Patrones creacionales

Abstract factory: Es un patrón de diseño creacional que nos permite producir familias de objetos relacionados sin especificar sus clases concretas.

```
=====================================================
interface GUIFactory { Button createButton(); }
class WinFactory implements GUIFactory {
    public Button createButton() { return new WinButton(); }
}
interface Button { void paint(); }
class WinButton implements Button {
    public void paint() { System.out.println("Windows Button"); }
}
public class Main {
    public static void main(String[] args) {
        GUIFactory factory = new WinFactory();
        factory.createButton().paint();
    }
}
=====================================================
```

GUIFactory define un método para crear un Button, y WinFactory lo implementa para crear un WinButton.

Builder: Es un patrón de diseño creacional que nos permite construir objetos complejos paso a paso. El patrón nos permite producir distintos tipos y representaciones de un objeto empleando el mismo código de construcción.

==========

```
class Pizza {
    private String topping;
    public void setTopping(String topping) { this.topping = topping; }
    public String getDetails() { return "Pizza with " + topping; }
}
class PizzaBuilder {
    private Pizza pizza = new Pizza();
    public PizzaBuilder addTopping(String topping) {
        pizza.setTopping(topping);
        return this;
    }
    public Pizza build() { return pizza; }
}
public class Main {
    public static void main(String[] args) {
        Pizza pizza = new PizzaBuilder().addTopping("cheese").build();
        System.out.println(pizza.getDetails());
    }
}
```

==========

El PizzaBuilder construye una Pizza paso a paso.

Factory Method: Es un patrón de diseño creacional que proporciona una interfaz para crear objetos en una superclase, mientras permite a las subclases alterar el tipo de objetos que se crearán.

==========

```
interface Transport { void deliver(); }
class Truck implements Transport {
    public void deliver() { System.out.println("Deliver by Truck"); }
}
abstract class Logistics {
    public void planDelivery() {
        Transport transport = createTransport();
        transport.deliver();
    }
    protected abstract Transport createTransport();
}
class RoadLogistics extends Logistics {
    protected Transport createTransport() { return new Truck(); }
}
public class Main {
    public static void main(String[] args) {
        new RoadLogistics().planDelivery();
    }
}
```

==========

RoadLogistics decide crear un Truck usando el método createTransport.

Prototype: Es un patrón de diseño creacional que nos permite copiar objetos existentes sin que el código dependa de sus clases.

==========

```
class Circle implements Cloneable {
    public Circle clone() {
        try { return (Circle) super.clone(); }
        catch (CloneNotSupportedException e) { return null; }
    }
}
public class Main {
    public static void main(String[] args) {
        Circle c1 = new Circle();
        Circle c2 = c1.clone();
        System.out.println("Cloned Circle");
    }
}
```

==========

Circle se clona a sí mismo utilizando el método clone.

Singleton: Es un patrón de diseño creacional que nos permite asegurarnos de que una clase tenga una única instancia, a la vez que proporciona un punto de acceso global a dicha instancia.

==========

```
class Singleton {
    private static Singleton instance;
    private Singleton() {}
    public static Singleton getInstance() {
        if (instance == null) { instance = new Singleton(); }
        return instance;
    }
}
public class Main {
    public static void main(String[] args) {
        Singleton s = Singleton.getInstance();
        System.out.println("Singleton instance");
    }
}
```

==========

Singleton asegura que solo haya una instancia mediante getInstance.