Guia de Repaso

Interfaces, Strategy, Interfaces Funcionales y Builder

Semana 02 - Miercoles 18 de Febrero de 2026
Academia Java - Ciudad de Mexico

Temas del dia:

1. Interfaces en Java: herencia de interfaces
2. Patron Strategy (Has-A): composicion sobre herencia
3. Interfaces Funcionales: lambdas y API funcional
4. Patron Builder: creacion de objetos complejos

Instructor: Miguel Rugerio

1. Interfaces en Java

Una **interface** en Java es un contrato que define metodos abstractos que las clases deben implementar. A diferencia de las clases, las interfaces permiten **herencia multiple**.

Herencia de interfaces (v0)

Las interfaces pueden extender otras interfaces usando `extends`, creando jerarquias de contratos:

```
interface A {}

interface B extends A {}

public class Principal {
    public static void main(String[] args) {
        // B hereda todo el contrato de A
        // Una clase que implemente B, debe implementar
        // los metodos de A y de B
    }
}
```

> **Clave:** Una interface puede extender **multiples** interfaces: `interface C extends A, B {}`. Esto es herencia multiple de tipos, algo que Java no permite con clases.

Diferencias: Interface vs Clase Abstracta

Aspecto	Interface	Clase Abstracta
Herencia	Multiple (`extends A, B`)	Simple (solo una)
Atributos	Solo `public static final`	Cualquier tipo
Constructores	No tiene	Si tiene
Metodos	abstract, default, static	abstract y concretos
Implementacion	`implements`	`extends`

> **Regla:** Usa interfaces para definir **que** puede hacer un objeto (contrato). Usa clases abstractas para definir **que es** un objeto (identidad comun).

2. Patron Strategy (Has-A)

El patron **Strategy** encapsula algoritmos en interfaces separadas, permitiendo que los objetos cambien su comportamiento **en tiempo de ejecucion** mediante composicion ("Has-A") en lugar de herencia rigida.

El problema: herencia rigida (v0)

Con herencia, el comportamiento `volar()` esta fijo en la clase padre:

```
public abstract class Pato {
    private String name;

    public Pato(String name) {
        this.name = name;
    }

    abstract void display();

    void volar() {
        System.out.println("Pato volar");
    }
}
```

```
public class PatoDummy extends Pato {
    PatoDummy(String nombre) { super(nombre); }

    void display() {
        System.out.println("Pato Dummy");
    }
}

// PatoDummy vuela igual que PatoSilvestre!
// No podemos diferenciar el comportamiento sin romper herencia
```

> **Problema:** PatoDummy (un pato de hule) NO deberia volar, pero hereda `volar()` de la clase padre. Sobreescribir en cada subclase no escala.

Solucion: extraer el comportamiento a una interface (v1)

Definimos una interface para el comportamiento variable:

```
public interface ComportamientoVolar {
    void ejecutaVuelo();
}
```

Creamos implementaciones concretas (estrategias):

```
public class VolarSi implements ComportamientoVolar {
    @Override
    public void ejecutaVuelo() {
        System.out.println("Si Volar");
    }
}

public class VolarNo implements ComportamientoVolar {
    @Override
    public void ejecutaVuelo() {
        System.out.println("No Volar");
    }
}

public class VolarAleatorio implements ComportamientoVolar {
    @Override
    public void ejecutaVuelo() {
        System.out.println("Si quiero Vuelo");
    }
}
```

Composicion Has-A en la clase Pato (v1)

El Pato ahora **tiene** un comportamiento de volar, en lugar de **ser** un volador:

```
public abstract class Pato {
    String name;
    ComportamientoVolar cv; // Has-A - Interface

    public Pato(String name) {
        this.name = name;
    }

    abstract void display();

    void volar() {
        cv.ejecutaVuelo(); // Delega al comportamiento
    }
}
```

```
// Uso: asignar comportamiento desde afuera
Pato pato1 = new PatoDummy("Yellow");
```

```
pato1.cv = cv2;  // No Volar
pato1.volar();

Pato pato2 = new PatoSilvestre("Lucas");
pato2.cv = cv1;  // Si Volar
pato2.volar();
```

Cambio de comportamiento en tiempo de ejecucion (v2)

Un mismo pato puede cambiar de estrategia dinamicamente:

```
Pato pato = new PatoSilvestre("Donald");

pato.cv = cv3;  // Aleatorio Volar
pato.volar();

pato.cv = cv2;  // No Volar
pato.volar();

pato.cv = cv1;  // Si Volar
pato.volar();
```

> **Ventaja:** El mismo objeto cambia su comportamiento sin necesidad de crear subclases nuevas.

Encapsulacion completa con getters/setters (v3)

Version final con atributos privados y comportamiento por defecto en el constructor:

```
public abstract class Pato {
    private String name;
    private ComportamientoVolar cv; // Has-A

    public Pato(String name) {
        this.name = name;
        cv = new VolarSi(); // Comportamiento por defecto
    }

    abstract void display();

    void volar() { cv.ejecutaVuelo(); }

    public void setCv(ComportamientoVolar cv) { this.cv = cv; }
    public String getName() { return name; }
}
```

```
public class PatoDummy extends Pato {
    PatoDummy(String nombre) {
```

```
        super(nombre);
        setCv(new VolarNo()); // Sobreescribe el default
    }

    void display() {
        System.out.println("Pato Dummy " + getName());
    }
}
```

```
// Uso final
Pato pato3 = new PatoSilvestre("Donald");
pato3.volar();                          // Si Volar (default)

pato3.setCv(new VolarAleatorio());
pato3.volar();                          // Si quiero Vuelo

pato3.setCv(new VolarNo());
pato3.volar();                          // No Volar
```

Resumen del patron Strategy

Aspecto	Herencia (v0)	Strategy (v3)
Comportamiento	Fijo en la clase padre	Intercambiable en runtime
Principio	IS-A (es un)	HAS-A (tiene un)
Agregar comportamiento	Nueva subclase	Nueva implementacion de interface
Flexibilidad	Baja	Alta
Acoplamiento	Alto	Bajo

> **Principio de diseno:** "Favorece la **composicion** sobre la herencia" - Gang of Four. El patron Strategy es la aplicacion directa de este principio.

3. Interfaces Funcionales

Una **interface funcional** es una interface que tiene exactamente **un solo metodo abstracto**. Esto permite usarla con expresiones **lambda** desde Java 8.

Interface tradicional con clases concretas (v0)

Forma clasica: una interface y multiples clases que la implementan:

```
public interface Operacion {
    int ejecuta(int x, int y);
}
```

```
public class Suma implements Operacion {
    @Override
    public int ejecuta(int x, int y) {
        return x + y;
    }
}

public class Resta implements Operacion {
    @Override
    public int ejecuta(int x, int y) {
        return x - y;
    }
}

// Division, Multiplicacion, Potencia... (una clase por operacion)
```

```
List<Operacion> operaciones = new ArrayList<>();
operaciones.add(new Suma());
operaciones.add(new Division());
operaciones.add(new Multiplicacion());
operaciones.add(new Resta());
operaciones.add(new Potencia());

for (Operacion o : operaciones) {
    System.out.println(o.getClass().getSimpleName());
    System.out.println(o.ejecuta(8, 4));
}
```

Problema: Se necesitan **5 clases separadas** solo para implementar operaciones simples. Mucho boilerplate.

@FunctionalInterface con lambdas (v1)

Con la anotacion `@FunctionalInterface`, podemos usar lambdas en lugar de clases:

```
@FunctionalInterface
public interface Operacion {
    int ejecuta(int x, int y);
    // Un solo metodo abstracto
    // Puede tener n metodos static y default
}
```

```
List<Operacion> operaciones = new ArrayList<>();

operaciones.add((y, z) -> y + z);          // Suma
operaciones.add((int1, int2) -> int1 / int2); // Division
operaciones.add((pato1, pato2) -> pato1 * pato2); // Multiplicacion
operaciones.add((x, y) -> x - y);          // Resta
operaciones.add((data1, data2) -> (int) Math.pow(data1, data2)); // Potencia
```

Resultado: De 5 clases separadas pasamos a 5 lineas de lambdas. Los nombres de los parametros son libres.

BiFunction del API de Java (v2)

En lugar de crear nuestra propia interface funcional, podemos usar `BiFunction<T,U,R>` del paquete `java.util.function`:

```
import java.util.function.BiFunction;

BiFunction<Integer, Integer, Integer> biFunSuma = (y, z) -> y + z;
BiFunction<Integer, Integer, Integer> biFunMulti = (pato1, pato2) -> pato1 * pato2;

List<BiFunction<Integer, Integer, Integer>> operaciones = new ArrayList<>();
operaciones.add(biFunSuma);
operaciones.add((int1, int2) -> int1 / int2);

for (BiFunction<Integer, Integer, Integer> o : operaciones) {
    System.out.println(o.apply(9, 3));  // Metodo: apply()
}
```

> **Nota:** `BiFunction<T,U,R>` recibe dos parametros de tipo T y U, y retorna R. El metodo se llama `apply()`.

BinaryOperator: cuando T, U y R son iguales (v3)

Si los dos parametros y el retorno son del mismo tipo, usamos `BinaryOperator<T>`:

```
import java.util.function.BinaryOperator;

BinaryOperator<Integer> biFunSuma = (y, z) -> y + z;
BinaryOperator<Integer> biFunMulti = (pato1, pato2) -> pato1 * pato2;

List<BinaryOperator<Integer>> operaciones = new ArrayList<>();
// ... mismas lambdas

for (BinaryOperator<Integer> o : operaciones) {
    System.out.println(o.apply(9, 3));
}
```

> **Simplificacion:** `BinaryOperator<Integer>` es mas limpio que `BiFunction<Integer,Integer,Integer>` cuando los 3 tipos son iguales.

IntBinaryOperator: evitando autoboxing (v4)

Para tipos primitivos `int`, Java ofrece `IntBinaryOperator` que evita el autoboxing:

```
import java.util.function.IntBinaryOperator;

IntBinaryOperator biFunSuma = (y, z) -> y + z;
IntBinaryOperator biFunMulti = (pato1, pato2) -> pato1 * pato2;

List<IntBinaryOperator> operaciones = new ArrayList<>();
// ... mismas lambdas

for (IntBinaryOperator o : operaciones) {
    System.out.println(o.applyAsInt(9, 3));  // applyAsInt()
}
```

Evolucion de las interfaces funcionales

Version	Tipo	Metodo	Ventaja
v0	Interface propia + clases	`ejecuta()`	Explicito, claro

v1	`@FunctionalInterface` + lambda	`ejecuta()`	Sin clases extra
v2	`BiFunction<Integer,Integer,Integer>`	`apply()`	API estandar de Java
v3	`BinaryOperator<Integer>`	`apply()`	Mas conciso
v4	`IntBinaryOperator`	`applyAsInt()`	Sin autoboxing

4. Patron Builder

El patron **Builder** separa la construccion de un objeto complejo de su representacion, permitiendo crear objetos paso a paso con una **interfaz fluida** (method chaining).

Concepto: encadenamiento de metodos (v0)

Java ya usa este patron en clases como `StringBuilder`:

```
// String: crea un nuevo objeto por cada concat (4 objetos)
String resultado1 = ""
        .concat("Hola")
        .concat(" ")
        .concat("mundo");

// StringBuilder: un solo objeto mutable
String resultado2 = new StringBuilder()
        .append("Hola")
        .append(" ")
        .append("mundo")    // 1 solo objeto
        .toString();
```

> **Clave:** Cada metodo `append()` retorna `this`, permitiendo encadenar llamadas. Este es el principio del patron Builder.

El problema: constructores telescopicos

Cuando una clase tiene muchos atributos, el constructor se vuelve inmanejable:

```
public class Task {
    private int id;
    private String name;
    private LocalDate fechaInicio;
    private LocalDate fechaTermino;
    private String responsable;
    private String asignar;
    private boolean completada;
    private double calificacion;
    private String categoria;

    // Constructor con 9 parametros!
    public Task(int id, String name, LocalDate fechaInicio,
```

```
            LocalDate fechaTermino, String responsable,
            String asignar, boolean completada,
            double calificacion, String categoria) {
        // ... asignar todos
    }
}
```

Problema: ¿Que pasa si solo necesitas id y name? Debes pasar `null` en los demas o crear multiples constructores. No escala.

Builder como clase separada (v1)

Creamos una clase `TaskBuilder` que construye el objeto paso a paso:

```
public class TaskBuilder {
    private int id;
    private String name;
    private LocalDate fechaInicio;
    // ... demas atributos

    public TaskBuilder(int id) {
        this.id = id;  // Solo el campo obligatorio
    }

    public TaskBuilder setName(String name) {
        this.name = name;
        return this;  // Retorna this para encadenar
    }

    public TaskBuilder setFechaInicio(LocalDate fechaInicio) {
        this.fechaInicio = fechaInicio;
        return this;
    }

    // ... demas setters que retornan this

    Task build() {
        return new Task(id, name, fechaInicio, fechaTermino,
                responsable, asignar, completada,
                calificacion, categoria);
    }
}
```

```
// Solo con id (minimo)
Task tarea1 = new TaskBuilder(1).build();

// Con algunos campos
Task tarea2 = new TaskBuilder(2)
        .setName("Investigar POO")
```

```
        .setFechaInicio(LocalDate.now())
        .build();

// Con todos los campos, en cualquier orden
Task tarea3 = new TaskBuilder(3)
        .setCompletada(true)
        .setResponsable("Patrobas")
        .setName("Investigar POO")
        .setCategoria("OO")
        .setAsignar("Filologo")
        .setFechaInicio(LocalDate.now())
        .setCalificacion(100.0)
        .setFechaTermino(LocalDate.now())
        .build();
```

Ventaja: Solo un objeto `TaskBuilder` se crea. Los campos opcionales se configuran a gusto, en cualquier orden.

Builder como clase anidada estatica (v2)

La version optima: el Builder vive **dentro** de la clase Task como `static class`:

```
public class Task {
    private int id;
    private String name;
    // ... demas atributos

    private Task(int id, String name, ...) {
        // Constructor PRIVADO - solo el Builder puede crear
    }

    static public class TaskBuilder {
        private int id;
        private String name;
        // ... mismos atributos

        public TaskBuilder(int id) { this.id = id; }

        public TaskBuilder setName(String name) {
            this.name = name;
            return this;
        }
        // ... demas setters

        Task build() {
            return new Task(id, name, fechaInicio,
                    fechaTermino, responsable, asignar,
                    completada, calificacion, categoria);
        }
```

```
    }
}
```

```
// Uso: Task.TaskBuilder
Task tarea1 = new Task.TaskBuilder(1).build();

Task tarea2 = new Task.TaskBuilder(2)
        .setName("Investigar POO")
        .setFechaInicio(LocalDate.now())
        .build();

Task tarea3 = new Task.TaskBuilder(3)
        .setCompletada(true)
        .setResponsable("Patrobas")
        .setName("Investigar POO")
        .setCategoria("OO")
        .setAsignar("Filologo")
        .setFechaInicio(LocalDate.now())
        .setCalificacion(100.0)
        .setFechaTermino(LocalDate.now())
        .build();
```

Clave: El constructor de `Task` ahora es **privado**. Solo el Builder puede crear instancias. Esto garantiza que todos los objetos Task se crean de forma controlada.

Evolucion del patron Builder

Version	Tecnica	Caracteristica
v0	Method chaining (StringBuilder)	Concepto base: retornar `this`
v1	Builder como clase separada	TaskBuilder independiente de Task
v2	Builder como static nested class	Constructor privado + Builder interno

Resumen de conceptos clave

Concepto	Descripcion
Interface	Contrato que define metodos abstractos. Permite herencia multiple de tipos
Has-A	Composicion: un objeto **tiene** otro objeto (vs. IS-A: **es** un)
Patron Strategy	Encapsula algoritmos intercambiables en interfaces. Cambio en runtime
`@FunctionalInterface`	Interface con un solo metodo abstracto. Habilita lambdas
Lambda	Sintaxis corta: `(a, b) -> a + b`
`BiFunction<T,U,R>`	Interface funcional estandar: dos parametros, un retorno
`BinaryOperator<T>`	BiFunction donde T, U y R son del mismo tipo
`IntBinaryOperator`	Especializacion para `int` primitivo (sin autoboxing)
Patron Builder	Crea objetos complejos paso a paso con interfaz fluida
Method Chaining	Retornar `this` para encadenar llamadas
Static Nested Class	Clase anidada estatica: ideal para Builder dentro de la clase objetivo

Progresion de los ejercicios del dia

Proyecto	Paquete	Version	Tema
Interface	`com.curso.v0`	v0	Herencia de interfaces
hasAstrategy	`com.curso.v0`	v0	Herencia rigida (problema)
hasAstrategy	`com.curso.v1`	v1	Interface + Has-A
hasAstrategy	`com.curso.v2`	v2	Cambio dinamico de comportamiento
hasAstrategy	`com.curso.v3`	v3	Encapsulacion con getters/setters

interfaceFunctional	`com.curso.v0`	v0	Interface + clases concretas
interfaceFunctional	`com.curso.v1`	v1	@FunctionalInterface + lambdas
interfaceFunctional	`com.curso.v2`	v2	BiFunction
interfaceFunctional	`com.curso.v3`	v3	BinaryOperator
interfaceFunctional	`com.curso.v4`	v4	IntBinaryOperator
patternBuilder	`com.curso.v0`	v0	Method chaining (StringBuilder)
patternBuilder	`com.curso.v1`	v1	Builder como clase separada
patternBuilder	`com.curso.v2`	v2	Builder como static nested class