia
Manejar null	**`if (objeto != null)`**	**`Optional.ofNullable()`**	**`Optional + pattern matching`**
Comparar objetos	Implementar **`Comparable`** manual	**`Comparator.comparing()`**	**`Records`** tienen **`equals`** automático
Extensibilidad	Herencia clásica	**`Default methods`** en interfaces	**`Sealed classes`** (control de herencia)

Ejemplo: Diferencias en la práctica diaria

Escenario: Buscar una persona por nombre

Java 7

```
Persona encontrar(List<Persona> lista, String nombre) {
    for (Persona p : lista) {
        if (p.getNombre().equals(nombre)) {
            return p;
        }
    }
    return null; // Peligro de NPE
}

// Uso riesgoso
Persona p = encontrar(personas, "Luis");
System.out.println(p.getNombre().toUpperCase()); // NullPointerException si no existe
```

Java 8+ (con Optional)

```
Optional<Persona> encontrar(List<Persona> lista, String nombre) {
    return lista.stream()
        .filter(p -> p.getNombre().equals(nombre))
        .findFirst();  // Retorna Optional, nunca null
}

// Uso seguro
encontrar(personas, "Luis")
    .map(p -> p.getNombre().toUpperCase())
    .ifPresentOrElse(
        System.out::println,
        () -> System.out.println("No encontrado")
    );
```

Java 17+ (con records)

```
// El record ya tiene equals/hashCode, podemos usarlo directamente
Optional<Persona> encontrada = personas.stream()
    .filter(p -> p.nombre().equals("Luis")) // Sin getters
    .findFirst();

// Pattern matching con switch (Java 21)
switch (encontrada) {
    case Optional<Persona> (var p) -> System.out.println(p.nombre()); // Deconstrucción
    case Optional<?> empty -> System.out.println("Vacío");
}
```

Caso Práctico: Sistema de Gestión de Empleados

Requisitos:

1. Calcular el bono anual de los empleados (10
2. Filtrar empleados que ganan menos de cierto umbral
3. Ordenar por salario descendente
4. Generar un reporte con formato especial

Java 7 (Base tradicional)

```
import java.util.*;

// Clase Empleado tradicional
class Empleado {
    private String nombre;
    private double salario;
    private int hijos;
    private String departamento;

    public Empleado(String nombre, double salario, int hijos, String departamento) {
        this.nombre = nombre;
        this.salario = salario;
        this.hijos = hijos;
        this.departamento = departamento;
    }

    // Getters y setters (solo getters para este ejemplo)
    public String getNombre() { return nombre; }
    public double getSalario() { return salario; }
    public int getHijos() { return hijos; }
    public String getDepartamento() { return departamento; }

    @Override
    public String toString() {
        return String.format("%s - %.2f€ - %d hijos", nombre, salario, hijos);
    }
}

// Clase principal con toda la lógica imperativa
public class EmpresaJava7 {

    // Método para calcular bono (lógica de negocio)
    public static double calcularBono(Empleado e) {
        double bonoBase = e.getSalario() * 0.10;
        double bonoHijos = e.getHijos() * 500;
        return bonoBase + bonoHijos;
    }

    // Método para filtrar empleados por salario
    public static List<Empleado> filtrarPorSalario(List<Empleado> empleados, double umbral) {
        List<Empleado> resultado = new ArrayList<>();
        for (Empleado e : empleados) {
            if (e.getSalario() < umbral) {
                resultado.add(e);
            }
        }
        return resultado;
    }

    // Método para ordenar por salario descendente
    public static void ordenarPorSalarioDesc(List<Empleado> empleados) {
        Collections.sort(empleados, new Comparator<Empleado>() {
            @Override
            public int compare(Empleado e1, Empleado e2) {
                // Orden descendente: mayor salario primero
                return Double.compare(e2.getSalario(), e1.getSalario());
            }
        });
    }

    // Método para generar reporte
    public static List<String> generarReporte(List<Empleado> empleados) {
        List<String> reporte = new ArrayList<>();
        for (Empleado e : empleados) {
            double bono = calcularBono(e);
            String linea = e.getNombre() + " | Salario: " + e.getSalario() +
                           " | Bono: " + bono + " | Total: " + (e.getSalario() + bono);
            reporte.add(linea);
        }
        return reporte;
    }

    public static void main(String[] args) {
        // Crear lista de empleados
        List<Empleado> empleados = new ArrayList<>();
        empleados.add(new Empleado("Ana García", 25000, 2, "Ventas"));
        empleados.add(new Empleado("Luis Pérez", 18000, 0, "Soporte"));
        empleados.add(new Empleado("Carlos López", 32000, 3, "IT"));
        empleados.add(new Empleado("Beatriz Ruiz", 22000, 1, "Ventas"));
        empleados.add(new Empleado("David Gómez", 15000, 0, "Soporte"));

        // Procesamiento paso a paso
        System.out.println("=== EMPLEADOS CON SALARIO < 25000 ===");
        List<Empleado> filtrados = filtrarPorSalario(empleados, 25000);
        ordenarPorSalarioDesc(filtrados);
        List<String> reporte = generarReporte(filtrados);

        // Mostrar reporte
        for (String linea : reporte) {
            System.out.println(linea);
        }
        // Salida esperada:
        // Beatriz Ruiz | Salario: 22000.0 | Bono: 2700.0 | Total: 24700.0
        // Luis Pérez | Salario: 18000.0 | Bono: 1800.0 | Total: 19800.0
        // David Gómez | Salario: 15000.0 | Bono: 1500.0 | Total: 16500.0
    }
```

```
}
```

Problemas detectados

- Código verboso (muchas líneas para operaciones simples)
- Bucles manuales repetitivos
- Mutabilidad intermedia (creamos listas temporales)
- Dificultad para paralelizar
- Clase anónima para Comparator (mucho boilerplate)

Java 8 (Streams + Lambdas)

```
import java.util.*;
import java.util.stream.*;

// Clase Empleado (misma estructura que Java 7, pero añadimos métodos de negocio)
class Empleado {
    private String nombre;
    private double salario;
    private int hijos;
    private String departamento;

    public Empleado(String nombre, double salario, int hijos, String departamento) {
        this.nombre = nombre;
        this.salario = salario;
        this.hijos = hijos;
        this.departamento = departamento;
    }

    // Getters
    public String getNombre() { return nombre; }
    public double getSalario() { return salario; }
    public int getHijos() { return hijos; }
    public String getDepartamento() { return departamento; }

    // Métodos de negocio dentro de la clase (más cohesión)
    public double calcularBono() {
        return salario * 0.10 + hijos * 500;
    }

    public double getTotalAnual() {
        return salario + calcularBono();
    }

    @Override
    public String toString() {
        return String.format("%s - %.2f€ - %d hijos", nombre, salario, hijos);
    }
}

public class EmpresaJava8 {

    // Método auxiliar para crear reporte (opcional, puede ser parte del stream)
    private static String formatearReporte(Empleado e) {
        return String.format("%s | Salario: %.2f | Bono: %.2f | Total: %.2f",
            e.getNombre(), e.getSalario(), e.calcularBono(), e.getTotalAnual());
    }

    public static void main(String[] args) {
        // Crear lista (más concisa con Arrays.asList)
        List<Empleado> empleados = Arrays.asList(
            new Empleado("Ana García", 25000, 2, "Ventas"),
            new Empleado("Luis Pérez", 18000, 0, "Soporte"),
            new Empleado("Carlos López", 32000, 3, "IT"),
            new Empleado("Beatriz Ruiz", 22000, 1, "Ventas"),
            new Empleado("David Gómez", 15000, 0, "Soporte")
        );

        System.out.println("=== EMPLEADOS CON SALARIO < 25000 (Java 8 Style) ===");

        // TODO el procesamiento en UNA cadena funcional
        empleados.stream()                          // Crear stream
            .filter(e -> e.getSalario() < 25000)    // Filtrar
            .sorted((e1, e2) -> Double.compare(     // Ordenar con lambda
                e2.getSalario(), e1.getSalario()))  // Descendente
            .map(EmpresaJava8::formatearReporte)    // Transformar a reporte
            .forEach(System.out::println);          // Imprimir

        // Versión alternativa con Comparator más elegante
        System.out.println("\n=== MISMO RESULTADO (con Comparator.comparing) ===");
```

```
        empleados.stream()
            .filter(e -> e.getSalario() < 25000)
            .sorted(Comparator.comparingDouble(Empleado::getSalario).reversed())
            .map(e -> String.format("%s | Salario: %.2f | Bono: %.2f | Total: %.2f",
                e.getNombre(), e.getSalario(), e.calcularBono(), e.getTotalAnual()))
            .forEach(System.out::println);

        // Demostración adicional: Agrupar por departamento (nueva capacidad)
        System.out.println("\n=== BONO PROMEDIO POR DEPARTAMENTO ===");
        Map<String, Double> bonoPorDepto = empleados.stream()
            .collect(Collectors.groupingBy(
                Empleado::getDepartamento,
                Collectors.averagingDouble(Empleado::calcularBono)
            ));
        bonoPorDepto.forEach((depto, bono) ->
            System.out.printf("%s: %.2f€%n", depto, bono));

        // Calcular estadísticas con streams
        System.out.println("\n=== ESTADÍSTICAS DE SALARIOS ===");
        DoubleSummaryStats stats = empleados.stream()
            .filter(e -> e.getSalario() < 25000)
            .mapToDouble(Empleado::getSalario)
            .summaryStatistics();
        System.out.printf("Empleados: %d, Mín: %.2f, Máx: %.2f, Prom: %.2f%n",
            stats.getCount(), stats.getMin(), stats.getMax(), stats.getAverage());
    }
}
```

Mejoras respecto a Java 7:

- Código 70 % más corto y legible
- Sin bucles explícitos ni colecciones intermedias
- Declarativo: expresamos “qué” hacer, no “cómo”
- Lambdas eliminan clases anónimas
- Streams permiten operaciones encadenadas
- Collectors para agrupaciones complejas
- Fácil paralelización (**`.parallelStream()`**)

Java 21 (Records + Pattern Matching + Mejoras modernas)

```
import java.util.*;
import java.util.stream.*;
import java.time.Year;

// RECORD: Elimina todo el boilerplate de la clase
record Empleado(
    String nombre,
    double salario,
    int hijos,
    String departamento
) implements Comparable<Empleado> {

    // Método compacto del record (validación)
    public Empleado {
        if (salario < 0) throw new IllegalArgumentException("Salario negativo");
        if (hijos < 0) throw new IllegalArgumentException("Hijos negativo");
        if (departamento == null || departamento.isBlank())
            throw new IllegalArgumentException("Departamento inválido");
    }

    // Métodos de negocio (siguen funcionando igual)
    public double calcularBono() {
        return salario * 0.10 + hijos * 500;
    }

    public double getTotalAnual() {
        return salario + calcularBono();
    }

    // Método de ejemplo con pattern matching (Java 21)
    public String obtenerCategoria() {
        return switch (calcularBono()) {
            case double b when b < 2000 -> "Bono bajo";
            case double b when b < 5000 -> "Bono medio";
            case double b -> "Bono alto";
```

```
        };
    }

    // Implementación de Comparable usando pattern matching
    @Override
    public int compareTo(Empleado other) {
        // Pattern matching en instanceof (Java 21)
        if (other instanceof Empleado e) {
            return Double.compare(this.salario, e.salario);
        }
        return 0;
    }
}

public class EmpresaJava21 {

    // Método principal con nuevas características
    public static void main(String[] args) {
        // var para inferencia de tipos
        var empleados = List.of(  // List.of crea lista inmutable
            new Empleado("Ana García", 25000, 2, "Ventas"),
            new Empleado("Luis Pérez", 18000, 0, "Soporte"),
            new Empleado("Carlos López", 32000, 3, "IT"),
            new Empleado("Beatriz Ruiz", 22000, 1, "Ventas"),
            new Empleado("David Gómez", 15000, 0, "Soporte")
        );

        System.out.println("=== EMPLEADOS CON SALARIO < 25000 (Java 21 Style) ===");

        // Mismo stream pero más limpio (sin getters)
        empleados.stream()
            .filter(e -> e.salario() < 25000)      // Acceso directo a campos
            .sorted(Comparator.comparingDouble(Empleado::salario).reversed())
            .map(e -> formatearReporteModerno(e))   // Text block en método
            .forEach(System.out::println);

        // Pattern matching con switch enriquecido
        System.out.println("\n=== PROCESAMIENTO CON PATTERN MATCHING ===");
        procesarEmpleado(empleados.get(0));  // Ana
        procesarEmpleado(null);              // Caso null

        // Uso de Optional y pattern matching
        System.out.println("\n=== BUSCANDO EMPLEADO MÁS ANTIGUO (SIMULADO) ===");
        Optional<Empleado> empleadoAntiguo = encontrarEmpleadoAntiguo(empleados);

        // Pattern matching con Optional (Java 21 preview)
        switch (empleadoAntiguo) {
            case Optional<Empleado> (var emp) ->
                System.out.printf("Encontrado: %s con salario %.2f%n",
                    emp.nombre(), emp.salario());
            case Optional<?> empty ->
                System.out.println("No se encontró empleado");
        }

        // Agrupación mejorada con records y toList()
        System.out.println("\n=== BONO PROMEDIO POR DEPARTAMENTO (Java 21) ===");
        var bonoPorDepto = empleados.stream()
            .collect(Collectors.groupingBy(
                Empleado::departamento,           // Referencia directa
                Collectors.averagingDouble(Empleado::calcularBono)
            ));
        bonoPorDepto.forEach((depto, bono) ->
            System.out.printf("%s: %.2f€%n", depto, bono));

        // Demostración de text blocks (Java 15+)
        System.out.println("\n" + generarReporteCompleto(empleados));
    }

    // Método con text block (Java 15/17/21)
    private static String generarReporteCompleto(List<Empleado> empleados) {
        var stats = empleados.stream()
            .mapToDouble(Empleado::salario)
            .summaryStatistics();

        return """
            ===================================
            REPORTE DE EMPLEADOS
            ===================================
            Total empleados: %d
            Salario promedio: %.2f€
            Bono total: %.2f€
            ===================================
            """.formatted(
                stats.getCount(),
                stats.getAverage(),
                empleados.stream().mapToDouble(Empleado::calcularBono).sum()
            );
    }

    // Método con pattern matching en switch (Java 21)
    private static void procesarEmpleado(Empleado emp) {
        // Pattern matching con null safety
        switch (emp) {
            case null -> System.out.println("Empleado nulo");
            case Empleado e when e.salario() < 20000 ->
                System.out.println("Empleado con salario bajo: " + e.nombre());
            case Empleado e when e.salario() < 30000 ->
                System.out.println("Empleado con salario medio: " + e.nombre());
            case Empleado e ->
```

```
                System.out.println("Empleado con salario alto: " + e.nombre());
            }
        }

        // Método auxiliar con formato
        private static String formatearReporteModerno(Empleado e) {
            // String.format sigue siendo útil
            return String.format("%s | Salario: %.2f | Bono: %.2f | Total: %.2f | %s",
                e.nombre(), e.salario(), e.calcularBono(),
                e.getTotalAnual(), e.obtenerCategoria());
        }

        // Simular búsqueda (devuelve Optional)
        private static Optional<Empleado> encontrarEmpleadoAntiguo(List<Empleado> empleados) {
            // Simulación: empleado con mayor experiencia (por nombre alfabético)
            return empleados.stream()
                .min(Comparator.comparing(Empleado::nombre));
        }
}
```

Comparativa Directa de Características

Característica	Java 7	Java 8	Java 21
Definición de clase	~25 líneas (campos, constructor, getters)	~25 líneas (igual)	1 línea (record)
Acceso a propiedades	`obj.getNombre()`	`obj.getNombre()`	`obj.nombre()`
Filtrar lista	Bucle `for` + condicional + lista temporal	`.filter(lambda)`	`.filter(lambda)` (mismo que Java 8)
Ordenar	`Collections.sort` + clase anónima	`.sorted(lambda)`	`.sorted (Comparator.comparing(...))`
Transformar datos	Bucle + añadir a lista	`.map()`	`.map()` (mejor con text blocks)
Manejo de null	`if (obj == null)`	`Optional.ofNullable()`	`Optional + pattern matching`
Validaciones	En constructor manual	En constructor manual	En record (sintaxis compacta)
Agrupaciones	Bucles + Map manual	`Collectors.groupingBy`	Mismo + `toList()` más limpio
Formato de strings	Concatenación o `String.format`	Igual	`Text blocks + formatted()`