

Práctico 3: Introducción a la Orientación a Objetos

Soluciones

Kata 1: Registro de Estudiantes (Nivel Básico)

Solución

Paso 1: Definir la Clase Estudiante

Creemos una clase **Estudiante** que modela a un estudiante con sus atributos y métodos.

```
// Definimos la clase Estudiante
public class Estudiante {
    // Atributos privados para mantener el encapsulamiento
    private String nombre;
    private String apellido;
    private String curso;
    private double calificacion;

    // Constructor para inicializar un estudiante
    public Estudiante(String nombre, String apellido, String curso, double calificacion) {
        this.nombre = nombre;
        this.apellido = apellido;
        this.curso = curso;
        setCalificacion(calificacion); // Usamos setter para aplicar reglas de validación
    }

    // Método para mostrar la información del estudiante
    public void mostrarInfo() {
        System.out.println("Estudiante: " + apellido + ", " + nombre);
        System.out.println("Curso: " + curso);
        System.out.println("Calificación: " + calificacion);
    }

    // Método para subir la calificación del estudiante
    public void subirCalificacion(double puntos) {
        // Verificamos si al sumar los puntos la calificación no supera el máximo permitido (10)
        if (calificacion + puntos <= 10) {
            // Si no supera el máximo, aumentamos la calificación
            calificacion += puntos;
            // Mostramos un mensaje con la nueva calificación
            System.out.println("La calificación ha aumentado a: " + calificacion);
        } else {
            // Si supera el máximo, mostramos un mensaje de error
            System.out.println("Error: No se puede superar la calificación máxima de 10.");
        }
    }

    // Método para bajar la calificación del estudiante
    public void bajarCalificacion(double puntos) {
```

```
// Verificamos si al restar los puntos la calificación no es menor que el mínimo permitido (0)
if (calificacion - puntos >= 0) {
    // Si no es menor que 0, disminuimos la calificación
    calificacion -= puntos;
    // Mostramos un mensaje con la nueva calificación
    System.out.println("La calificación ha disminuido a: " + calificacion);
} else {
    // Si es menor que 0, mostramos un mensaje de error
    System.out.println("Error: No se puede tener una calificación menor a 0.");
}
}

// Getters y Setters para encapsulación y validación

public String getNombre() {
    return nombre;
}

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

public String getApellido() {
    return apellido;
}

public void setApellido(String apellido) {
    this.apellido = apellido;
}

public String getCurso() {
    return curso;
}

public void setCurso(String curso) {
    this.curso = curso;
}

public double getCalificacion() {
    return calificacion;
}

public void setCalificacion(double calificacion) {
    if (calificacion < 0) {
        this.calificacion = 0; // Evita valores negativos
    } else if (calificacion > 10) {
        this.calificacion = 10; // Evita valores mayores a 10
    } else {
        this.calificacion = calificacion;
    }
}
}
```

Paso 2: Crear una instancia y probar los métodos

En el método **main**, instanciamos un estudiante y realizamos las operaciones requeridas.

```
public class Main {  
    public static void main(String[] args) {  
        // Crear un estudiante con datos iniciales  
        Estudiante estudiante1 = new Estudiante("Ana", "Gómez", "Matemáticas", 8.5);  
  
        // Mostrar información inicial del estudiante  
        System.out.println("\n--- Información Inicial ---");  
        estudiante1.mostrarInfo();  
  
        // Intentar subir la calificación en 1.5 puntos (debe ser válido)  
        System.out.println("\n--- Intentando subir la calificación en 1.5 puntos ---");  
        estudiante1.subirCalificacion(1.5);  
  
        // Intentar subir la calificación en 2 puntos (debe mostrar error porque supera 10)  
        System.out.println("\n--- Intentando subir la calificación en 2 puntos ---");  
        estudiante1.subirCalificacion(2);  
  
        // Intentar bajar la calificación en 3 puntos (debe ser válido)  
        System.out.println("\n--- Intentando bajar la calificación en 3 puntos ---");  
        estudiante1.bajarCalificacion(3);  
  
        // Intentar bajar la calificación en 10 puntos (debe mostrar error porque quedaría negativa)  
        System.out.println("\n--- Intentando bajar la calificación en 10 puntos ---");  
        estudiante1.bajarCalificacion(10);  
  
        // Mostrar información final del estudiante  
        System.out.println("\n--- Información Final ---");  
        estudiante1.mostrarInfo();  
    }  
}
```



Explicación del Código:

Encapsulación:

- Se usan atributos privados (**private**) para evitar modificaciones directas.
- Se implementan métodos **get** y **set** para acceder y modificar los atributos de manera controlada.

Métodos con validaciones:

- Método **subirCalificacion**:
- **subirCalificacion(double puntos)**: No permite valores mayores a **10**.
 - Paso 1: Verificamos si al sumar los **puntos** a la **calificacion** actual, el resultado no supera el valor máximo permitido (10).
 - Paso 2: Si no supera el máximo, aumentamos la **calificacion** sumando los **puntos**.
 - Paso 3: Mostramos un mensaje con la nueva calificación.

- Paso 4: Si supera el máximo, mostramos un mensaje de error indicando que no se puede superar la calificación máxima.
- Método **bajarCalificacion**:
- **bajarCalificacion(double puntos)**: No permite valores menores a 0.
 - Paso 1: Verificamos si al restar los **puntos** a la **calificacion** actual, el resultado no es menor que el valor mínimo permitido (0).
 - Paso 2: Si no es menor que 0, disminuimos la **calificacion** restando los **puntos**.
 - Paso 3: Mostramos un mensaje con la nueva calificación.
 - Paso 4: Si es menor que 0, mostramos un mensaje de error indicando que no se puede tener una calificación menor a 0.

Implementación del **main** para pruebas:

- Se crea una instancia de **Estudiante**.
- Se muestra la información inicial.
- Se aumentan y disminuyen las calificaciones validando los límites.
- Se muestra la información final del estudiante.

Ejemplo de Salida en Consola

--- Información Inicial ---

Estudiante: Gómez, Ana

Curso: Matemáticas

Calificación: 8.5

--- Intentando subir la calificación en 1.5 puntos ---

La calificación ha aumentado a: 10.0

--- Intentando subir la calificación en 2 puntos ---

Error: No se puede superar la calificación máxima de 10.

--- Intentando bajar la calificación en 3 puntos ---

La calificación ha disminuido a: 7.0

--- Intentando bajar la calificación en 10 puntos ---

Error: No se puede tener una calificación menor a 0.

--- Información Final ---

Estudiante: Gómez, Ana

Curso: Matemáticas

Calificación: 7.0

Qué aprendiste con este ejercicio:

Modelar clases y objetos en Java.

- ✓ Implementar métodos y encapsulación.
- ✓ Controlar la visibilidad con **private**, **public** y **protected**.
- ✓ Utilizar **getters** y **setters** para acceder y modificar atributos.
- ✓ Aplicar validaciones para evitar valores inválidos.

💡 Este ejercicio es un excelente punto de partida para aprender la POO en Java! 🚀

Kata 2: Registro de Mascotas (Nivel Básico)

Enunciado

Imagina que estás desarrollando un sistema de gestión de mascotas para un refugio de animales.

Debes modelar la clase Mascota con los siguientes atributos y métodos:

- **Atributos:**

nombre
especie
edad

- **Métodos:**

mostrarInfo()
cumplirAños().

🎯 Tarea a realizar

- Crear una instancia de la clase Mascota con un nombre, especie y edad inicial.
- Mostrar la información de la mascota: nombre, especie y edad inicial.
- Llamar a cumplirAños() para aumentar la edad en 1 año.
- Mostrar la información actualizada.

Paso 1: Definir la Clase mascota

```
// Clase Mascota que modela una mascota en el refugio.
public class Mascota {
    // Atributos privados para encapsular los datos
    private String nombre;
    private String especie;
    private int edad;

    // Constructor de la clase Mascota para inicializar los atributos
    public Mascota(String nombre, String especie, int edad) {
        this.nombre = nombre;
        this.especie = especie;
        this.edad = edad;
    }

    // Método para mostrar la información de la mascota
    public void mostrarInfo() {
        System.out.println("🐾 Nombre: " + nombre);
        System.out.println("🦋 Especie: " + especie);
        System.out.println("🎂 Edad: " + edad + " años");
    }
}
```

```
}

// Método para incrementar la edad en 1 año
public void cumplirAnios() {
    edad++; // Aumenta la edad en 1
    System.out.println("🎉 ¡Feliz cumpleaños! Ahora " + nombre + " tiene " + edad + " años.");
}

// Métodos Getters y Setters para acceder y modificar atributos privados
public String getNombre() {
    return nombre;
}

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

public String getEspecie() {
    return especie;
}

public void setEspecie(String especie) {
    this.especie = especie;
}

public int getEdad() {
    return edad;
}

public void setEdad(int edad) {
    if (edad >= 0) { // Validación para evitar edades negativas
        this.edad = edad;
    } else {
        System.out.println("⚠️ La edad no puede ser negativa.");
    }
}
}
```

Paso 2: Crear una instancia y probar los métodos

En el método **main**, instanciamos una Mascota y realizamos las operaciones requeridas.

```
public class Main {
    // Método main para probar la clase Mascota
    public static void main(String[] args) {
        // Crear una instancia de Mascota con nombre, especie y edad inicial
        Mascota miMascota = new Mascota("Firulais", "Perro", 3);

        // Mostrar la información inicial de la mascota
        System.out.println("📌 Información inicial:");
        miMascota.mostrarInfo();


        // Llamar al método cumplirAnios para incrementar la edad
        miMascota.cumplirAnios();
    }
}
```


```
// Mostrar la información actualizada de la mascota
System.out.println("\n 🐾 Información actualizada:");
miMascota.mostrarInfo();
}}
```


Explicación del Código:


1. **Encapsulación:**
 - Los atributos (**nombre**, **especie**, **edad**) son **private**, lo que impide su acceso directo desde fuera de la clase.
 - Se proporcionan métodos **get** y **set** para acceder y modificar estos valores de forma controlada.
2. **Constructor:**
 - Se define un **constructor** para inicializar los atributos de la mascota.
3. **Métodos:**
 - **mostrarInfo()**: Imprime en consola los detalles de la mascota.
 - **cumplirAnios()**: Incrementa la edad en 1 año y muestra un mensaje.
4. **Control de Datos:**
 - En **setEdad()**, se valida que la edad no sea negativa.
5. **Prueba en main():**
 - Se crea un objeto **Mascota** con valores iniciales.
 - Se muestra la información de la mascota antes y después de aumentar la edad.


Ejemplo de Salida en Consola


 Información inicial:


 Nombre: Firulais


 Especie: Perro


 Edad: 3 años

 ¡Feliz cumpleaños! Ahora Firulais tiene 4 años.

 Información actualizada:

 Nombre: Firulais

 Especie: Perro

 Edad: 4 años

Este ejercicio introduce los conceptos fundamentales de clases, objetos, atributos, métodos, encapsulación y uso de getters/setters en Java

Kata 3: Primeros Pasos en Encapsulamiento (Nivel Intermedio)

Enunciado

Desarrolla una clase Libro para gestionar libros en una biblioteca.
Para evitar cambios incorrectos, implementa encapsulamiento.

- **Atributos privados:**

título
autor
añoPublicacion.

- **Métodos públicos:**

getTitulo()
getAutor()
getAñoPublicacion().

Método **setAñoPublicacion(int nuevoAño)**, con validación: **No se puede modificar si el año es menor a 1900 o mayor al año actual.**

Tarea a realizar

1. Crear un objeto de la clase Libro.
2. Intentar modificar el año de publicación con un valor inválido y otro válido.
3. Mostrar la información del libro:: título, autor, añoPublicacion.

Paso 1: Definir la Clase Libro

```
import java.time.Year; // Importación para obtener el año actual
```

```
// Definición de la clase Libro
```

```
public class Libro {  
    // Atributos privados para aplicar encapsulamiento  
    private String titulo;  
    private String autor;  
    private int añoPublicacion;  
  
    // Constructor para inicializar los valores del libro  
    public Libro(String titulo, String autor, int añoPublicacion) {  
        this.titulo = titulo;  
        this.autor = autor;  
        this.añoPublicacion = añoPublicacion;  
    }  
  
    // Métodos Getters para obtener los atributos del libro  
    public String getTitulo() {  
        return titulo;  
    }  
  
    public String getAutor() {  
        return autor;  
    }
```



```
}

public int getAñoPublicacion() {
    return añoPublicacion;
}

// Método Setter para modificar el año de publicación con validación
public void setAñoPublicacion(int nuevoAño) {
    int añoActual = Year.now().getValue(); // Obtener el año actual

    if (nuevoAño >= 1900 && nuevoAño <= añoActual) {
        this.añoPublicacion = nuevoAño;
        System.out.println("✅ Año de publicación actualizado correctamente.");
    } else {
        System.out.println("⚠️ Error: El año debe estar entre 1900 y " + añoActual);
    }
}

// Método para mostrar la información del libro
public void mostrarInfo() {
    System.out.println("📖 Título: " + titulo);
    System.out.println("✍️ Autor: " + autor);
    System.out.println("📅 Año de Publicación: " + añoPublicacion);
}
}
```

Paso 2: Crear una instancia y probar los métodos

En el método `main`, instanciamos la clase `Libro` y realizamos las operaciones requeridas.

```
public class Main {

    // Método main para probar la clase Libro
    public static void main(String[] args) {
        // Crear un objeto Libro con valores iniciales
        Libro miLibro = new Libro("Java How to Program", "Deitel", 2017);

        // Mostrar la información inicial del libro
        System.out.println("📌 Información Inicial del Libro:");
        miLibro.mostrarInfo();

        // Intentar modificar el año de publicación con un valor válido
        System.out.println("\n🔄 Intentando cambiar el año a 2000...");
        miLibro.setAñoPublicacion(2000);
        miLibro.mostrarInfo();

        // Intentar modificar el año de publicación con un valor incorrecto
        System.out.println("\n🔄 Intentando cambiar el año a 1800...");
        miLibro.setAñoPublicacion(1800); // No debería permitirlo
        miLibro.mostrarInfo();
    }
}
```

```
// Intentar modificar el año de publicación con un valor futuro
System.out.println("\n↩ Intentando cambiar el año al futuro (2030)...");
miLibro.setAñoPublicacion(2030); // No debería permitirlo
miLibro.mostrarInfo();
}
```

Explicación del Código

1 Encapsulamiento

- Los atributos `titulo`, `autor` y `añoPublicacion` son `private`, lo que impide su modificación directa desde fuera de la clase.
- Se usan **métodos `get`** para leer los valores y un **método `set` con validación** para modificar el `añoPublicacion`.

2 Constructor

- Se usa un **constructor** para inicializar los valores al crear un objeto `Libro`.

3 Métodos Getters

- `getTitulo()`, `getAutor()`, `getAñoPublicacion()` permiten acceder a los atributos sin modificarlos.




4 Método `setAñoPublicacion(int nuevoAño)`

- Valida que el año esté entre 1900 y el año actual.
- Si el año es válido, se actualiza.
- Si el año es inválido, se muestra un mensaje de error y no se modifica.

5 Método `mostrarInfo()`

- Muestra los datos del libro en la consola.

6 Pruebas en `main()`

- Se crea un libro con datos iniciales.
- Se intenta cambiar su año de publicación a:
 - 2000 (válido) →  Se actualiza correctamente.
 - 1800 (inválido) →  No se permite.
 - 2030 (inválido) →  No se permite.

Salida Esperada en Consola

 Información Inicial del Libro:

 Título: 1984

 Autor: George Orwell

 Año de Publicación: 1949

 Intentando cambiar el año a 2000...

 Año de publicación actualizado correctamente.

 Título: 1984

 Autor: George Orwell

 Año de Publicación: 2000

 Intentando cambiar el año a 1800...

 Error: El año debe estar entre 1900 y 2025

 Título: 1984

 Autor: George Orwell

 Año de Publicación: 2000

 Intentando cambiar el año al futuro (2030)...

 Error: El año debe estar entre 1900 y 2025

 Título: 1984

 Autor: George Orwell


 Año de Publicación: 2000

¿Qué aprendiste con este ejercicio?

 **Encapsulamiento:** Los atributos están protegidos y solo pueden modificarse con métodos específicos.

 **Getters y Setters:** Uso de `get` para obtener valores y `set` con validación para modificar datos de forma segura.

 **Validación de datos:** Se evita que se introduzcan años incorrectos (menores a 1900 o futuros).

 **Organización del código:** Se estructuran bien los métodos y se aplican buenas prácticas de programación.

 **Pruebas en `main()`:** Se verifica que el comportamiento de la clase es el esperado.

Kata 4: Estado e Identidad de los Objetos (Nivel Intermedio-Avanzado)

En esta actividad, vas a programar un sistema para gestionar gallinas en una granja digital, registrando su producción de huevos y envejecimiento.

Requisitos del modelo

Cada **gallina** tendrá los siguientes atributos:

- `idGallina` → Identificador único.
- `edad` → Representa la edad de la gallina en años.
- `huevosPuestos` → Cantidad total de huevos que ha puesto.

Además, contará con los siguientes métodos:

- `ponerHuevo()` → Incrementa en 1 la cantidad de huevos puestos por la gallina.
- `envejecer()` → Aumenta en 1 su edad.
- `mostrarEstado()` → Muestra en pantalla la información actual de la gallina (`idGallina`, `edad`, `huevosPuestos`).

Tarea a realizar

1. **Crear dos gallinas** diferentes, asignando un identificador único a cada una.
2. Hacer que **cada gallina ponga al menos un huevo**.
3. Hacer que **cada gallina envejezca un año**.
4. **Mostrar el estado** de cada gallina después de estas acciones.

Pasos para resolver el problema

1 Crear la clase **Gallina** con los atributos:

- `idGallina`: Identificador único de la gallina.
- `edad`: Representa la edad en años.
- `huevosPuestos`: Número total de huevos que ha puesto.

2 Encapsular los atributos (**private**) para proteger los datos.

3 Implementar los métodos:

- `ponerHuevo()`: Aumenta en 1 la cantidad de huevos.
- `envejecer()`: Aumenta en 1 la edad de la gallina.
- `mostrarEstado()`: Muestra en consola la información actual.

4 Probar el código en el método **main()**:

- Crear dos objetos **Gallina** con IDs únicos.
- Hacer que cada una ponga al menos un huevo.
- Hacer que cada una envejezca un año.
- Mostrar la información final de cada gallina.

Paso 1: Definir la Clase Gallina

// Definición de la clase Gallina

```
public class Gallina {  
    // Atributos privados para aplicar encapsulamiento  
    private int idGallina;  
    private int edad;  
    private int huevosPuestos;  
  
    // Constructor para inicializar los valores de la gallina  
    public Gallina(int idGallina, int edadInicial) {  
        this.idGallina = idGallina;  
        this.edad = edadInicial;  
        this.huevosPuestos = 0; // Comienza sin haber puesto huevos  
    }  
  
    // Método para incrementar en 1 la cantidad de huevos puestos  
    public void ponerHuevo() {  
        huevosPuestos++;  
        System.out.println("🥚 La gallina " + idGallina + " ha puesto un huevo. Total: " +  
huevosPuestos);  
    }  
  
    // Método para aumentar la edad de la gallina en 1 año  
    public void envejecer() {  
        edad++;  
        System.out.println("🎂 La gallina " + idGallina + " ha envejecido. Ahora tiene " + edad + "  
años.");  
    }  
  
    // Método para mostrar el estado actual de la gallina  
    public void mostrarEstado() {  
        System.out.println("🐔 Gallina ID: " + idGallina);  
        System.out.println("📅 Edad: " + edad + " años");  
        System.out.println("🥚 Huevos puestos: " + huevosPuestos);  
        System.out.println("-----");  
    }  
}
```

Paso 2: Crear una instancia y probar los métodos

En el método **main**, instanciamos la clase Gallina y realizamos las operaciones requeridas.

```
public class Main {
```

```
// Método main para probar la clase Gallina
public static void main(String[] args) {
    // Crear dos gallinas con identificadores únicos
    Gallina gallina1 = new Gallina(101, 2);
    Gallina gallina2 = new Gallina(102, 3);

    // Mostrar el estado inicial de las gallinas
    System.out.println("📌 Estado Inicial de las Gallinas:");
    gallina1.mostrarEstado();
    gallina2.mostrarEstado();

    // Hacer que cada gallina ponga al menos un huevo
    gallina1.ponerHuevo();
    gallina2.ponerHuevo();

    // Hacer que cada gallina envejezca un año
    gallina1.envejecer();
    gallina2.envejecer();

    // Mostrar el estado final después de las acciones
    System.out.println("\n📌 Estado Final de las Gallinas:");
    gallina1.mostrarEstado();
    gallina2.mostrarEstado();
}}
```

Explicación del Código

Encapsulamiento

- Se declaran los atributos como **private** para que solo se puedan modificar mediante métodos de la clase.
- **idGallina**, **edad**, y **huevosPuestos** están protegidos del acceso directo desde fuera de la clase.

Constructor

- **Gallina(int idGallina, int edadInicial)** permite inicializar el ID y la edad.
- **huevosPuestos** inicia en **0** ya que la gallina aún no ha puesto huevos.


Métodos Implementados


- ✓ **ponerHuevo()**: Incrementa **huevosPuestos** en 1 y muestra un mensaje.
- ✓ **envejecer()**: Incrementa **edad** en 1 y muestra un mensaje.
- ✓ **mostrarEstado()**: Imprime la información actual de la gallina.


Pruebas en **main()**


1. Se crean **dos gallinas** con IDs únicos (**101** y **102**).
2. Se muestra su estado **antes** de realizar acciones.
3. Cada gallina pone **al menos un huevo**.
4. Cada gallina **envejece un año**.
5. Se muestra el estado **después de las acciones**.

Salida Esperada en Consola


 Estado Inicial de las Gallinas:


 Gallina ID: 101


 Edad: 2 años


 Huevos puestos: 0

 Gallina ID: 102


 Edad: 3 años


 Huevos puestos: 0


 La gallina 101 ha puesto un huevo. Total: 1


 La gallina 102 ha puesto un huevo. Total: 1


 La gallina 101 ha envejecido. Ahora tiene 3 años.

 La gallina 102 ha envejecido. Ahora tiene 4 años.


 Estado Final de las Gallinas:


 Gallina ID: 101

 Edad: 3 años

 Huevos puestos: 1

 Gallina ID: 102

 Edad: 4 años

 Huevos puestos: 1

¿Qué aprendiste con este ejercicio?

- ✓ Encapsulamiento: Se protegen los atributos de la gallina (**idGallina**, **edad**, **huevosPuestos**).
- ✓ Estado e Identidad de los Objetos: Cada objeto **Gallina** tiene un **idGallina** único y un estado que cambia con el tiempo.
- ✓ Métodos y comportamiento: Implementamos métodos (**ponerHuevo()**, **envejecer()**, **mostrarEstado()**) que permiten interactuar con los objetos.
- ✓ Organización del código: Se estructura bien el código con un constructor, métodos y pruebas en **main()**.

Kata 5: Comportamiento de los Objetos (Nivel Avanzado)

Enunciado

- Imagina que trabajas en una **agencia espacial** y necesitas programar un simulador de **naves espaciales**.
- Debes desarrollar una clase **NaveEspacial** con un sistema de **combustible limitado**, lo que obliga a gestionar eficientemente los recursos.

Especificaciones

Atributos:

- nombre (String) → Nombre de la nave espacial.
- combustible (int) → Cantidad actual de combustible disponible.

Métodos:

- despegar(): Reduce 10 unidades de combustible al despegar. No puede despegar si hay menos de 10 unidades.
- avanzar(int distancia): Consume 1 unidad de combustible por cada unidad de distancia. No puede avanzar si no hay suficiente combustible.
- recargarCombustible(int cantidad): Aumenta la cantidad de combustible en la nave. No puede superar el límite máximo de 100 unidades.
- mostrarEstado(): Muestra el nombre de la nave y la cantidad de combustible actual.

Reglas:

- ✓ No puede **despegar** con menos de 10 unidades de combustible.
- ✓ No puede **avanzar** si el combustible no es suficiente para la distancia requerida.
- ✓ No puede **sobrecargar combustible** más allá del límite de **100 unidades**.

Tarea a realizar

1. **Intentar resolver la kata sin mirar la solución.**
2. Crear una instancia de la clase **NaveEspacial** con un nombre y **50 unidades de combustible**.
3. Intentar **avanzar 60 unidades** sin recargar (debe fallar por falta de combustible).
4. Recargar **40 unidades** de combustible.
5. Intentar **avanzar 60 unidades nuevamente** (ahora debe funcionar).
6. Mostrar el **estado actual de la nave**.
7. Comparar con la solución y **ajustar el código** si es necesario.
8. **Repetir el ejercicio** con diferentes valores para reforzar la comprensión.

Pasos para resolver el problema

1 Crear la clase `NaveEspacial` con los atributos:

- `nombre`: Nombre de la nave.
- `combustible`: Cantidad actual de combustible.

2 Encapsular los atributos (`private`) para proteger los datos.

3 Implementar los métodos:

- `despegar()`: Reduce 10 unidades de combustible si hay suficiente.
- `avanzar(int distancia)`: Consume 1 unidad por cada unidad de distancia.
- `recargarCombustible(int cantidad)`: Recarga combustible sin superar 100 unidades.
- `mostrarEstado()`: Muestra el estado actual de la nave.

4 Probar el código en el método `main()`:

- Crear una nave con 50 unidades de combustible.
- Intentar avanzar 60 unidades sin recargar (debe fallar).
- Recargar 40 unidades.
- Intentar avanzar 60 unidades nuevamente (ahora debe funcionar).
- Mostrar el estado final.

Paso 1: Definir la Clase Gallina

// Definición de la clase `NaveEspacial`

```
public class NaveEspacial {
    // Atributos privados para encapsular los datos
    private String nombre;
    private int combustible;
    private static final int MAX_COMBUSTIBLE = 100; // Límite máximo de combustible

    // Constructor para inicializar los valores de la nave
    public NaveEspacial(String nombre, int combustibleInicial) {
        this.nombre = nombre;
        this.combustible = Math.min(combustibleInicial, MAX_COMBUSTIBLE); // No puede
superar 100
    }

    // Método para despegar, reduciendo 10 unidades de combustible si es posible
    public void despegar() {
        if (combustible >= 10) {
            combustible -= 10;
            System.out.println("🚀 " + nombre + " ha despegado. Combustible restante: " +
combustible);
        } else {
```

```
        System.out.println("⚠ No hay suficiente combustible para despegar. Se requieren al  
menos 10 unidades.");  
    }  
}  
  
    // Método para avanzar una cierta distancia, consumiendo 1 unidad de combustible  
    por unidad de distancia  
    public void avanzar(int distancia) {  
        if (combustible >= distancia) {  
            combustible -= distancia;  
            System.out.println("🚀 " + nombre + " ha avanzado " + distancia + " unidades.  
Combustible restante: " + combustible);  
        } else {  
            System.out.println("🛑 No hay suficiente combustible para avanzar " + distancia + "  
unidades. Combustible actual: " + combustible);  
        }  
    }  
  
    // Método para recargar combustible sin superar el límite de 100 unidades  
    public void recargarCombustible(int cantidad) {  
        if (cantidad <= 0) {  
            System.out.println("⚠ No se puede agregar una cantidad negativa o cero de  
combustible.");  
            return;  
        }  
  
        if (combustible + cantidad > MAX_COMBUSTIBLE) {  
            combustible = MAX_COMBUSTIBLE;  
            System.out.println("🔧 Combustible recargado al máximo (100 unidades).");  
        } else {  
            combustible += cantidad;  
            System.out.println("🔧 " + nombre + " ha recargado " + cantidad + " unidades.  
Combustible actual: " + combustible);  
        }  
    }  
  
    // Método para mostrar el estado actual de la nave  
    public void mostrarEstado() {  
        System.out.println("\n🚀 Nave: " + nombre);  
        System.out.println("🔧 Combustible disponible: " + combustible + " unidades");  
        System.out.println("-----");  
    }  
}
```

Paso 2: Crear una instancia y probar los métodos

En el método `main`, instanciamos la clase `NaveEspacial` y realizamos las operaciones requeridas.

```
public class Main {  
    // Método main para probar la clase NaveEspacial  
    public static void main(String[] args) {  
        // Crear una instancia de NaveEspacial con un nombre y 50 unidades de combustible  
        NaveEspacial nave1 = new NaveEspacial("Apolo 11", 50);  
  
        // Mostrar estado inicial de la nave  
        System.out.println("📌 Estado Inicial de la Nave:");  
        nave1.mostrarEstado();  
  
        // Intentar avanzar 60 unidades sin recargar (debe fallar)  
        System.out.println("\n🔄 Intentando avanzar 60 unidades sin recargar...");  
        nave1.avanzar(60);  
  
        // Recargar 40 unidades de combustible  
        System.out.println("\n🔄 Recargando 40 unidades de combustible...");  
        nave1.recargarCombustible(40);  
  
        // Intentar avanzar 60 unidades nuevamente (ahora debe funcionar)  
        System.out.println("\n🔄 Intentando avanzar 60 unidades nuevamente...");  
        nave1.avanzar(60);  
  
        // Mostrar estado final de la nave  
        System.out.println("\n📌 Estado Final de la Nave:");  
        nave1.mostrarEstado();  
    }  
}
```

Explicación del Código

1 Encapsulamiento

- Los atributos `nombre` y `combustible` son `private` para evitar modificaciones directas.
- Se usa `static final int MAX_COMBUSTIBLE = 100;` para definir el límite máximo de combustible.

2 Constructor

- Inicializa la nave con un nombre y una cantidad de combustible (máximo 100 unidades).




3 Métodos Implementados



- ✓ `despegar()`: Reduce 10 unidades de combustible si hay suficiente.
- ✓ `avanzar(int distancia)`: Consume 1 unidad de combustible por cada unidad de distancia. Si no hay suficiente, muestra un error.
- ✓ `recargarCombustible(int cantidad)`: Agrega combustible sin exceder 100 unidades.
- ✓ `mostrarEstado()`: Muestra el estado de la nave.



4) Pruebas en `main()`



1. Se crea una nave "**Apolo 11**" con **50 unidades de combustible**.
2. Se intenta avanzar **60 unidades** sin recargar (debe fallar).
3. Se recargan **40 unidades de combustible**.
4. Se intenta avanzar **60 unidades** nuevamente (ahora debe funcionar).
5. Se muestra el estado final de la nave.




Salida Esperada en Consola

 Estado Inicial de la Nave:
 Nave: Apolo 11
 Combustible disponible: 50 unidades

 Intentando avanzar 60 unidades sin recargar...
 No hay suficiente combustible para avanzar 60 unidades. Combustible actual: 50

 Recargando 40 unidades de combustible...
 Apolo 11 ha recargado 40 unidades. Combustible actual: 90

 Intentando avanzar 60 unidades nuevamente...
 Apolo 11 ha avanzado 60 unidades. Combustible restante: 30

 Estado Final de la Nave:
 Nave: Apolo 11
 Combustible disponible: 30 unidades

¿Qué aprendiste con este ejercicio?

- ✓ Encapsulamiento: Los atributos están protegidos y solo pueden modificarse con métodos específicos.
- ✓ Control de Estado: Se asegura que la nave no pueda despegar o avanzar sin combustible suficiente.
- ✓ Límites y Validaciones: Se evita sobrecargar el combustible o usar valores negativos.
- ✓ Organización del código: Se estructuran bien los métodos y se aplican buenas prácticas de programación.
- ✓ Pruebas en `main()`: Se verifica el comportamiento correcto del código.

◆ Conclusión

Este ejercicio refuerza la gestión de recursos y validaciones en objetos Java. La nave espacial es un objeto con reglas de comportamiento bien definidas, aplicando buenas prácticas de Programación Orientada a Objetos (POO). 🚀🔥