**Patrones de diseño**

## **Introducción**

En este apartado, repasaremos los principios de diseño de clases Java y los principales patrones de diseño empleados en el acceso a datos, sobre todo los patrones de creación.

Un patrón de diseño es una **solución general establecida para un problema común en el desarrollo de software**. El propósito de un patrón de diseño es **buscar estrategias comunes** para aprovechar el conocimiento y la experiencia acumulada de los desarrolladores para **resolver problemas fácilmente**. También proporciona a los desarrolladores un vocabulario común en el que pueden discutir problemas y soluciones comunes. Por ejemplo, si dices que escribiste getters/setters o implementaste el patrón singleton, la mayoría de los desarrolladores entenderán la estructura de tu código sin tener que profundizar en los detalles de bajo nivel.

Los patrones que pueden resultar más interesantes para acceso a datos, son los **patrones de creación**, un tipo de patrón que **gestiona la creación de objetos dentro de una aplicación**.

Escritor escritor = new Poeta();

EL problema con la creación de objetos radica en **cómo crear y gestionar objetos en sistemas más complejos**. Por ejemplo, necesitamos saber exactamente qué tipo de objeto es Escritor, en este caso, Poeta, que se crea en tiempo de compilación. Pero, **en muchos casos no se sabe hasta tiempo de ejecución**, además de crear un solo objeto Escritor en la memoria compartido por todas las clases dentro de nuestra aplicación (patrón Singleton).

Los patrones de creación simplemente aplican un nivel de indirección a la creación de objetos al crear el objeto en alguna otra clase, en lugar de crear el objeto directamente en tu aplicación. **El nivel de indirección es un término general para resolver un problema de diseño de software dividiendo conceptualmente la tarea en múltiples niveles**.

# **01. PRINCIPIOS DE DISEÑO**

* [Introducción](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/03patronesdisenho/01designprinciples/#introducci%C3%B3n)
* [Encapsulación de Datos](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/03patronesdisenho/01designprinciples/#encapsulaci%C3%B3n-de-datos)
  + [Bloqueo de acceso directo a atributos privados](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/03patronesdisenho/01designprinciples/#bloqueo-de-acceso-directo-a-atributos-privados)
* [Creación de JavaBeans](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/03patronesdisenho/01designprinciples/#creaci%C3%B3n-de-javabeans)
* [Relación Es‐un](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/03patronesdisenho/01designprinciples/#relaci%C3%B3n-esun)
* [Relación Tiene‐un](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/03patronesdisenho/01designprinciples/#relaci%C3%B3n-tieneun)
  + [Problemas del modelo de datos empleando Es‐un y Tiene‐un](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/03patronesdisenho/01designprinciples/#problemas-del-modelo-de-datos-empleando-esun-y-tieneun)
* [Composición de Objetos](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/03patronesdisenho/01designprinciples/#composici%C3%B3n-de-objetos)

## **Introducción**

Haremos un inciso para repasar y descubrir nuevos principios de diseño, pues son un elemento clave en el diseño de estructura de clases de acceso a Datos, **patrones como Singleton son interesantes para establecer conexiones a BD**, por ejemplo.

Un **principio de diseño** es una idea establecida o mejor práctica que facilita el proceso de diseño de software. Al crear clases en Java **estos principios conducen a bases de código mejores y más manejables**. En general, seguir buenos principios de diseño lleva a:

* Código más **lógico**
* Código más **fácil de entender**
* Clases más **fáciles de reutilizar** en otras relaciones y aplicaciones
* Código más **fácil de mantener** y que se adapta más fácilmente a cambios en los requisitos de la aplicación

Un **modelo de datos es la representación de nuestros objetos y sus propiedades dentro de la aplicación y cómo se relacionan con elementos del mundo real**.

## **Encapsulación de Datos**

Un principio fundamental del diseño orientado a objetos es el concepto de encapsular datos.  
En el desarrollo de software, la encapsulación es declarar que los atributos y métodos en una clase de manera que los **métodos operen sobre los atributos, en lugar de que los usuarios de la clase accedan directamente a los atributos**. En Java, de manera general se implementan con **atributos de instancia privados que tienen métodos públicos** para recuperar o modificar los datos, comúnmente conocidos como **getters y setters**, respectivamente.

public class EjemploClase {  
 private int datos;  
  
 // Método getter  
 public int getDatos() {  
 return datos;  
 }  
  
 // Método setter  
 public void setDatos(int nuevosDatos) {  
 this.datos = nuevosDatos;  
 }  
}

Ningún elemento además de la propia clase debería tener acceso directo a sus datos.

Se dice que la clase **encapsula los datos que contiene y evita que alguien acceda directamente a ellos**. Con la encapsulación, una clase puede mantener ciertas **invariantes** sobre sus datos internos. Una invariante es una **propiedad o verdad que se mantiene incluso después de que los datos son modificados**. Por ejemplo, imaginemos que estamos diseñando una nueva clase Animal y tenemos los siguientes requisitos de diseño:

* Cada animal tiene un campo de especie no nulo y no vacío.
* Cada animal tiene un campo de edad que es mayor o igual a cero.

El objetivo al diseñar nuestra clase Animal sería asegurarnos de que nunca lleguemos a una instancia de Animal que viole una de estas propiedades. Al usar miembros de instancia privados junto con métodos getter y setter que validan los datos de entrada, podemos garantizar que estas invariantes sigan siendo verdaderas.

Por ejemplo, si definimos la clase Animal sin encapsulación:

public class Animal {  
 public String especie;  
 public int edad;  
}

Al definir la clase Animal de esta manera, es fácil crear una instancia de Animal que no cumpla las invariantes:

Animal animal = new Animal(); // La especie no debería ser nula.  
animal.edad = -80; // La edad no podría ser negativa.

L primera invariante se viola tan pronto como se crea el objeto, con la especie predeterminada a nula. Luego, el usuario establece el campo de edad en -80, ya que este campo es accesible públicamente, lo que resulta en la violación de la segunda invariante.

HAciendo los atributos privados, la clase es la única que puede modificar los datos directamente. Al definir constructores, getters y setters que cumplan las condiciones:

public class Animal {  
 private String especie;  
 private int edad;  
  
 public Animal(String especie) {  
 this.setEspecie(especie);  
 }  
  
 public String getEspecie() {  
 return especie;  
 }  
  
 public void setEspecie(String especie) {  
 if(especie == null || especie.trim().length()==0) {  
 throw new IllegalArgumentException("La especie es obligatoria");  
 }  
 this.especie = especie;  
 }  
  
 public int getAge() {  
 return edad;  
 }  
  
 public void setAge(int edad) {  
 if(edad < 0) {  
 throw new IllegalArgumentException("La edad no puede ser un número negativo");  
 }  
 this.edad = edad;  
 }  
}

Las variables especie y edad están marcadas como privadas, con métodos públicos getEspecie() y getEdad() para leer los datos. Además, setEspecie() y setEdad() ahora validan la entrada y lanzan una excepción si se viola una de nuestras invariantes. Además, se ha añadido un constructor no predeterminado que requiere un valor de especie y utiliza el método setter para validar la entrada.

La ventaja de esta nueva implementación de la clase Animal es que **utiliza la encapsulación para hacer cumplir los principios de diseño de la clase**. Cada vez que se pasa una instancia de un objeto Animal a un método, se puede usar sin requerir que se validen sus invariantes.

### **Bloqueo de acceso directo a atributos privados**

EN la práctica, los getter o setter a menudo ofrece un acceso casi directo a los atributos privados:

private String nombre;  
  
public String getNombre() {  
 return nombre;  
}  
  
public void setNombre(String nombre) {  
 this.nombre = nombre;  
}

Aunque puede parecer una mala encapsulación, el campo nombre se puede cambiar sin aplicar ninguna regla, es mucho mejor que permitir el acceso directo a la variable privada nombre. La ventaja radica en que fácilmente se puede actualizar el método getter o setter para tener reglas más complejas sin hacer que las otras clases que lo usan tengan que recompilar el código. El método setNombre() podría reescribirse de la siguiente manera:

public void setNombre(String nombre) {  
 this.nombre = (nombre == null || nombre.trim().length() == 0) ? null : nombre;  
}

Dado que la firma del método setNombre() no cambió, la invocación de este método no implica tener que modificar y recompilar su código.

Se considera una buena práctica de diseño encapsular siempre todas las variables en una clase, incluso si no hay reglas de datos establecidas, como una forma de proteger los datos cuando dichas reglas puedan añadirse en el futuro.

## **Creación de JavaBeans**

La encapsulación es tan prevalente en Java que **existe un estándar para crear clases que almacenan datos**, llamado JavaBeans. Un JavaBean es un **principio de diseño para encapsular datos** en un objeto en Java;

**Convenciones de nomenclatura de JavaBean:**

|  |  |
| --- | --- |
| **Regla** | **Ejemplo** |
| Las propiedades son privadas. | private int edad; |
| El getter para propiedades no booleanas comienza con get. | public int getEdad() { return edad; } |
| Los getters para propiedades booleanas pueden comenzar con is, has o get. | public boolean isAve() { return ave; }public boolean getAve() { return ave; } |
| Los métodos setters comienzan con set. | public void setEdad(int edad) { this.edad = edad; } |
| El nombre del método debe tener un prefijo de set/get/is/has, seguido de la primera letra de la propiedad en mayúscula y el resto del nombre de la propiedad. | public void setNumHijos(int numHijos) { this.numHijos = numHijos; } |

*Aunque los valores booleanos utilizan is para comenzar sus métodos getters, NO se aplica a las instancias de la clase envolvente Boolean, que utilizan get.*

Por ejemplo:

private boolean jugando;  
private Boolean bailando;

¿Cuál de las siguientes podría incluirse correctamente en un JavaBean?

public boolean isJugando() { return jugando; }  
public boolean getJugando() { return jugando; }  
public Boolean isBailando() { return bailando; }

La primera línea es correcta porque define un getter adecuado para una variable booleana. El segundo ejemplo también es correcto, ya que boolean puede usar is o get. La tercera línea es incorrecta, porque un envoltorio Boolean debería comenzar con get, ya que es un objeto.

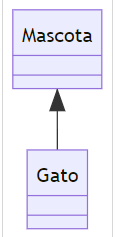
public String nombre;  
public String nombre() { return nombre; }  
public void actualizarNombre(String n) { nombre = n; }  
public void setnombre(String n) { nombre = n; }

¡Ninguna de estas líneas sigue las prácticas correctas de JavaBean! La primera línea hace público el nombre, cuando debería ser privado. La segunda línea no define un getter adecuado y debería ser getNombre(). Las dos últimas líneas son incorrectos setters, ya que la primera no comienza con set y la segunda no tiene la primera letra del nombre del atributo en mayúscula.

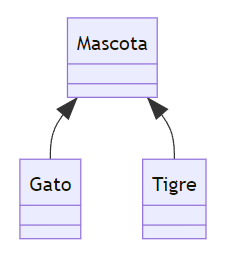
## **Relación Es‐un**

El operador instanceof se puede usar para determinar cuándo un objeto es una instancia de una clase, superclase o interfaz particular. En el diseño orientado a objetos, se describe la relación de que un objeto es una instancia de un tipo de datos como tener una relación es‐un. La relación es‐un también se conoce como la **prueba de herencia**.

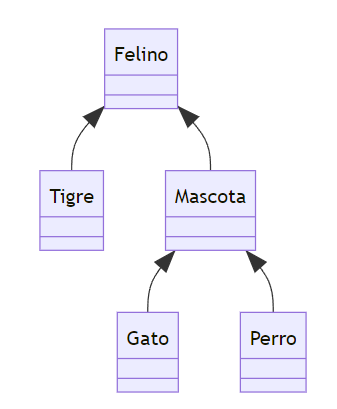
Cuando se construye un **modelo de datos basado en la herencia**, es importante aplicar la relación es‐un regularmente, para **diseñar clases que conceptualmente tengan sentido**. Por ejemplo, una clase Gato que hereda de una clase Mascota, como se muestra en la imagen:



La clase principal, Mascota, tiene campos comunes como nombre y edad. Se podría incurrir en el error de diseñar una clase Tigre, y dado que los tigres también tienen una edad y un nombre, se podría estar tentado en reutilizar la clase principal Mascota con el propósito de ahorrar tiempo y líneas de código, dando lugar a **diseños incorrectos**:

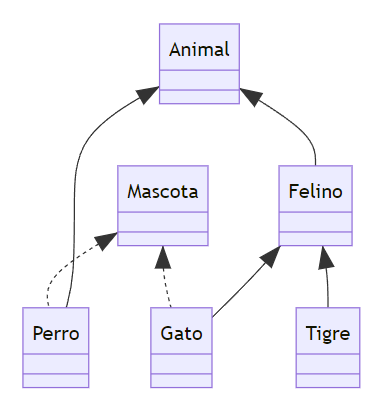


Por desgracias ;-), Mascota también tiene un método acariciar().  
Al reutilizar la clase principal Mascota, se está afirmando conceptualmente que un Tigre es‐una Mascota, pero un Tigre no es una Mascota. Aunque este ejemplo es funcionalmente correcto y ahorra tiempo y líneas de código, el resultado de no aplicar la relación es‐un es que se ha creado una relación que viola el modelo de datos. Intentemos solucionar el problema colocando Mascota y Tigre debajo de una clase padre Felino y veamos si eso resuelve el problema:



La estructura de clases es ahora más coherente, pero si se añade un hijo Perro a Mascota, nos encontramos con un problema con la prueba es‐un. Un Perro es‐una Mascota, y una Mascota es‐una Felino, pero el modelo implica que un Perro es‐un Felino, lo cual obviamente no es cierto.

La prueba de relación es‐un ayuda a evitar crear modelos de objetos que contengan contradicciones. Una solución en este ejemplo sería **no combinar Tigre y Mascota en el mismo modelo**, prefiriendo escribir código duplicado en lugar de crear datos inconsistentes. Una mejor solución podría ser utilizar las propiedades de las interfaces y declarar Mascota como una interfaz en lugar de una clase principal:

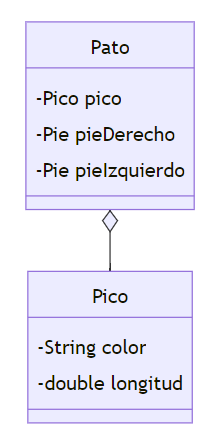


Ahora es correcto usando la prueba es‐un: Gato es‐un Animal, Tigre es‐un Felino, Perro es‐un Animal, y así sucesivamente. Mascota ahora está separada del modelo de herencia de clases, pero al usar interfaces, preservamos la relación de que Gato es‐una Mascota y Perro es‐una Mascota.

## **Relación Tiene‐un**

En el diseño orientado a objetos, a menudo queremos probar si un **objeto contiene una propiedad o valor en particular**. La relación tiene‐un cuando la propiedad de una clase tiene un nombre de un objeto o primitiva como miembro. La relación tiene‐un también se conoce como la prueba de **composición de objetos**.

Por ejemplo, las clases Pájaro y Pico:



Pájaro y Pico son clases con atributos y valores diferentes. Aunque obviamente fallan la prueba es‐un, ya que un Pájaro no es un Pico, ni un Pico es un Pájaro, sí pasan la prueba tiene‐un, ya que un Pájaro tiene un Pico. La herencia va un paso más allá al permitirnos decir que cualquier hijo de Pájaro también debe tener un Pico.

### **Problemas del modelo de datos empleando Es‐un y Tiene‐un**

A veces, las relaciones parecen pasar la prueba es‐un pero fallan al combinarse con la relación tiene‐un a través de la herencia. Por ejemplo:

public class Cola {}  
public class Primate {  
 protected Cola cola;  
}  
  
public class Mono extends Primate { // El mono tiene una cola ya que es un primate  
}  
public class Chimpance extends Primate { // El chimpancé tiene una cola ya que es un primate  
}

En el ejemplo, un Mono es‐un Primate y un Chimpance es‐un Primate. El modelo también establece que un Primate tiene‐una Cola, y mediante la herencia, un Mono tiene‐un Cola y un Chimpancé tiene‐un Cola. Sin embargo, los chimpancés no tienen cola, por lo que el modelo de datos subyacente es incorrecto.  
Deberíamos eliminar la propiedad Cola de la clase Primates, ya que no todos los primates tienen colas.

## **Composición de Objetos**

En el diseño orientado a objetos la composición de objetos es la propiedad de **construir una clase utilizando referencias a otras clases** para reutilizar la funcionalidad de esas clases.

**La clase contiene las otras clases en el sentido de tiene‐un y puede delegar métodos a las otras clases**.

La composición de objetos debe considerarse como una alternativa a la herencia y a menudo **se utiliza para simular un comportamiento polimórfico que no se puede lograr mediante la herencia simple**:

public class Aletas {  
 public void aletear() {  
 System.out.println("Las aletas se mueven de un lado a otro");  
 }  
}  
  
public class PatasPalmeadas {  
 public void patear() {  
 System.out.println("Las patas palmeadas patean de un lado a otro");  
 }  
}

Intentar relacionar estos objetos mediante la herencia no tiene sentido, ya que las PatasPalmeadas no son lo mismo que las Aletas. En cambio, podemos **crear una nueva clase que contenga ambas de estas clases y delegue sus métodos en ella**:

public class Pingüino {  
 private final Aletas aletas;  
 private final PatasPalmeadas patasPalmeadas;  
  
 public Pingüino() {  
 this.aletas = new Aletas();  
 this.patasPalmeadas = new PatasPalmeadas();  
 }  
  
 public void aletear() {  
 this.aletas.aletear();  
 }  
  
 public void patear() {  
 this.patasPalmeadas.patear();  
 }  
}

La clase Pingüino está compuesta por instancias de Aletas y PatasPalmeadas. La parte difícil de aletear() y patear() se delega a las otras clases, siendo los métodos en la clase Pingüino de solo una línea. Ten en cuenta que las implementaciones de estos métodos en las clases delegadas también tienen solo una línea, aunque podrían ser considerablemente más complejas.

Una de las ventajas de la composición de objetos sobre la herencia es que **tiende a promover una mayor reutilización de código**. Al utilizar la composición de objetos, se **accede a otras clases y métodos que serían difíciles de obtener mediante el modelo de herencia simple**.

En el ejemplo, la clase Aletas se puede reutilizar en clases completamente no relacionadas con un Pingüino o un Ave, como en una clase Delfín o Tortuga. Si la clase Aletas hubiera sido heredada de la clase Pingüino, entonces usarla en otras clases no relacionadas sería difícil sin romper el modelo de clases o tener que hacer que la otra clase contenga una instancia de un Pingüino. Por ejemplo, sería absurdo decir que un Delfín hereda de un Pingüino o tiene una instancia de una clase Pingüino, sólo porque un Delfín tiene Aletas y Aletas hereda de la clase Pingüino.

Por otro lado, **sobrecarga de métodos para determinar dinámicamente qué método seleccionar en tiempo de ejecución es una herramienta muy útil**.

# **02. PATRÓN SINGLETON**

* [1. El Patrón Singleton](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/03patronesdisenho/02singleton/#1-el-patr%C3%B3n-singleton)
* [2. Tipos de implementación del patrón Singleton](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/03patronesdisenho/02singleton/#2-tipos-de-implementaci%C3%B3n-del-patr%C3%B3n-singleton)
  + [1. Inicialización en la decleración](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/03patronesdisenho/02singleton/#1-inicializaci%C3%B3n-en-la-decleraci%C3%B3n)
  + [2. Usando bloques *static*](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/03patronesdisenho/02singleton/#2-usando-bloques-static)
  + [3. Instanciación “perezosa” de Singleton](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/03patronesdisenho/02singleton/#3-instanciaci%C3%B3n-perezosa-de-singleton)
    - [Singleton en varias computadoras](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/03patronesdisenho/02singleton/#singleton-en-varias-computadoras)
  + [4. Thread Safe Singleton](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/03patronesdisenho/02singleton/#4-thread-safe-singleton)
  + [5. Singletons con bloqueo de doble comprobación (*Double‐Checked Locking*)](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/03patronesdisenho/02singleton/#5-singletons-con-bloqueo-de-doble-comprobaci%C3%B3n-doublechecked-locking)
* [Conclusión](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/03patronesdisenho/02singleton/#conclusi%C3%B3n)

## **1. El Patrón Singleton**

En muchos casos, precisamos **crear un objeto en memoria sólo una vez en una aplicación** y **compartirlo entre varias clases**. Un ejemplo podría ser una conexión a una base de datos, pero sucede con frecuenta con otro tipo de recursos compartidos.

Por ejemplo, podríamos querer gestionar la cantidad de comida disponible para la alimentación de los animales de un zoológico en todas las clases que lo utilizan. Podríamos pasar el mismo objeto compartido ComidaManager a cada clase y método que lo utiliza, aunque esto crearía muchos punteros adicionales y podría ser difícil de gestionar si el objeto se utiliza en toda la aplicación. Al **crear un objeto singleton ComidaManager, centralizamos los datos y eliminamos la necesidad de pasarlos por toda la aplicación**.

* El **patrón singleton es un patrón de creación que permite sólo una instancia de un objeto en memoria dentro de una aplicación, compartida por todas las clases y hilos dentro de la aplicación**.
* El objeto disponible globalmente creado por el patrón singleton se denomina **objeto singleton**.
* Los singleton **también pueden mejorar el rendimiento cargando datos reutilizables que de otro modo serían costosos de almacenar y recargar cada vez que se necesitan**.

Una implementación sencilla de clase ComidaManager con un patrón singleton básico:

public class AlmacenComida {  
  
 private int cantidad = 0;  
  
 private AlmacenComida() {} // Privado, se crea como atributo estático:  
  
 private static final AlmacenComida instancia = new AlmacenComida(); // static: único.  
  
 // Método público único para obtener la instancia:  
 public static AlmacenComida getInstance() {  
 return instancia;  
 }  
  
 public synchronized void addComida(int cantidad) {  
 this.cantidad += cantidad;  
 }  
  
 public synchronized boolean removeComida(int cantidad) {  
 if (this.cantidad < cantidad) return false;  
 this.cantidad -= cantidad;  
 return true;  
 }  
  
 public synchronized int getCantidadComida() {  
 return cantidad;  
 }  
}

* Los **singleton** en Java se crean como **atributos privados estáticos dentro de la clase**, a menudo con el nombre instance (o instancia).
* Se **accede a ellos a través de un único método público estático**, a menudo **llamado getInstance()**, que devuelve la referencia al objeto singleton.
* Todos los c**onstructores en una clase singleton se declaran como privados**, lo que garantiza que **ninguna otra clase pueda instanciar otra versión de la clase**. Al marcar los constructores como privados, hemos marcado implícitamente la **clase como final**.

*Recuerda que cada clase requiere al menos un constructor, con el constructor por defecto si no se proporciona ninguno. Además, la primera línea de cualquier constructor es una llamada a un constructor padre super(). Si todos* ***los constructores se declaran privados en la clase singleton****, entonces es imposible crear una subclase con un constructor válido; por lo tanto, la* ***clase singleton es final****.*

* Se añade el modificador synchronized a addComida(), removeComida() y getCantidadComida(), para evitar que dos procesos ejecuten el mismo método exactamente al mismo tiempo y haya inconsistencias.

En el ejemplo AlmacenComida, un proceso que quiera usar este singleton primero llama a getInstance() y luego llama al método público necesario:

public class EntrenadorLlamas {  
  
 public boolean alimentarLlamas(int numeroLlamas) {  
  
 int cantidadNecesaria = 5 \* numeroLlamas;  
  
 AlmacenComida AlmacenComida = AlmacenComida.getInstance();  
  
 if (AlmacenComida.getCantidadComida() < cantidadNecesaria) {  
 AlmacenComida.addComida(cantidadNecesaria + 10);  
 }  
  
 boolean alimentadas = AlmacenComida.removeComida(cantidadNecesaria);  
  
 if (alimentadas)  
 System.out.println("Las llamas han sido alimentadas");  
 return alimentadas;  
 }  
  
}

Un aspecto a tener en cuenta es que **puede haber varias clases en la aplicación que requieran el objeto Singleton**.  
En el ejemplo, equivaldría a muchas instancias de EntrenadorLlamas pero sólo una instancia del objeto Singleton, AlmacenComida.  
El boolean devuelto por removeComida(), permite comprobar si hay disponible.  
En nuestro primer ejemplo de AlmacenComida, instanciamos el objeto singleton directamente en la definición de la referencia de instancia.

## **2. Tipos de implementación del patrón Singleton**

Se puede instanciar un singleton de varios modos:

a) En la **declaración del objeto**:

private static final ClaseSingleton instancia = new ClaseSingleton();

b) Utilizando un **bloque de inicialización estático**:

private static final ClaseSingleton instancia;  
  
 static {  
 instancia = new ClaseSingleton();  
 // Realizar pasos adicionales  
 }

c) **Instanciación “perezosa” de Singleton**. Se realiza en la primera llamada a la invocación del método getInstance:

private static ClaseSingleton instancia;  
  
 private ClaseSingleton() {  
 }  
  
 public static ClaseSingleton getInstance() {  
 if (instancia == null) {  
 instancia = new ClaseSingleton(); // ¡NO ES SEGURO PARA HILOS!  
 }  
 return instancia;  
 }

Por ejemplo:

// Instanciación usando un bloque estático  
public class RegistroPersonal {  
 private static final RegistroPersonal instancia;  
  
 static {  
 instancia = new RegistroPersonal();  
 // Realizar pasos adicionales  
 }  
  
 private RegistroPersonal() {  
 }  
  
 public static RegistroPersonal getInstance() {  
 return instancia;  
 }  
  
 // Métodos de acceso a datos  
 // ...  
}

Tanto la clase RegistroPersonal como la clase anterior AlmacenComida instancian el **singleton en el momento en que se carga la clase**. Sin embargo, a diferencia de la clase AlmacenComida, la clase RegistroPersonal instancía el singleton como parte de un bloque de inicialización estático.

Aunque **estas dos implementaciones son equivalentes**, ya que ambas crean el singleton cuando se carga la clase, el **bloque de inicialización estático permite realizar pasos adicionales para configurar el singleton después de que se ha creado**. También permite gestionar casos en los que el constructor de RegistroPersonal lanza una excepción.  
Dado que el singleton se crea cuando se carga la clase, **permite marcar la como final, lo que garantiza que sólo se creará una instancia dentro de la aplicación**.

Cuándo usar singleton

Los singletons se utilizan en situaciones donde **necesitamos acceso a un conjunto único de datos en toda una aplicación**. Por ejemplo: datos de configuración de la aplicación o las cachés de datos reutilizables se implementan comúnmente mediante singletons. Los singletons también se pueden utilizar para c**oordinar el acceso a recursos compartidos, como coordinar el acceso de escritura a un archivo o acceso a bases de datos**.

Hay muchas formas de hacer esto en Java. Todas estas formas difieren en su implementación del patrón, pero al final, todas logran el mismo resultado final de una única instancia.

### **1. Inicialización en la decleración**

Es el método más sencillo de crear una clase singleton. El objeto de la clase **se crea cuando se carga en la memoria por la JVM, asignando directamente la referencia de una instancia**.

Puede utilizarse cuando el programa siempre usará una instancia de esta clase, o el costo de crear la instancia no es demasiado grande en términos de recursos y tiempo.

// Código Java para crear una clase singleton mediante  
// "Inicialización Ansiosa"  
public class ClaseSingletonAD {  
 // instancia pública inicializada al cargar la clase  
 private static final ClaseSingletonAD instance = new ClaseSingletonAD();  
   
 private ClaseSingletonAD() {  
 // constructor privado  
 }  
   
 public static ClaseSingletonAD getInstance(){  
 return instance;  
 }  
}

**Ventajas/inconvenientes:**

* Muy **sencilla** de implementar.
* Puede llevar al **desperdicio de recursos**, ya que la instancia de la clase **se crea siempre**, ya sea que se necesite o no.
* También se **pierde tiempo de CPU** en la **creación de la instancia si no es necesaria**.
* **No es posible manejar excepciones**.

### **2. Usando bloques *static***

Es muy similar al caso anterior. La única diferencia es que **el objeto se crea en un bloque estático para que se pueda tener acceso a su creación**, **como el manejo de excepciones**. De esta manera, el objeto **también se crea en el momento de la carga de la clase**.

Puede utilizarse cuando hay posibilidad de excepciones al crear el objeto con inicialización ansiosa.

// Código Java para crear una clase singleton  
// Usando bloque estático  
public class ClaseSingletonAD {  
 // instancia pública  
 public static ClaseSingletonAD instance;  
   
 private ClaseSingletonAD() {  
 // constructor privado  
 }  
   
 static {  
 // bloque estático para inicializar la instancia  
 instance = new ClaseSingletonAD();  
 }  
}

**Ventanjas e inconvenientes:**

* Muy **sencillo** de implementar.
* No es necesario implementar el método getInstance(). **La instancia se puede acceder directamente**.
* Las **excepciones se pueden manejar** en el bloque estático.
* Puede llevar al **desperdicio de recursos**, ya que la instancia de la clase se crea siempre, ya sea que se necesite o no.
* También se **pierde tiempo de CPU en la creación de la instancia si no es necesaria**.

### **3. Instanciación “perezosa” de Singleton**

En este método, el objeto **se crea sólo si es necesario**. Esto puede evitar el desperdicio de recursos.

El objeto se **crea en la implementación del método getInstance() que devuelva la instancia**. Hay una comprobación de nulidad, y si el objeto no está creado, entonces se crea; de lo contrario, se devuelve el creado previamente.

Dado que el objeto se crea dentro de un método, se garantiza que el objeto no se creará a menos que sea necesario. La instancia se mantiene privada para que nadie pueda acceder directamente a ella.

Puede utilizarse **en un entorno de un solo hilo porque varios hilos pueden romper la propiedad singleton al acceder al método getInstance() simultáneamente y crear varios objetos**.

// inicialización perezosa (Lazy initialization)  
public class ClaseSingletonAD {  
 // instancia privada  
 // sólo accesible desde el méttodo getInstance()  
 private static ClaseSingletonAD instance;  
   
 private ClaseSingletonAD() {  
 // constructor privado  
 }  
   
 //devuelve la instancia de la clase  
 public static ClaseSingletonAD getInstance() {  
 if (instance == null) {  
 // si la instancia es nula se inicializa  
 instance = new ClaseSingletonAD();  
 }  
 return instance;  
 }  
}

Por ejemplo:

// Instanciación perezosa  
public class RastreadorTicketsVisitantes {  
 private static RastreadorTicketsVisitantes instancia;  
  
 private RastreadorTicketsVisitantes() {  
 }  
  
 public static RastreadorTicketsVisitantes getInstance() {  
 if (instancia == null) {  
 instancia = new RastreadorTicketsVisitantes(); // ¡NO ES SEGURO PARA HILOS!  
 }  
 return instancia;  
 }  
  
 // Métodos de acceso a datos  
 // ...  
}

El RastreadorTicketsVisitantes, al igual que nuestras clases singleton, declara sólo constructores privados, crea una instancia de singleton y devuelve el singleton con un método getInstance(). Sin embargo, la clase RastreadorTicketsVisitantes **crea el objeto singleton primera vez que un cliente lo solicita**. Crear un objeto reutilizable la primera vez que se solicita es un patrón de diseño de software conocido como instanciación perezosa (*Lazy Instantiation*), que se usa a menudo junto con el patrón singleton.

**Ventajas:** La instanciación perezosa **reduce el uso de memoria y mejora el rendimiento cuando se inicia una aplicación**. De hecho, sin instanciación perezosa, la mayoría de los sistemas operativos y aplicaciones que ejecutas tardarían significativamente más en cargarse y consumirían mucha más memoria, quizás más memoria de la disponible en tu computadora.

**Inconveniente:**" El inconveniente de la instanciación perezosa es que los usuarios **pueden notar un retraso notable la primera vez que se necesita un tipo particular de recurso**, como una conexión a una base de datos.

*Por ejemplo, muchas herramientas libres IDEs de desarrollo, como* ***Eclipse****, a menudo muestran un ligero retraso la primera vez que se abre un archivo Java en una ventana del editor después de iniciar el programa. Sin embargo, el retraso desaparece cuando se abren archivos Java adicionales. Este es un ejemplo de instanciación perezosa, ya que Eclipse* ***sólo carga las bibliotecas para analizar y presentar archivos Java la primera vez que se abre un archivo Java****.*

* El objeto **se crea sólo si es necesario**. Puede superar el desperdicio de recursos y tiempo de CPU.
* El **manejo de excepciones** también es posible en el método.
* Se debe **verificar la condición de nulo cada vez**.
* La instancia no se puede acceder directamente.
* En un **entorno multithread, puede romper la propiedad singleton**.

#### Singleton en varias computadoras

En principio, los singletons son siempre únicos. Cuando se escriben aplicaciones que se ejecutan en varias computadoras, la solución estática de singleton comienza a requerir consideraciones especiales, ya que cada computadora tendría su propio JVM. En esas situaciones, aún se podría usar el patrón singleton, aunque podría implementarse con una base de datos o **un servidor de colas en lugar de como un objeto estático**.

### **4. Thread Safe Singleton**

* Implementar verdaderamente el patrón singleton, debemos asegurarnos de que **sólo se cree una instancia del singleton** y está garantizado con el modelo anterior en entornos con un úncio hilo de ejecución.
* Marcar el constructor como privado evita que el singleton sea creado por otras clases, pero también **se necesita asegurar que el objeto singleton sólo se crea una vez dentro de la propia clase singleton**.

Se garantiza en los ejemplos de las clases AlmacenComida y RegistroPersonal utilizando el modificador final en la referencia estática.

Pero, en la **instanciación perezosa** en la clase RastreadorTicketsVisitantes, el compilador **no permite asignar el modificador final a la referencia estática**.

* La implementación de RastreadorTicketsVisitantes, como se muestra, **no se considera segura para hilos de ejecución (Threads)**, ya que dos hilos podrían llamar a getInstance() al mismo tiempo, lo que daría lugar a la creación dos objetos. Después de que ambos hilos terminen de ejecutarse, sólo se establecerá y utilizará un objeto por otros hilos en adelante, pero el objeto que recibieron los dos hilos iniciales puede no ser el mismo.

La seguridad de hilos (***Thread safety***) es la propiedad de un objeto que \*\*garantiza una ejecución segura por parte de múltiples hilos al mismo tiempo mediante el uso del modificador synchronized\*\* (por ejemplo, es la diferencia entre StringBuilder y StringBuffer). Hace que el método getInstance() sea sincronizado para que múltiples hilos no puedan acceder a él simultáneamente.

public static synchronized RastreadorTicketsVisitantes getInstance() {  
 if (instancia == null) {  
 instancia = new RastreadorTicketsVisitantes();  
 }  
 return instancia;  
}

El método getInstance() ahora está **sincronizado, lo que significa que sólo se permitirá que un hilo entre en el método a la vez, asegurando que sólo se cree un objeto**.

// Programa Java para crear una clase Singleton  
// segura para subprocesos  
public class ClaseSingletonAD   
{  
 // instancia privada, para que sólo pueda ser  
 // accedida por el método getInstance()  
 private static ClaseSingletonAD instance;  
   
 private ClaseSingletonAD() {  
 // constructor privado  
 }  
   
 // método sincronizado para controlar el acceso simultáneo  
 synchronized public static ClaseSingletonAD getInstance() {  
 if (instance == null) {  
 // si la instancia es nula, inicializar  
 instance = new ClaseSingletonAD();  
 }  
 return instance;  
 }  
}

**Ventajas e inconvenientes:**

* La inicialización perezosa se implanta.
* También es seguro para threads.
* El método **getInstance() está sincronizado**, por lo que provoca un **rendimiento lento, ya que varios hilos no pueden acceder a él simultáneamente**.

### **5. Singletons con bloqueo de doble comprobación (*Double‐Checked Locking*)**

La implementación sincronizada de getInstance(), aunque evita correctamente la creación de varios objetos singleton, **tiene el problema de que cada llamada a este método requerirá sincronización**. En la práctica, esto **puede ser costoso y afectar el rendimiento**.

La **sincronización sólo es necesaria la primera vez que se crea el objeto**. La solución es utilizar el doble bloqueo, un **patrón de diseño en el que primero probamos si se necesita sincronización antes de adquirir cualquier bloqueo**:

private static volatile ClaseSingleton instance;  
  
public static ClaseSingleton getInstance() {  
 if (instance == null) {  
 synchronized (ClaseSingleton.class) {  
 if (instance == null) {  
 instance = new ClaseSingleton();  
 }  
 }  
 }  
 return instance;  
}

Se añade el modificador volatile al objeto singleton. Esta palabra clave **evita un caso sutil donde el compilador intenta optimizar el código de manera que el objeto se acceda antes de que termine de construirse**.

Esta **solución es mejor** que la versión anterior, ya que realiza el paso de **sincronización sólo cuando el singleton no existe**. El singleton se accede miles de veces durante muchas horas o días, esto significa que sólo las primeras llamadas requerirían sincronización, y el resto no.

* Realiza **inicialización perezosa**.
* Es **thread-safe**
* Mejora el rendimiento, porque **reduce la sincronización**.
* La **primera vez puede afectar al rendimiento**.

## **Conclusión**

Singleton es un patrón de diseño útil para permitir sólo una instancia de su clase, pero los errores comunes pueden permitir que se cree más de una instancia sin darse cuenta.

El propósito del Singleton es **controlar la creación de objetos, limitando el número a uno pero permitiendo la flexibilidad de crear más objetos si la situación cambia**. Dado que sólo hay una instancia de Singleton, cualquier campo de instancia de un Singleton aparecerá sólo una vez por clase, al igual que los campos estáticos.

Los singleton a menudo controlan el **acceso a recursos como conexiones de bases de datos o sockets**. Por ejemplo, si se tiene una licencia para una sola conexión para su base de datos o su controlador JDBC tiene problemas con subprocesos múltiples, Singleton se asegura de que sólo se realice una conexión o que sólo un subproceso pueda acceder a la conexión a la vez. **Si añaden conexiones de base de datos o se utiliza un controlador JDBC que permite subprocesos múltiples, Singleton se puede ajustar fácilmente para permitir más conexiones**.

Además, los **Singleton pueden tener estado**; en este caso, su función es servir como depósito único del Estado. Si está implementando un contador que necesita dar números secuenciales y únicos (como la máquina que da números en la tienda de delicatessen), el contador debe ser globalmente único. El Singleton puede retener el número y sincronizar el acceso; Si más adelante desea mantener contadores en una base de datos para lograr persistencia, puede cambiar la implementación privada de Singleton sin cambiar la interfaz.

Por otro lado, los Singleton también pueden **proporcionar funciones de utilidad que no necesitan más información que sus parámetros**. En ese caso, no hay necesidad de crear instancias de múltiples objetos que no tienen ninguna razón para su existencia, por lo que un Singleton es apropiado.

# **03. OBJETOS INMUTABLES**

* [Manejo de Objetos Mutables en los Constructores de Objetos Inmutables](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/03patronesdisenho/03objetosinmutables/#manejo-de-objetos-mutables-en-los-constructores-de-objetos-inmutables)
* [“Modificación” de un Objeto Inmutable](https://manuais.pages.iessanclemente.net/plantillas/dam/ad/00repasojava/03patronesdisenho/03objetosinmutables/#modificaci%C3%B3n-de-un-objeto-inmutable)

## **1. Creación de Objetos Inmutables**

El próximo patrón creacional que discutiremos es el **patrón de objetos inmutables**, empelado para cuando deseamos tener **objetos de sólo lectura para que sean compartidos** y utilizados por varias clases.

A veces queremos crear objetos simples que puedan ser compartidos entre varias clases, pero por razones de seguridad no queremos que su valor sea modificado. Podríamos copiar el objeto antes de enviarlo a otro método, pero esto crea una gran sobrecarga que duplica el objeto cada vez que se pasa. Además, si tenemos varios hilos que acceden al mismo objeto, podríamos tener problemas de concurrencia, como veremos en el Capítulo 7.

**Solución:** El patrón de objeto inmutable es un patrón creacional basado en la idea de crear objetos cuyo estado no cambia después de que se crean y que se pueden compartir fácilmente entre varias clases. Los objetos inmutables van de la mano con la encapsulación, excepto que no existen métodos setter que modifiquen el objeto. Dado que el estado de un objeto inmutable nunca cambia, son inherentemente seguros para subprocesos.

## **Aplicación de una Estrategia Inmutable**

Aunque hay una variedad de técnicas para escribir una clase inmutable, debes estar familiarizado con una estrategia común para hacer que una clase sea inmutable para el examen:

1. Usa un constructor para establecer todas las propiedades del objeto.
2. Marca todas las variables de instancia como privadas y finales.
3. No definas ningún método setter.
4. No permitas que los objetos mutables referenciados se modifiquen o accedan directamente.
5. Evita que los métodos se anulen.

La primera regla define cómo creamos el objeto inmutable, pasando la información al constructor para que todos los datos se establezcan al crearlo. Las segundas y terceras reglas son directas, ya que provienen de la encapsulación adecuada. Si las variables de instancia son privadas y finales, y no hay métodos setter, entonces no hay una forma directa de cambiar la propiedad de un objeto. Todas las referencias y valores primitivos contenidos en el objeto se establecen en la creación y no se pueden modificar.

La cuarta regla requiere una explicación más detallada. Supongamos que tienes un objeto inmutable Animal, que contiene una referencia a una lista de alimentos favoritos del animal, como se muestra en el siguiente ejemplo:

import java.util.\*;  
  
public final class Animal {  
 private final List<String> favoriteFoods;  
  
 public Animal(List<String> favoriteFoods) {  
 if (favoriteFoods == null) {  
 throw new RuntimeException("favoriteFoods is required");  
 }  
 this.favoriteFoods = new ArrayList<String>(favoriteFoods);  
 }  
  
 public List<String> getFavoriteFoods() { // ¡HACE LA CLASE MUTABLE!  
 return favoriteFoods;  
 }  
}

Para asegurarnos de que la lista favoriteFoods no sea nula, la validamos en el constructor y lanzamos una excepción si no se proporciona. El problema en este ejemplo es que el usuario tiene acceso directo a la lista definida en nuestra instancia de Animal. Aunque no pueden cambiar el objeto List al que apunta, pueden modificar los elementos de la lista, por ejemplo, eliminando todos los elementos llamando a getFavoriteFoods().clear(). También podrían reemplazar, eliminar o incluso ordenar la lista.

La solución, entonces, es nunca devolver esa referencia de lista al usuario. De manera más general, nunca deb

es compartir referencias a un objeto mutable contenido dentro de un objeto inmutable. Si el usuario necesita acceso a los datos en la lista, ya sea creando métodos envolventes para iterar sobre los datos o creando una copia única de los datos que se devuelve al usuario y nunca se almacena como parte del objeto. De hecho, la API de Collections incluye el método Collections.unmodifiableList(), que hace exactamente esto.

La clave aquí es que ninguno de los métodos que creas debe modificar el objeto mutable.

Volviendo a nuestras cinco reglas, la última regla es importante porque evita que alguien cree una subclase de tu clase en la que un valor previamente inmutable ahora parece mutable. Por ejemplo, podrían anular un método que modifica una variable diferente en la subclase, ocultando essencialmente la variable privada definida en la clase principal. La solución más simple es marcar la clase o los métodos con el modificador final, aunque esto limita el uso de la clase. Otra opción es hacer que el constructor sea privado y aplicar el patrón de fábrica, que discutiremos más adelante en este capítulo.

Una clase Animal inmutable:

import java.util.\*;  
  
public final class Animal {  
 private final String species;  
 private final int age;  
 private final List<String> favoriteFoods;  
  
 public Animal(String species, int age, List<String> favoriteFoods) {  
 this.species = species;  
 this.age = age;  
 if (favoriteFoods == null) {  
 throw new RuntimeException("favoriteFoods is required");  
 }  
 this.favoriteFoods = new ArrayList<String>(favoriteFoods);  
 }  
  
 public String getSpecies() {  
 return species;  
 }  
  
 public int getAge() {  
 return age;  
 }  
  
 public int getFavoriteFoodsCount() {  
 return favoriteFoods.size();  
 }  
  
 public String getFavoriteFood(int index) {  
 return favoriteFoods.get(index);  
 }  
}

¿Sigue este ejemplo las cinco reglas? Bueno, todos los campos están marcados como privados y finales, y el constructor los establece al crear el objeto. Luego, no hay métodos setter y la clase en sí está marcada como final, por lo que los métodos no pueden ser anulados por una subclase. La clase contiene un objeto mutable, List, pero no hay referencias al objeto disponibles públicamente. Proporcionamos dos métodos para recuperar el número total de alimentos favoritos, así como un método para recuperar un alimento basado en un valor de índice. Ten en cuenta que String se considera inmutable, por lo que no tenemos que preocuparnos de que los objetos String se modifiquen. Por lo tanto, se cumplen las cinco reglas y las instancias de esta clase son inmutables.

## MANEJO DE OBJETOS MUTABLES EN LOS CONSTRUCTORES DE OBJETOS INMUTABLES

Puedes notar que creamos un nuevo ArrayList en el constructor de Animal. Esto es absolutamente importante para evitar que la clase que crea inicialmente el objeto mantenga una referencia a la List mutable utilizada por Animal. Considera si hubiéramos hecho lo siguiente en el constructor:

this.favoriteFoods = favoriteFoods;

Con este cambio, el llamador que crea el objeto está utilizando la misma referencia que el objeto inmutable, lo que significa que ¡tiene la capacidad de cambiar la List! Es importante, al crear objetos inmutables, copiar cualquier argumento de entrada mutable a la instancia en lugar de usarlo directamente.

## “MODIFICACIÓN” DE UN OBJETO INMUTABLE

¿Cómo modificamos objetos inmutables si son inherentemente inmodificables? La respuesta es que ¡no podemos! Alternativamente, podemos crear nuevos objetos inmutables que contengan toda la misma información que el objeto original más lo que queríamos cambiar. Esto sucede cada vez que combinamos dos cadenas:

String firstName = "Grace";  
String fullName = firstName + " Hopper";

En este ejemplo, firstName es inmutable y no se modifica al agregarse a fullName, que también es un objeto inmutable. También podemos hacer lo mismo con nuestra clase Animal. Imagina que queremos aumentar la edad de un Animal en uno. Lo siguiente crea dos instancias de Animal, la segunda utilizando una copia de los datos de la primera instancia:

// Crea una nueva instancia de Animal  
Animal leon = new Animal("león", 5, Arrays.asList("carne", "más carne"));  
// Crea una nueva instancia de Animal usando datos de la primera instancia  
List<String> alimentosFavoritos = new ArrayList<String>();  
for (int i = 0; i < leon.getFavoriteFoodsCount(); i++) {  
 alimentosFavoritos.add(leon.getFavoriteFood(i));  
}  
Animal leonActualizado = new Animal(leon.getSpecies(), leon.getAge() + 1, alimentosFavoritos);

Dado que no teníamos acceso directo a la List mutable favoriteFoods, tuvimos que copiarla utilizando los métodos disponibles en la clase inmutable. También podríamos simplificar esto definiendo un método en Animal que devuelva una copia de la List de alimentos favoritos, siempre que el llamador comprenda que modificar esta List copiada no cambia el objeto Animal original de ninguna manera.