Guia de Repaso

Enums, Clases Anidadas, Singleton, Comparable y Comparator

Semana 02 - Martes 17 de Febrero de 2026

Academia Java - Ciudad de Mexico

Temas del dia:

1. Enums en Java
2. Clases anidadas (Nested Classes)
3. Patron de diseno Singleton
4. Comparable: ordenamiento natural
5. Comparator: ordenamiento personalizado

Instructor: Miguel Rugerio

1. Enums en Java

Un **enum** es un tipo especial de clase que representa un conjunto fijo de constantes. Antes de Java 5, se simulaban con clases finales y constantes estaticas.

Antes de Java 5: simulacion manual (v00)

Asi se simulaba un enum antes de que existiera la palabra clave `enum`:

```
public final class DiaSemana {

    private final String nombre;
    private final int ordinal;

    public static final DiaSemana LUNES     = new DiaSemana("LUNES", 0);
    public static final DiaSemana MARTES    = new DiaSemana("MARTES", 1);
    public static final DiaSemana MIERCOLES = new DiaSemana("MIERCOLES", 2);
    // ... etc

    private DiaSemana(String nombre, int ordinal) {
        this.nombre = nombre;
        this.ordinal = ordinal;
    }

    public String name() { return nombre; }
    public int ordinal() { return ordinal; }
}
```

> El constructor es **privado** para evitar instancias externas. Los metodos `name()`, `ordinal()`, `values()` y `valueOf()` replican lo que Java 5+ nos da gratis.

Declaracion basica de un enum (v0)

Con Java 5+, la declaracion se simplifica enormemente:

```
enum DiaSemana {
    LUNES, MARTES, MIERCOLES, JUEVES, VIERNES,
    SABADO, DOMINGO
```

```
}

DiaSemana hoy = DiaSemana.MARTES;
System.out.println(hoy); // MARTES
```

Enums con switch (v1)

Los enums funcionan directamente con `switch`:

```
DiaSemana hoy = DiaSemana.DOMINGO;

switch(hoy) {
case LUNES: System.out.println("lunes");
    break;
case MARTES: System.out.println("martes");
    break;
case MIERCOLES,JUEVES: System.out.println("miercoles,jueves");
    break;
default: System.out.println("otro dia");
}
```

Nota: En el `switch` se usan los nombres simples (`LUNES`), no el nombre completo (`DiaSemana.LUNES`).

Metodos utiles: values(), ordinal(), name() (v2)

```
for (DiaSemana ds : DiaSemana.values()) {
    System.out.print(ds);             // enum (toString)
    System.out.print(" " + ds.ordinal()); // posicion (0-indexed)
    System.out.println(" " + ds.name());  // String
}
```

Metodo	Retorna	Ejemplo
`values()`	Array con todas las constantes	`DiaSemana[]`
`ordinal()`	Posicion (empieza en 0)	`LUNES → 0`
`name()`	Nombre como String	`"LUNES"`
`valueOf(String)`	Constante a partir de un String	`DiaSemana.valueOf("LUNES")`

Enums con constructor y propiedades (v4)

Los enums pueden tener atributos, constructores y metodos:

```
enum DiaSemana {
    LUNES("Bajo"),
    MARTES("Bajo"),
    MIERCOLES("Bajo"),
    JUEVES("Medio"),
    VIERNES("Medio"),
    SABADO("Alto"),
    DOMINGO("Alto");

    String visitantes;

    DiaSemana(String visitantes) {
        this.visitantes = visitantes;
    }
}

DiaSemana hoy = DiaSemana.SABADO;
System.out.println(hoy.visitantes); // Alto
```

Importante: El constructor de un enum es siempre **privado** (implicito o explicito). No puedes hacer `new DiaSemana("Bajo")`.

Enums con metodos abstractos (v5)

Cada constante puede sobrescribir un metodo abstracto, logrando **polimorfismo**:

```
enum DiaSemana {
    LUNES("Bajo") {
```

```
        @Override
        void getVisitantes() {
            System.out.println("Visitantes baja demanda");
        }
    },
    SABADO("Alto") {
        @Override
        void getVisitantes() {
            System.out.println("Visitantes alta demanda");
        }
    };
    // ... demas constantes

    private String visitantes;
    DiaSemana(String visitantes) { this.visitantes = visitantes; }

    abstract void getVisitantes();
}

DiaSemana hoy = DiaSemana.SABADO;
hoy.getVisitantes(); // Visitantes alta demanda
```

Regla: Si un enum tiene un metodo abstracto, **todas** las constantes deben implementarlo.

2. Clases anidadas (Nested Classes)

Java permite definir clases dentro de otras clases. Existen **cuatro tipos** de clases anidadas:

Tipo	Palabra clave	Instanciacion
Inner Class	(ninguna)	`new Outer().new Inner()`
Static Nested	`static`	`new Outer.Inner()`
Local Class	(dentro de metodo)	`new Local()`
Anonymous Class	(sin nombre)	`new Interface() { }`

Inner Class (clase interna)

Necesita una instancia de la clase externa para ser creada:

```
class Animal {
    String name;
    class Pato {} // Inner Class
}

// Opcion 1: en una linea
Animal.Pato pato1 = new Animal().new Pato();

// Opcion 2: con referencia al outer
Animal animal = new Animal();
Animal.Pato pato2 = animal.new Pato();
```

> La inner class tiene acceso a todos los atributos de la clase externa, incluyendo los **privados**.

Static Nested Class (clase estatica anidada)

Le pertenece a la clase, no a una instancia:

```
class Animal {
    static int contador;
```

```
    static class Pato {} // Static nested class
}

// No necesita instancia del outer
Animal.Pato pato1 = new Animal.Pato();
```

Diferencia clave: La static nested class **no** puede acceder a miembros de instancia de la clase externa. Solo a los `static`.

Local Class (clase local)

Se define dentro de un metodo. Su alcance se limita a ese metodo:

```
public static void main(String[] args) {

    class Pato {} // Local class

    Pato pato1 = new Pato();
    System.out.println(pato1);
}
```

Anonymous Class (clase anonima)

Clase sin nombre que se crea en linea. Puede implementar una interfaz, extender una clase abstracta, o extender una clase concreta:

```
// v0: Implementando una interface
interface Animal {}
Animal pato = new Animal() {};

// v1: Extendiendo una clase abstracta
abstract class Animal {}
Animal pato = new Animal() {};

// v2: Extendiendo una clase concreta
class Animal {}
Animal pato = new Animal() {};
```

Importante: La clase anonima crea una **subclase** en todos los casos. El `{}` despues del constructor es el cuerpo de la nueva clase.

3. Patron de diseno Singleton

El patron **Singleton** garantiza que una clase tenga **una sola instancia** en toda la aplicacion, con un punto de acceso global a ella.

Receta del Singleton

Paso	Que hacer
1	Constructor **privado**
2	Atributo `static private` del mismo tipo
3	Metodo `static public getInstance()`

Lazy Initialization (v0)

La instancia se crea **la primera vez** que se solicita:

```
public class ConexionDB {

    static private ConexionDB conexion;

    private ConexionDB() {
        System.out.println("PASO POR CONSTRUCTOR!");
    }

    static public ConexionDB getInstance() {
        if (conexion == null)
            conexion = new ConexionDB();
        return conexion;
    }
}
```

Problema: No es thread-safe. Si dos hilos llaman a `getInstance()` al mismo tiempo cuando `conexion` es null, ambos pueden crear una instancia.

Eager Initialization (v1)

La instancia se crea **al cargar la clase**:

```
public class ConexionDB {

    static private ConexionDB conexion = new ConexionDB();

    private ConexionDB() {
        System.out.println("PASO POR CONSTRUCTOR!");
    }

    static public ConexionDB getInstance() {
        return conexion;
    }
}
```

> **Ventaja:** Thread-safe por diseno. La JVM garantiza que la inicializacion de la clase es atomica.
> **Trade-off:** la instancia se crea aunque nunca se use.

Aspecto	Lazy (v0)	Eager (v1)
Cuando se crea	Al primer `getInstance()`	Al cargar la clase
Thread-safe	**No**	**Si**
Memoria	Eficiente (bajo demanda)	Se reserva desde el inicio
Complejidad	Requiere verificacion null	Simple y directo

4. Comparable: ordenamiento natural

La interfaz `Comparable<T>` define el **orden natural** de los objetos de una clase. Permite usar `Arrays.sort()` y `Collections.sort()` directamente.

El problema: ordenar sin Comparable (v1)

Si intentamos ordenar objetos que no implementan Comparable, obtenemos una excepcion:

```
class Empleado {
    private String nombre;
    public Empleado(String nombre) { this.nombre = nombre; }
    public String toString() { return nombre; }
}

Empleado[] empleados = {
    new Empleado("Patrobas"),
    new Empleado("Enepeto"),
    new Empleado("Andronico"),
    new Empleado("Filologo")
};

Arrays.sort(empleados); // RuntimeException!
```

Lanza `ClassCastException` porque Empleado no implementa `Comparable`. Java no sabe como comparar dos empleados.

Implementar compareTo() (v3)

La clase debe implementar `Comparable<T>` y definir `compareTo()`:

```
class Empleado implements Comparable<Empleado> {
    private String nombre;
    private int edad;

    @Override
    public int compareTo(Empleado o) { // POR EDAD
        return edad - o.edad;
```

```
    }
}
```

Retorno de compareTo()	Significado
Negativo (< 0)	`this` va antes que `o`
Cero (== 0)	Son iguales en orden
Positivo (> 0)	`this` va despues que `o`

Delegar comparacion a otro Comparable (v4)

Para ordenar por nombre, delegamos a `String.compareTo()`:

```
@Override
public int compareTo(Empleado o) { // POR NOMBRE
    return nombre.compareTo(o.nombre);
}

Arrays.sort(empleados);
// Andronico, Enepeto, Filologo, Patrobas
```

Limitacion: Comparable define **un unico** criterio de ordenamiento. Si necesitas ordenar por edad, luego por sueldo, luego por nombre... necesitas `Comparator`.

5. Comparator: ordenamiento personalizado

La interfaz `Comparator<T>` permite definir **multiples criterios** de ordenamiento sin modificar la clase original. Se pasa como segundo argumento a `Arrays.sort()`.

Aspecto	Comparable	Comparator
Paquete	`java.lang`	`java.util`
Metodo	`compareTo(T o)`	`compare(T o1, T o2)`
Quien lo implementa	La propia clase	Una clase externa
Criterios	Uno solo	Multiples
Uso	`Arrays.sort(arr)`	`Arrays.sort(arr, comp)`

Static Nested Class (v3)

Los Comparators como clases estaticas anidadas dentro de Empleado:

```
class Empleado {
    private String nombre;
    private int edad;
    private double sueldo;

    static class ComparatorEdad implements Comparator<Empleado> {
        @Override
        public int compare(Empleado emp1, Empleado emp2) {
            return emp1.edad - emp2.edad;
        }
    }

    static class ComparatorNombre implements Comparator<Empleado> {
        @Override
        public int compare(Empleado emp1, Empleado emp2) {
            return emp1.nombre.compareTo(emp2.nombre);
```

```
        }
    }
}

Comparator<Empleado> compEdad = new Empleado.ComparatorEdad();
Arrays.sort(empleados, compEdad);
```

Clases anonimas (v5)

Sin necesidad de crear clases con nombre:

```
Comparator<Empleado> compEdad = new Comparator<>() {
    @Override
    public int compare(Empleado emp1, Empleado emp2) {
        return emp1.edad - emp2.edad;
    }
};

Arrays.sort(empleados, compEdad);
```

Expresiones Lambda (v7)

Desde Java 8, como `Comparator` es una interfaz funcional (un solo metodo abstracto), podemos usar lambdas:

```
// Por edad
Arrays.sort(empleados, (emp1, emp2) -> emp1.edad - emp2.edad);

// Por sueldo
Arrays.sort(empleados, (x, y) -> (int)(x.sueldo - y.sueldo));

// Por nombre
Arrays.sort(empleados, (p1, p2) -> p1.nombre.compareTo(p2.nombre));
```

> Los nombres de los parametros son libres. No importa si se llaman `emp1, emp2` o `x, y` o `pato1, pato2`.

Composicion funcional con Method References (v8)

La forma mas moderna y poderosa: encadenar multiples criterios con `Comparator.comparingInt()`, `thenComparing()` y method references:

```
Comparator<Empleado> comp =
    Comparator.comparingInt(Empleado::getEdad)
    .thenComparingDouble(Empleado::getSueldo)
    .thenComparing(Empleado::getNombre)
    .reversed();

Arrays.sort(empleados, comp);
```

> **Regla:** Para usar method references (`Empleado::getEdad`), los atributos deben ser **privados con getters**. Es la forma mas limpia y encapsulada.

Resumen de la evolucion de Comparator:

Version	Tecnica	Lineas aprox.
v0-v2	Inner class	~10 por comparator
v3	Static nested class	~8 por comparator
v4	Local class	~6 por comparator
v5-v6	Clase anonima	~5 por comparator
v7	Lambda	**1 linea**
v8	Method reference + composicion	**1 linea compuesta**

Resumen de conceptos clave

Concepto	Descripcion
`enum`	Tipo especial para conjuntos fijos de constantes. Puede tener constructores, atributos y metodos
Inner Class	Clase anidada no estatica. Requiere instancia del outer: `new Outer().new Inner()`
Static Nested	Clase anidada estatica. No requiere instancia: `new Outer.Inner()`
Local Class	Clase definida dentro de un metodo. Alcance limitado al metodo
Anonymous Class	Clase sin nombre creada en linea. Implementa interface o extiende clase
Singleton	Patron que garantiza una sola instancia. Constructor privado + getInstance()
`Comparable`	Orden natural. La clase implementa `compareTo()`. Un criterio
`Comparator`	Orden personalizado externo. Implementa `compare()`. Multiples criterios
Lambda	Sintaxis corta para interfaces funcionales: `(a, b) -> a - b`
Method Reference	Referencia a un metodo existente: `Clase::metodo`

Progresion de los ejercicios del dia

Proyecto	Paquete	Version	Tema
enums	`com.curso`	v00	Simulacion manual de enum (pre-Java 5)
enums	`com.curso`	v0	Declaracion basica de enum
enums	`com.curso`	v1	Enums con switch
enums	`com.curso`	v2	Metodos values(), ordinal(), name()
enums	`com.curso`	v4	Constructor y propiedades
enums	`com.curso`	v5	Metodos abstractos y polimorfismo
nestedClass	`com.inner`	v0	Inner class (no estatica)
nestedClass	`com.statica`	v0	Static nested class
nestedClass	`com.local`	v0	Local class (dentro de metodo)
nestedClass	`com.anomima`	v0-v2	Anonymous class (interface, abstract, concreta)
singleton	`com.curso`	v0	Singleton lazy initialization
singleton	`com.curso`	v1	Singleton eager initialization
comparable	`com.curso`	v0-v1	Problema: sort sin Comparable
comparable	`com.curso`	v2-v4	Implementar compareTo() por edad y nombre
comparator	`com.curso`	v0-v3	Comparator: inner y static nested class
comparator	`com.curso`	v4-v6	Comparator: local y clases anonimas
comparator	`com.curso`	v7	Lambdas
comparator	`com.curso`	v8	Method references y composicion funcional