Guia de Repaso

Clases Abstractas, Factory Patterns y Threads

Semana 02 - Jueves 19 de Febrero de 2026
Academia Java - Ciudad de Mexico

Temas del dia:

1. Clases Abstractas: de concretas a abstractas
2. Patron Factory Simple: encapsular la creacion
3. Patron Factory Method (Template Method)
4. Inversion de Control (IoC): hacia Spring
5. Patron Abstract Factory: familias de objetos
6. Threads basicos: concurrencia en Java

Instructor: Miguel Rugerio

1. Clases Abstractas

Una **clase abstracta** es una clase que no puede instanciarse directamente. Sirve para definir un contrato parcial: algunos metodos tienen implementacion y otros son **abstractos** (sin cuerpo), obligando a las subclases a implementarlos.

El problema: clase base concreta (v0)

Si la clase base es concreta, alguien puede instanciarla directamente sin especializarla:

```
public class Fabrica {
    void producir() {
        System.out.println("Fabrica producir....");
    }
}

public class FabricaMty extends Fabrica {
    @Override
    void producir() {
        System.out.println("Fabrica Monterrey producir....");
    }
}
```

```
// Se puede crear una Fabrica generica - no tiene sentido
Fabrica fab1 = new Fabrica();
fab1.producir();

Fabrica fab2 = new FabricaMty();
fab2.producir();
```

Problema: `new Fabrica()` crea una fabrica generica sin especializacion. No deberia permitirse instanciar la clase base directamente.

Solucion: clase abstracta (v1)

Al hacer la clase `abstract`, el compilador impide instanciarla directamente:

```
public abstract class Fabrica {
    abstract void producir();
}
```

> **Regla:** Una clase con al menos un metodo abstracto **debe** ser abstracta. Las subclases estan **obligadas** a implementar los metodos abstractos.

Template Method con abstract (v2)

Combinamos `abstract` con `final` para crear un template method: el algoritmo esta fijo, pero los pasos los define cada subclase:

```
public abstract class Fabrica {

    abstract void producir(); // cada subclase decide como

    final void entregarProducto() { // algoritmo FIJO
        System.out.println("Inicio Produccion");
        producir(); // paso que varia
        System.out.println("Fin Produccion");
    }
}
```

```
Fabrica fab1 = new FabricaMty();
fab1.entregarProducto();
// Inicio Produccion
// Fabrica Monterrey producir....
// Fin Produccion
```

> **Clave:** `final` en `entregarProducto()` impide que las subclases sobreescriban el algoritmo. Solo pueden variar el paso `producir()`.

Metodo concreto por defecto (v3)

Una clase abstracta puede tener metodos concretos (con cuerpo). Si una subclase no sobreescribe, usa el comportamiento por defecto:

```
public abstract class Fabrica {

    void producir() { // concreto: comportamiento por defecto
        System.out.println("PRODUCIR!!!!");
    }

    final void entregarProducto() {
        System.out.println("Inicio Produccion");
        producir();
        System.out.println("Fin Produccion");
    }
}
```

```
Fabrica fab3 = new FabricaTlaxcala(); // No sobreescribio producir()
fab3.entregarProducto();
// Inicio Produccion
// PRODUCIR!!!!          <-- usa el default
// Fin Produccion
```

Evolucion de las clases abstractas

Version	Fabrica	Caracteristica
v0	Clase concreta	Se puede instanciar (problema)
v1	`abstract` + metodo abstracto	Obliga a implementar `producir()`
v2	Template Method con `final`	Algoritmo fijo + pasos variables
v3	Metodo concreto por defecto	Subclases pueden o no sobreescribir

2. Patron Factory Simple

El **Factory Simple** encapsula la logica de creacion de objetos en una clase separada. El cliente no sabe que clase concreta se crea, solo pide por tipo.

Pizza como clase abstracta

```
abstract public class Pizza {
    String name;
    String dough;
    String sauce;
    List<String> toppings = new ArrayList<>();

    public void prepare() { System.out.println("Preparing " + name); }
    public void bake()    { System.out.println("Baking " + name); }
    public void cut()     { System.out.println("Cutting " + name); }
    public void box()     { System.out.println("Boxing " + name); }
}
```

La fabrica simple

```
public class SimplePizzaFactory {

    public Pizza createPizza(String type) {
        Pizza pizza = null;

        if (type.equals("cheese")) {
            pizza = new CheesePizza();
        } else if (type.equals("pepperoni")) {
            pizza = new PepperoniPizza();
        } else if (type.equals("clam")) {
            pizza = new ClamPizza();
        } else if (type.equals("veggie")) {
            pizza = new VeggiePizza();
        }
        return pizza;
    }
}
```

El PizzaStore usa la fabrica (HAS-A)

```
public class PizzaStore {
    SimplePizzaFactory factory; // HAS-A

    public PizzaStore(SimplePizzaFactory factory) {
        this.factory = factory;
    }

    public Pizza orderPizza(String type) {
        Pizza pizza;
        pizza = factory.createPizza(type); // DELEGACION
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
    }
}
```

Ventaja: Si se agrega un nuevo tipo de pizza, solo se modifica la fabrica. El `PizzaStore` no cambia. Esto es el principio **Open/Closed**.

Limitacion: Con una sola fabrica, todas las tiendas crean las mismas pizzas. No podemos tener pizzas estilo NY y estilo Chicago al mismo tiempo.

3. Patron Factory Method (Template Method)

El **Factory Method** mueve la decision de creacion a las **subclases**. La clase padre define el algoritmo (template method), y cada subclase implementa el metodo fabrica.

El anti-patron: DependentPizzaStore

Sin el patron, una sola clase depende de TODAS las clases concretas:

```
public class DependentPizzaStore {
    // MAL DISENO
    public Pizza createPizza(String style, String type) {
        Pizza pizza = null;
        if (style.equals("NY")) {
            if (type.equals("cheese")) {
                pizza = new NYStyleCheesePizza();
            } else if (type.equals("veggie")) {
                pizza = new NYStyleVeggiePizza();
            }
            // ... mas tipos
        } else if (style.equals("Chicago")) {
            if (type.equals("cheese")) {
                pizza = new ChicagoStyleCheesePizza();
            }
            // ... mas tipos
        }
        return pizza;
    }
}
```

Problema: Esta clase depende de **todas** las implementaciones concretas. Cada nueva region o tipo de pizza requiere modificar esta clase. Viola el principio Open/Closed.

Solucion: Factory Method

```
public abstract class PizzaStore {

    // The Factory Method Pattern
    abstract Pizza createPizza(String item); // subclase decide

    final public Pizza orderPizza(String type) {
        Pizza pizza = createPizza(type); // <== Subclass decide
        System.out.println("--- Making a " + pizza.getName() + " ---");
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
    }
}
```

Cada region implementa su fabrica

```
public class NYPizzaStore extends PizzaStore {

    @Override
    Pizza createPizza(String item) {
        if (item.equals("cheese")) {
            return new NYStyleCheesePizza();
        } else if (item.equals("veggie")) {
            return new NYStyleVeggiePizza();
        } else if (item.equals("clam")) {
            return new NYStyleClamPizza();
        } else if (item.equals("pepperoni")) {
            return new NYStylePepperoniPizza();
        } else return null;
    }
}
```

Principio: "Depende de abstracciones, no de clases concretas" - **Dependency Inversion Principle**. El `PizzaStore` solo conoce `Pizza` (abstracta), no las concretas.

Aspecto	Factory Simple	Factory Method
Creacion	Una clase fabrica separada	Metodo abstracto en la clase padre
Quien decide	La fabrica	Las subclases
Flexibilidad	Una fabrica por aplicacion	Una subclase por variante
Escalabilidad	Limitada	Alta: nueva region = nueva subclase

4. Inversion de Control (IoC)

La **Inversion de Control** es el principio donde el control de la creacion de objetos se invierte: en lugar de que la clase cree sus dependencias, alguien externo se las **inyecta**.

IoC v1: Inyeccion manual con setter

El `PizzaStore` ya no crea la pizza internamente. Se le inyecta desde afuera via `setPizza()`:

```
public abstract class PizzaStore {
    private Pizza pizza; // se inyecta desde afuera

    final public Pizza orderPizza() {
        Pizza pizza = createPizza();
        System.out.println("--- Making a " + pizza.getName() + " ---");
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
    }

    public void setPizza(Pizza pizza) { this.pizza = pizza; }
}
```

```
// Uso: inyeccion manual
PizzaStore nyStore = new NYPizzaStore();

nyStore.setPizza(new NYStyleCheesePizza()); // Inyeccion via setter
Pizza pizza = nyStore.orderPizza();
```

> **Clave:** El `PizzaStore` ya no sabe que pizza concreta va a procesar. La responsabilidad de creacion se movio afuera.

IoC v2: Inyector centralizado (simulando Spring)

Creamos una clase `Inyector` que decide que pizza inyectar segun el tipo de tienda:

```
public class Inyector {

    public static void inyecta(PizzaStore store, String item) {

        if (store instanceof NYPizzaStore) {
            if (item.equals("cheese")) {
                store.setPizza(new NYStyleCheesePizza());
            } else if (item.equals("clam")) {
                store.setPizza(new NYStyleClamPizza());
            }
        } else if (store instanceof ChicagoPizzaStore) {
            if (item.equals("cheese")) {
                store.setPizza(new ChicagoStyleCheesePizza());
            } else if (item.equals("clam")) {
                store.setPizza(new ChicagoStyleClamPizza());
            }
        }
    }
}
```

```
// Uso: el Inyector resuelve la dependencia
PizzaStore nyStore = new NYPizzaStore();

Inyector.inyecta(nyStore, "cheese"); // El inyector decide
Pizza pizza = nyStore.orderPizza();
```

Nota: Este es el concepto detras de **Spring Framework**. Spring hace esto automaticamente con anotaciones como `@Autowired` y `@Component`, sin necesidad de escribir el `Inyector` manualmente.

Version	Tecnica	Quien crea el objeto
Factory Method	Metodo abstracto	La subclase (internamente)
IoC v1	Setter injection manual	El cliente (main)
IoC v2	Inyector centralizado	El inyector (simulando Spring)
Spring real	`@Autowired`	El contenedor IoC de Spring

5. Patron Abstract Factory

El **Abstract Factory** provee una interfaz para crear **familias de objetos relacionados** sin especificar sus clases concretas. Cada fabrica crea un conjunto completo y consistente de productos.

La interfaz de la fabrica de ingredientes

```
public interface PizzaIngredientFactory {

    public Dough createDough();
    public Sauce createSauce();
    public Cheese createCheese();
    public Veggies[] createVeggies();
    public Pepperoni createPepperoni();
    public Clams createClam();
}
```

Implementacion por region: NY

```
public class NYPizzaIngredientFactory implements PizzaIngredientFactory {

    public Dough createDough() {
        return new ThinCrustDough();     // masa delgada
    }
    public Sauce createSauce() {
        return new MarinaraSauce();      // salsa marinara
    }
    public Cheese createCheese() {
        return new ReggianoCheese();     // queso reggiano
    }
    public Veggies[] createVeggies() {
        return new Veggies[] {
            new Garlic(), new Onion(), new Mushroom(), new RedPepper()
        };
    }
    public Pepperoni createPepperoni() {
        return new SlicedPepperoni();
    }
    public Clams createClam() {
        return new FreshClams();         // almejas frescas (cerca del mar)
    }
}
```

La pizza usa la fabrica (HAS-A)

```
public class CheesePizza extends Pizza {
    PizzaIngredientFactory ingredientFactory; // HAS-A

    public CheesePizza(PizzaIngredientFactory ingredientFactory) {
        this.ingredientFactory = ingredientFactory;
    }

    public void prepare() {
        System.out.println("Preparing " + name);
        dough  = ingredientFactory.createDough();  // la fabrica decide
        sauce  = ingredientFactory.createSauce();
        cheese = ingredientFactory.createCheese();
    }
}
```

Clave: La `CheesePizza` no sabe si tiene masa delgada o gruesa. Solo le pide a su fabrica que cree los ingredientes. Una misma pizza con diferente fabrica = diferentes ingredientes.

Principio: Cada fabrica garantiza que los ingredientes sean **consistentes entre si**. NY siempre usa masa delgada + salsa marinara + almejas frescas.

Patron	Que crea	Como decide
Factory Simple	Un producto	Un metodo con if/else
Factory Method	Un producto	Subclases sobreescriben metodo
Abstract Factory	**Familia de productos**	Interfaz con multiples metodos create

6. Threads Basicos

Un **hilo (thread)** es una unidad de ejecucion dentro de un programa. Todo programa Java tiene al menos un hilo: el hilo `main`. Cuando creas hilos adicionales, tu programa puede hacer **varias cosas al mismo tiempo**.

extends Thread (v0)

La forma mas directa: extender la clase `Thread` y sobreescribir `run()`:

```
public class MiHilo extends Thread {

    private int contador;

    public MiHilo(String nombre, int contador) {
        super(nombre);
        this.contador = contador;
    }

    @Override
    public void run() {
        for (int i = 1; i <= contador; i++) {
            System.out.println(getName() + " -> " + i);
        }
        System.out.println(getName() + " termino!");
    }
}
```

```
MiHilo hilo1 = new MiHilo("Hilo-A", 5);
MiHilo hilo2 = new MiHilo("Hilo-B", 5);

hilo1.start(); // crea un hilo nuevo
hilo2.start(); // crea otro hilo nuevo

// hilo1.run(); // ERROR COMUN: esto NO crea un hilo nuevo
```

Error comun: Llamar `run()` directamente NO crea un hilo nuevo, se ejecuta en el hilo actual. Siempre usar `start()`.

implements Runnable (v1)

La forma **preferida**: separar la tarea del mecanismo de ejecucion. Composicion sobre herencia:

```
public class MiTarea implements Runnable {
    private String nombre;
    private int contador;

    public MiTarea(String nombre, int contador) {
        this.nombre = nombre;
        this.contador = contador;
    }

    @Override
    public void run() {
        for (int i = 1; i <= contador; i++) {
            System.out.println(nombre + " -> " + i);
        }
    }
}

// Uso: la tarea se pasa al Thread
Thread hilo1 = new Thread(new MiTarea("Tarea-A", 5));
hilo1.start();
```

Lambda (v2)

`Runnable` es una interfaz funcional, por lo que podemos usar lambdas:

```
Thread hilo1 = new Thread(() -> {
    for (int i = 1; i <= 5; i++) {
        System.out.println("Lambda-A -> " + i);
    }
});
hilo1.start();

// Forma ultra compacta
new Thread(() -> System.out.println("Hilo de una sola linea!")).start();
```

Forma	Ventaja	Desventaja
`extends Thread`	Simple, directa	No puedes extender otra clase
`implements Runnable`	Flexible, reutilizable	Necesitas clase extra
Lambda	Compacta, moderna	Solo para tareas simples

sleep(): simular tareas largas (v3)

`Thread.sleep(ms)` pausa el hilo actual. Mientras uno duerme, otros hilos se ejecutan:

```
Thread descarga1 = new Thread(() -> descargarArchivo("foto.jpg", 3));
Thread descarga2 = new Thread(() -> descargarArchivo("video.mp4", 5));
Thread descarga3 = new Thread(() -> descargarArchivo("musica.mp3", 2));

descarga1.start();
descarga2.start();
descarga3.start();

static void descargarArchivo(String nombre, int segundos) {
    System.out.println("Iniciando descarga de " + nombre + "...");
    for (int i = 1; i <= segundos; i++) {
        try {
            Thread.sleep(1000); // pausa 1 segundo
        } catch (InterruptedException e) {
            return;
        }
        System.out.println(nombre + " -> " + i + "/" + segundos + " seg");
    }
    System.out.println(nombre + " descargado!");
}
```

Clave: Las tres descargas corren en **paralelo**. El tiempo total es ~5 seg (el maximo), no 3+5+2=10 seg.

join(): esperar a que terminen (v4)

`join()` hace que el hilo actual **espere** a que otro hilo termine antes de continuar:

```
Thread hilo1 = new Thread(() -> {
    System.out.println("Hilo-1: Consultando BD...");
    pausar(3000);
    System.out.println("Hilo-1: Consulta terminada!");
});

Thread hilo2 = new Thread(() -> {
    System.out.println("Hilo-2: Llamando API...");
    pausar(2000);
    System.out.println("Hilo-2: Respuesta recibida!");
});

Thread hilo3 = new Thread(() -> {
    System.out.println("Hilo-3: Procesando archivo...");
    pausar(1000);
    System.out.println("Hilo-3: Archivo procesado!");
});

long inicio = System.currentTimeMillis();

hilo1.start(); hilo2.start(); hilo3.start();

hilo1.join(); hilo2.join(); hilo3.join(); // esperar

long duracion = System.currentTimeMillis() - inicio;
System.out.println("Total: " + duracion + " ms");
// ~3000 ms (no 6000 ms)
```

Nota: Sin hilos tardaria ~6000 ms (3+2+1). Con hilos tarda ~3000 ms (el maximo de los tres). `join()` es esencial cuando necesitas los resultados antes de continuar.

Race Condition: recurso compartido (v5)

Cuando dos hilos modifican la misma variable, el resultado es **impredecible**:

```
public class Contador {
    private int valor = 0;

    // NO es thread-safe
    // valor++ internamente son 3 operaciones: leer, sumar, escribir
    public void incrementar() {
        valor++;
    }

    public int getValor() { return valor; }
}
```

```
Contador contador = new Contador(); // recurso COMPARTIDO

Thread hilo1 = new Thread(() -> {
    for (int i = 0; i < 10_000; i++) contador.incrementar();
});
Thread hilo2 = new Thread(() -> {
    for (int i = 0; i < 10_000; i++) contador.incrementar();
});

hilo1.start(); hilo2.start();
hilo1.join();  hilo2.join();

System.out.println("Esperado: 20000");
System.out.println("Real:     " + contador.getValor()); // ~13000!
```

Problema: `valor++` parece atomico pero son 3 pasos: LEER, SUMAR, ESCRIBIR. Si dos hilos leen el mismo valor antes de escribir, uno pisa al otro. Esto se llama **race condition**.

Solucion (siguiente tema): Usar `synchronized` para que solo un hilo a la vez ejecute el metodo: `public synchronized void incrementar()`

Resumen de conceptos clave

Concepto	Descripcion
Clase abstracta	Clase que no se puede instanciar. Define metodos abstractos que las subclases deben implementar
`abstract`	Modificador para clases y metodos sin implementacion
`final` en metodo	Impide que las subclases sobreescriban el metodo (Template Method)
Factory Simple	Clase separada que encapsula la creacion de objetos
Factory Method	Metodo abstracto: las subclases deciden que crear
Abstract Factory	Interfaz para crear familias de objetos relacionados
IoC	El control de creacion se invierte: alguien externo inyecta las dependencias
`Thread`	Clase que representa un hilo de ejecucion
`Runnable`	Interfaz funcional que define la tarea a ejecutar
`start()`	Crea un hilo nuevo y ejecuta `run()`
`sleep(ms)`	Pausa el hilo actual N milisegundos
`join()`	Espera a que un hilo termine
Race condition	Problema cuando dos hilos modifican el mismo recurso sin sincronizacion

Progresion de los ejercicios del dia

Proyecto	Paquete	Version	Tema
abstract	`como.curso.v0`	v0	Clase base concreta (problema)
abstract	`como.curso.v1`	v1	Clase abstracta
abstract	`como.curso.v2`	v2	Template Method con final
abstract	`como.curso.v3`	v3	Metodo concreto por defecto
factorySimple	`com.simple.factory`	-	Factory Simple con pizzas
factoryTemplateMethod	`com.curso.v0`	v0	Factory Method + Template Method
factoryTemplateMethodIOC	`com.curso.v0`	v0	IoC con setter injection
factoryTemplateMethodIOCv2	`com.curso.v0`	v0	Inyector centralizado (Spring)
factoryAbstract	`factory, pizza, ...`	-	Abstract Factory con ingredientes
threads	`com.curso.v0`	v0	`extends Thread`
threads	`com.curso.v1`	v1	`implements Runnable`
threads	`com.curso.v2`	v2	Lambda
threads	`com.curso.v3`	v3	`sleep()`
threads	`com.curso.v4`	v4	`join()`
threads	`com.curso.v5`	v5	Race condition