

**PROGRAMACIÓN**

## **UP6. USO AVANZADO DE CLASES Y OBJETOS**



**1º CFGS DAW**

Curso 2024-25



## 6.1. Herencia y polimorfismo

### 6.1. Herencia y redefinición de métodos (@Override + super)

La **herencia** en Java es un mecanismo que permite que una clase (**subclase** o clase hija) adquiera los atributos y métodos de otra clase (**superclase** o clase padre).

Esto nos ayuda a reutilizar código, evitar duplicaciones y crear una jerarquía de clases más organizada.

#### Uso de *extends* y *super* en la Herencia

En Java, la herencia se implementa usando la palabra clave **extends**, y para acceder a los métodos o atributos de la superclase usamos **super**. Si hacemos analogía con lo que ya conocemos de la POO, *super* equivaldría a *this*, pero apuntando a los elementos de nuestra clase "madre" en lugar de a los propios de la clase en la que nos encontramos.

#### Un ejemplo...

Imaginemos que estamos programando un sistema para un *Festival*. En un concierto hay diferentes tipos de personas, como *asistentes* y *artistas*. Todos tienen atributos en común, pero también características únicas que los diferencia.

- 1 **Superclase.** Definimos la clase "madre", o mejor dicho, los elementos que se tendrán en común. En este caso, todas las personas en el festival tienen un nombre y una edad:

```
1 //superclase
2 class Persona {
3     String nombre;
4     int edad;
5
6     public Persona(String nombre, int edad) {
7         this.nombre = nombre;
8         this.edad = edad;
9     }
10
11 }
```

```
11
12     public void mostrarInfo() {
13         System.out.println("Nombre: " + nombre);
14         System.out.println("Edad: " + edad);
15     }
}
```

- 2 **extends.** Creamos una subclase a partir de una superclase, lo que permite que una clase herede atributos y métodos de otra:

```
1 //subclase que hereda de Persona
2 class Asistente extends Persona {
3     private String entrada; //tipo de entrada (General, VIP, etc.)
4
5 }
```

- **Asistente** hereda de *Persona nombre* y *edad*, pero añade un nuevo atributo: tipo de *entrada*.

- 3 Usar **super()** en el constructor y métodos:

```
1 //subclase que hereda de Persona
2 class Asistente extends Persona {
3     private String entrada; //tipo de entrada (General, VIP, etc.)
4
5     public Asistente(String nombre, int edad, String entrada) {
6         super(nombre, edad); //llamamos al constructor de la clase Persona
7         this.entrada = entrada;
8     }
9
10    @Override
11    public void mostrarInfo() {
12        super.mostrarInfo(); //llamamos al método de la clase Persona
13        System.out.println("Tipo de entrada: " + entrada);
14    }
15 }
```

- Se usa *super(nombre, edad)* para llamar al constructor de *Persona* para evitar repetir código, además de asegurarnos de que su *nombre* y *edad* se inicializan

correctamente.

- *super.mostrarInfo()* reutiliza el método de *Persona* antes de agregar más información dentro de su propio método *mostrarInfo()*. Si *Persona* cambia su *mostrarInfo()*, todas las subclases se actualizan automáticamente.

Además de *Asistentes*, podría haber *Artistas*:

```
1 //subclase que también hereda de Persona
2 class Artista extends Persona {
3
4     String generoMusical;
5
6     public Artista(String nombre, int edad, String generoMusical) {
7         super(nombre, edad);
8         this.generoMusical = generoMusical;
9     }
10
11    @Override
12    public void mostrarInfo() {
13        super.mostrarInfo();
14        System.out.println("Género Musical: " + generoMusical);
15    }
16
17 }
```

Vamos a escribir un *main* para probar nuestras clases:

```
1 public class Concierto {
2     public static void main(String[] args) {
3
4         Asistente a1 = new Asistente("Carlos", 25, "VIP");
5         Artista art1 = new Artista("Dua Lipa", 28, "Pop");
6
7         System.out.println("Información del asistente:");
8         a1.mostrarInfo();
9
10        System.out.println();
11
12        System.out.println("Información del artista:");
13        art1.mostrarInfo();
14
15
16 }
```

```
    }  
}
```

```
"C:\Program Files\Java\jdk-23\bin\java.exe" "-ja  
Información del asistente:  
Nombre: Carlos  
Edad: 25  
Tipo de entrada: VIP  
  
Información del artista:  
Nombre: Dua Lipa  
Edad: 28  
Género Musical: Pop
```



## Ejercicio

---

Sigue con el ejemplo anterior y añade una subclase *Organizador* para personas que trabajan en el evento. Dentro del *Festival* tendrá un un *rol* específico (ejemplo: "Seguridad", "Producción", etc.).

Añade lógica al programa principal para probar que funciona correctamente.

---

### Resumen del uso de herencia:

- Persona* es la superclase y almacena información común.
- Asistente*, *Artista* y *Organizador* son subclases que extienden *Persona*.
- Cada subclase añade características específicas.
- Se usa *super(...)* para reutilizar código de la clase padre.

# Ejercicio

---



## Ejercicio 1: Redes Sociales

---

Vamos a replicar una estructura de herencia que podría tener una red social.

- a) Crea una superclase de tipo *Usuario*, con los siguientes atributos: *nombre*, *edad*, *nombre de usuario* y *seguidores*.
- b) Crea una subclase *Influencer*, que incorpore un atributo nuevo de tipo *ArrayList<String>* llamado *colaboraciones*.
- c) Crea otra subclase llamada *Streamer*, con dos atributos extra: *numero\_retransmisiones* y *horas\_directo*.
- d) Por último, crea una subclase llamada *Basico*, sin atributos adicionales.
- e) Crea la clase principal *RedSocial* para ir creando instancias de los diferentes tipos de usuarios y mostrando sus datos.

**Información del Usuario:**

Nombre: Ana

Edad: 30

Nombre de usuario: @ana

Seguidores: 231

**Información del Influencer:**

Nombre: Luis

Edad: 25

Nombre de usuario: @luis\_influencer

Seguidores: 50000

Colaboraciones: "MediaMarkt", "Yoigo", "Temu".

**Información del Streamer:**

Nombre: Carlos

Edad: 27

Nombre de usuario: @carlos\_stream

Retransmisiones realizadas: 150

Horas de directo: 2000

**Información de la Persona Normal:**

Nombre: Pedro

Edad: 22

Nombre de usuario: @pedro123

Seguidores: 455

## 6.2. Polimorfismo

---

Diferencia entre crear un objeto de una subclase directamente vs. usarlo como una referencia de la superclase

En el ejemplo del *Festival* de música, en el *main()* usamos:

```
1 | Asistente a1 = new Asistente("Carlos", 25, "VIP");
```

Pero también podríamos haber hecho esto:

```
1 | Persona a1 = new Asistente("Carlos", 25, "VIP");
```

¿Cuál es la diferencia?

### 1. Usar una referencia de la superclase (*Persona a1 = new Asistente(...)*)

- Se usa **polimorfismo**, lo que significa que a1 se comporta como una *Persona*.
- **Sólo podemos acceder a los métodos y atributos declarados en *Persona***, incluso si el objeto es un *Asistente*.
- Si un método está sobrescrito (@Override), sí que se usará la versión de la subclase. **Aunque no escribiéramos la anotación @Override también sería así**, ya que si sobrescribimos un método en una subclase, Java lo reconoce aunque no usemos @Override, siempre que el método siga la misma estructura (nombre, parámetros y tipo de retorno).

```
1 | Persona a1 = new Asistente("Carlos", 25, "VIP");
2 | p1.mostrarInfo(); //llamará al método de Asistente, NO al de Persona
```

Pero si intentamos acceder a *p1.entrada* (un atributo de *Asistente*), nos dará error:

```
1 | System.out.println(p1.entrada); //ERROR: no se puede acceder desde una
```

Esto es útil cuando queremos manejar objetos de distintas subclases de forma genérica, por ejemplo, en un *array* de *Persona*:

```
1 Persona[] personas = {  
2     new Asistente("Lucas", 18, "VIP"),  
3     new Artista("Sofía", 25, "Rock Alternativo"),  
4     new Organizador("Martín", 30, "Producción")  
5 };  
6  
7 for (Persona p : personas) {  
8     p.mostrarInfo(); //se ejecuta la versión sobrescrita de cada subcla  
9 }
```

```
"C:\Program Files\Java\jdk-23\bin\java.exe" "-javaagent:  
Nombre: Lucas  
Edad: 18  
Tipo de entrada: VIP  
Nombre: Sofía  
Edad: 25  
Género Musical: Rock Alternativo  
Nombre: Martín  
Edad: 30  
Rol en el festival: Producción
```

## 2. Usar una referencia de la subclase (*Asistente a1 = new Asistente(...)*)

```
1 Asistente p1 = new Asistente("Lucas", 18, "VIP");  
2 p1.mostrarInfo(); //llama al método de Asistente  
3 System.out.println(p1.entrada); //FUNCIONA: puede acceder a su propio a
```

- En este caso, *a1* es específicamente un *Asistente*, por lo que podemos acceder a todos sus atributos y métodos.
- **No hay polimorfismo aquí**, porque *a1* siempre se tratará como *Asistente*.
- Esto es útil cuando necesitamos acceder a funcionalidades específicas de la subclase.

¿Cuándo usar cada uno?

Caso	¿Cuál usar?
Manejar diferentes tipos de objetos en un solo <i>array/lista</i>	<i>Persona p = new Asistente(...)</i> (polimorfismo)
Sólo queremos tratar un objeto como su propia clase sin restricciones	<i>Asistente p = new Asistente(...)</i>
Queremos asegurarnos de que usamos la versión correcta de un método sobrescrito	Cualquiera, ya que <i>@Override</i> funcionará igual
Necesitamos acceder a métodos o atributos específicos de la subclase	<i>Asistente p = new Asistente(...)</i> (porque con <i>Persona</i> no podríamos)



## Implementación del polimorfismo en el Festival

1. Crearemos en la clase Persona un método nuevo llamado *accederEvento()*, que imprima lo siguiente:

*"Accediendo al evento..."*

2. Las clases Artista, Asistente y Organizador sobreescriben *accederEvento* para hacer algo diferente. Por ejemplo:

*Artista: "Accediendo como Artista: Preparando el show."*

*Asistente: "Accediendo como Asistente: Buscando asiento."*

*Organizador: "Accediendo como Organizador: Coordinando el evento."*

3. En la clase principal *Concierto*, deberemos crear un nuevo método *mostrarAcceso(Persona persona)* que llame a *accederEvento()* para el tipo de persona que se le pase por parámetro. Además, actualizaremos su lógica para incorporar lo siguiente:

```
public static void main(String[] args) {
    Persona artista = new Artista("Sofía", 25, "Rock Alternativo");
    Persona asistente = new Asistente("Lucas", 18, "VIP");
    Persona organizador = new Organizador("Martín", 30, "Producción");

    //usamos el método con polimorfismo
}
```

```
7 |     mostrarAcceso(artista);      //Accediendo como Artista: Preparando el show.
8 |     mostrarAcceso(asistente);    //Accediendo como Asistente: Buscando asiento.
9 |     mostrarAcceso(organizador); //Accediendo como Organizador: Coordinando el evento.
10| }
```

```
"C:\Program Files\Java\jdk-23\bin\java.exe" "-javaagent:C:\Program Files\Java\jdk-23\lib\jvm-debugger.jar"
Accediendo como Artista: Preparando el show.
Accediendo como Asistente: Buscando asiento.
Accediendo como Organizador: Coordinando el evento.
```

# Ejercicio

---



## Ejercicio 2: Empleados y sus tareas

---

Vamos a crear un programa para una empresa, donde diferentes tipos de empleados realizan tareas específicas. Todos son tratados como *Empleados*, pero su método *realizarTarea()* se comporta diferente según el tipo de empleado.

1. Crear una clase base llamada *Empleado* con: ***public void realizarTarea()*** → Imprime "*Empleado realizando una tarea genérica.*".

2. Crear tres subclases que hereden de *Empleado*:

- ***Desarrollador*** : sobrescribe *realizarTarea()* para imprimir "*Escribiendo código y solucionando bugs.*".
- ***Diseñador***: sobrescribe *realizarTarea()* para imprimir "*Creando diseños gráficos y prototipos.*".
- ***Gerente*** : sobrescribe *realizarTarea()* para imprimir "*Supervisando el proyecto y organizando reuniones.*".

3. Haz un programa principal que cree un *array* polimórfico de tipo ***Empleado[]*** que contenga al menos un desarrollador, un diseñador y un gerente. Recorre todos los elementos del *array* y llama a *realizarTarea()* en cada elemento.

```
== Usando el array polimórfico ==
Escribiendo código y solucionando bugs.
Creando diseños gráficos y prototipos.
Supervisando el proyecto y organizando reuniones.
```

4. Crea en la clase de tu programa principal un método ***asignarTarea(Empleado empleado)*** → Recibe un objeto de tipo *Empleado*, imprime "*Asignando tarea al empleado...*" y llama a su método *realizarTarea()*.

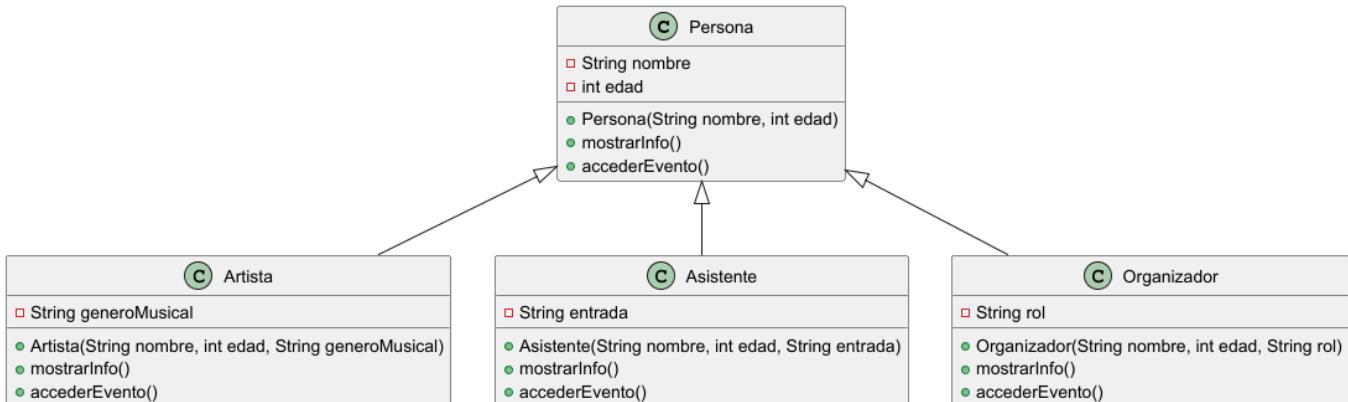
5. En el programa principal, instancia directamente un *Gerente* y un *Desarrollador* de tipo *Persona* (sin usar el *array*) y pásaselos al método *asignarTarea()* para comprobar que el polimorfismo funciona al pasar objetos de tipo *Persona* como parámetros.

```
== Usando el método asignarTarea() ==
Asignando tarea al empleado...
Supervisando el proyecto y organizando reuniones.
Asignando tarea al empleado...
Escribiendo código y solucionando bugs.
```

# Diagramas UML con herencia

---

En **UML**, la **herencia** se representa mediante una flecha con punta hueca que apunta de la subclase hacia la superclase:



## Ejercicio 3. Diagramas UML de Redes Sociales y Empresa

---

Recopila las clases usadas en los ejercicios 1 y 2 (Redes Sociales y Trabajadores de una empresa), y pídele a *ChatGPT* que genere el código *PlantUML* para obtener su diagrama correspondiente.



Obra publicada con Licencia Creative Commons Reconocimiento Compartir igual 4.0  
<<http://creativecommons.org/licenses/by-sa/4.0/>>

## 6.3. Clases abstractas

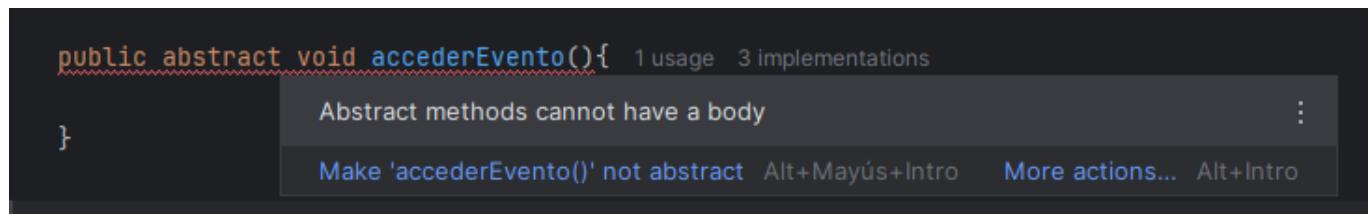
### 6.3. Clases abstractas (abstract + @Override)

Una **clase abstracta** es una clase que **no puede instanciarse por sí sola**. Es decir, solamente sirve como una plantilla para que otras clases la hereden y completen su comportamiento. En pocas palabras, es una base común para un grupo de clases relacionadas por el mismo tipo. Estas clases pueden tener atributos y constructores, aunque **el constructor sólo será llamado a través de sus subclases**.

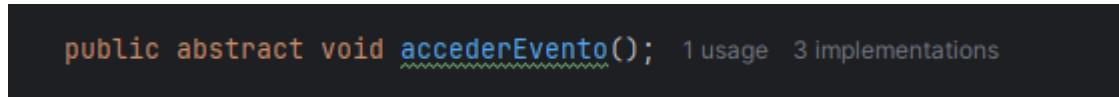
**OJO.** Aunque no sea posible instanciarse un objeto de la clase abstracta directamente, sí que se sigue pudiendo usar polimorfismo.

Además de clases, también podemos tener **métodos abstractos** que deben ser creados vacíos (sin cuerpo ni implementación), y **las subclases están obligadas a sobrescribirlos**.

--> Fíjate que si intentamos abrir llaves para darle un cuerpo, *IntelliJ IDEA* nos avisa de que no se puede:



Deberemos acabar la cabecera con un punto y coma (;) como si fuera una instrucción:



Ejemplo en Java:

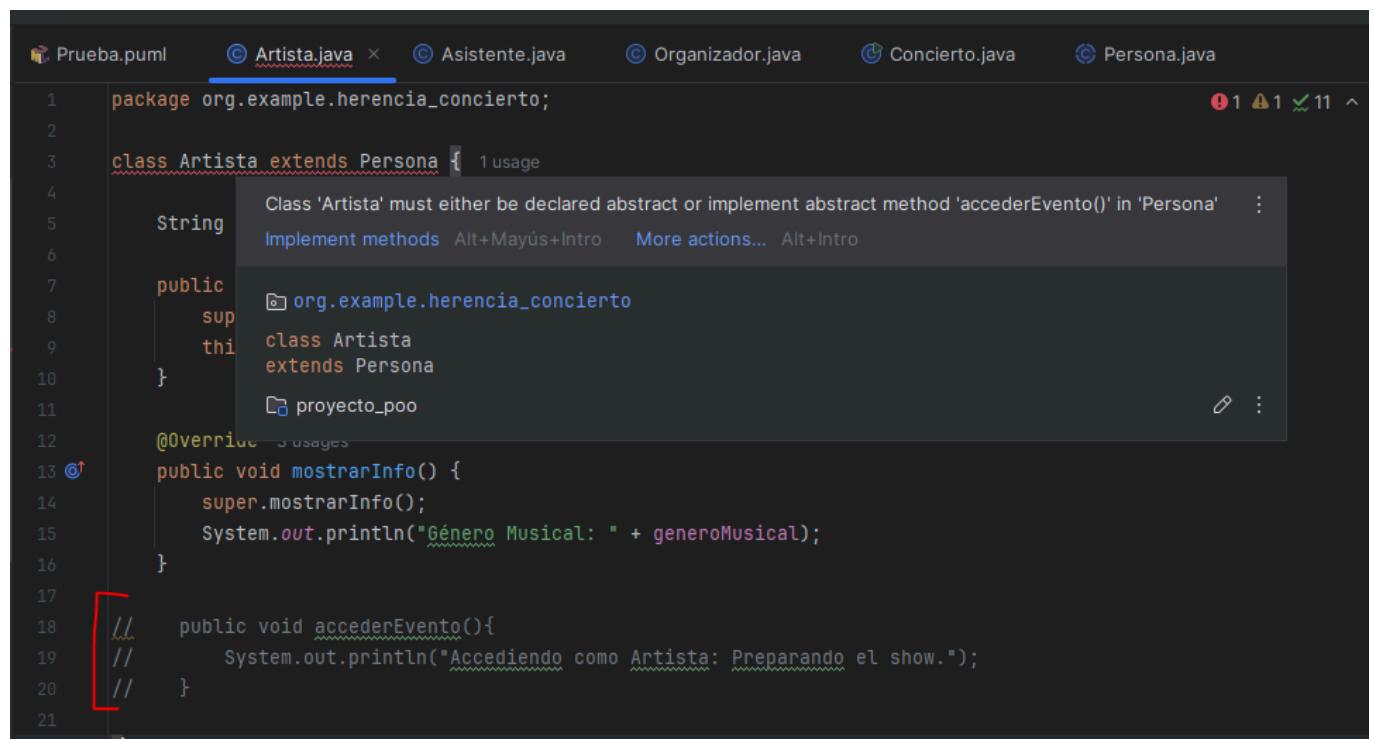
Supongamos que seguimos con el ejemplo del *Festival*, pero ahora queremos asegurarnos de que todos los participantes tengan que implementar su forma de acceder al evento. Podemos hacer que *Persona* sea una clase abstracta:

```
1 | abstract class Persona {  
2 |     String nombre;  
3 |     int edad;
```

```

4
5     public Persona (String nombre, int edad) {
6         this.nombre = nombre;
7         this.edad = edad;
8     }
9
10    //método tradicional (con implementación)
11    public void mostrarInfo() {
12        System.out.println("Nombre: " + nombre);
13        System.out.println("Edad: " + edad);
14    }
15
16    //método abstracto (sin implementación)
17    public abstract void accederEvento();
18
19 }
```

Si ahora intentamos borrar de cualquiera de las subclases hijas el método *accederEvento()* que ya teníamos sobrescrito, el intérprete de Java se quejará:



The screenshot shows an IDE interface with several tabs at the top: Prueba.puml, Artista.java (selected), Asistente.java, Organizador.java, Concierto.java, and Persona.java. The Artista.java tab has a red underline under the word 'Artista'. The code editor shows the following Java code:

```

1 package org.example.herencia_concierto;
2
3 class Artista extends Persona { 1 usage
4     String
5         Class 'Artista' must either be declared abstract or implement abstract method 'accederEvento()' in 'Persona'  :
6             Implement methods Alt+Mayús+Intro More actions... Alt+Intro
7
8     public
9         sup
10        thi
11    }
12
13 @Override 3 usages
14 public void mostrarInfo() {
15     super.mostrarInfo();
16     System.out.println("Género Musical: " + generoMusical);
17 }
18
19 // public void accederEvento(){
20 //     System.out.println("Accediendo como Artista: Preparando el show.");
21 }
```

A tooltip window is open over the 'Artista' class definition, displaying the error message: 'Class 'Artista' must either be declared abstract or implement abstract method 'accederEvento()' in 'Persona''. It also provides options to 'Implement methods' (Alt+Mayús+Intro), 'More actions...', and 'Alt+Intro'. A red bracket highlights the 'accederEvento()' method declaration in the code.

El método *mostrarInfo()* será heredado normalmente por las subclases, igual que pasaba en la herencia tradicional.

## ¿Cuándo usar una clase abstracta?

- Cuando tenemos un comportamiento común que queremos compartir entre varias clases.
- Cuando queremos forzar a las subclases a implementar ciertos métodos.



## Ejercicio. Sistema de Suscripciones en una Plataforma de Streaming

---

Imagina que estás desarrollando un sistema para gestionar diferentes tipos de suscripciones en una plataforma de *streaming*. Cada tipo de suscripción ofrece características y servicios distintos.

1. Clase abstracta ***Suscripcion*** con los atributos ***nombrePlan (String)*** y ***precio (double)***. Tendrá un método tradicional ***mostrarInfo()*** para imprimir el nombre del plan y su precio.
2. En la clase ***Suscripcion***, crea un método abstracto ***obtenerBeneficios()*** sin implementación. Cada tipo de plan detallará sus beneficios.
3. Subclases que heredan de ***Suscripcion***:
  - ***PlanGratis***: implementa ***obtenerBeneficios()*** imprimiendo: "Acceso limitado con anuncios."
  - ***PlanBasico***: implementa ***obtenerBeneficios()*** imprimiendo: "Acceso a todo el contenido en calidad estándar sin anuncios."
  - ***PlanPremium***: implementa ***obtenerBeneficios()*** imprimiendo: "Acceso a todo el contenido en alta definición y descargas offline."
  - ***PlanFamiliar***: implementa ***obtenerBeneficios()*** imprimiendo: "Acceso para varios perfiles simultáneamente en alta definición."
4. El programa principal ***PlataformaStreaming*** deberá:
  - Crear un array de elementos de tipo ***Suscripcion*** que contenga al menos un ***PlanGratis***, un ***PlanBasico***, un ***PlanPremium*** y un ***PlanFamiliar***.
  - Usar un bucle para recorrer el array y llamar a ***mostrarInfo()*** y ***obtenerBeneficios()*** para mostrar los datos de cada plan y ver los beneficios de cada suscripción..

```

Plan: Gratis, Precio: 0.0€
Acceso limitado con anuncios.
Plan: Básico, Precio: 9.99€
Acceso a todo el contenido en calidad estándar sin anuncios.
Plan: Premium, Precio: 14.99€
Acceso a todo el contenido en alta definición y descargas offline.
Plan: Familiar, Precio: 19.99€
Acceso para varios perfiles simultáneamente en alta definición.
  
```

5. Añade otro método abstracto en *Suscripcion* que se llame *obtenerPeriodoPrueba()* y sobrescríbelo en cada subclase. El método debe devolver información sobre el periodo de prueba gratuito que ofrece cada tipo de suscripción. Por ejemplo:

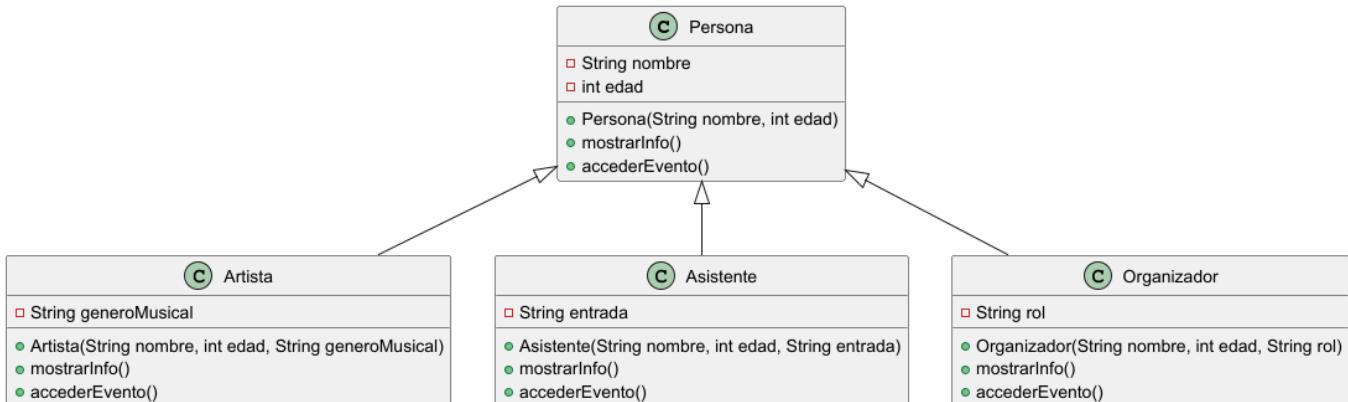
- *PlanGratis*: "Sin periodo de prueba"
- *PlanBasico*: "30 días de prueba gratuita"
- *PlanPremium*: "14 días de prueba gratuita"
- *PlanFamiliar*: "7 días de prueba para cuentas nuevas"

6. Añade lógica a tu programa principal para probar el nuevo método.

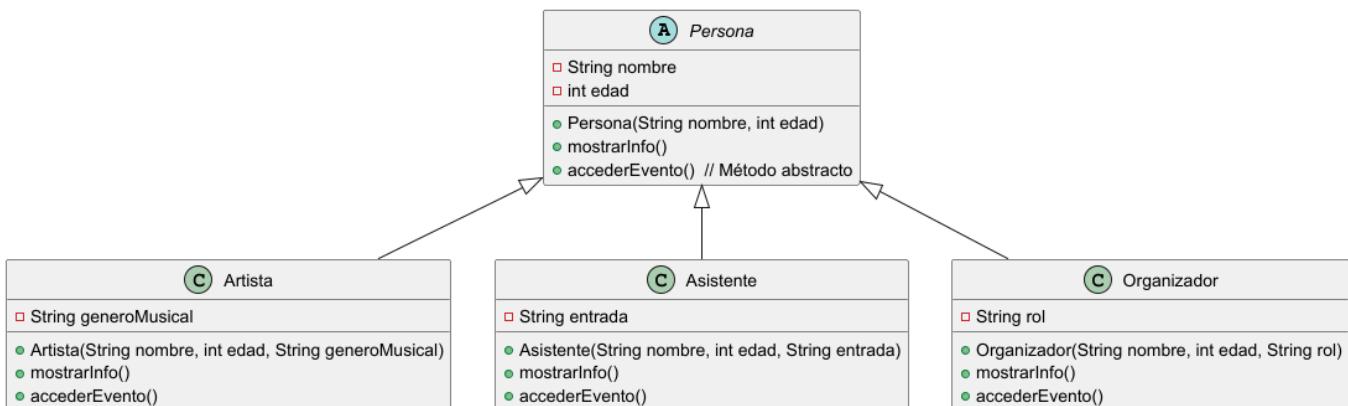
```
Plan: Gratis, Precio: 0.0€
Acceso limitado con anuncios.
Periodo de prueba: Sin periodo de prueba
Plan: Básico, Precio: 9.99€
Acceso a todo el contenido en calidad estándar sin anuncios.
Periodo de prueba: 30 días de prueba gratuita
Plan: Premium, Precio: 14.99€
Acceso a todo el contenido en alta definición y descargas offline.
Periodo de prueba: 14 días de prueba gratuita
Plan: Familiar, Precio: 19.99€
Acceso para varios perfiles simultáneamente en alta definición.
Periodo de prueba: 7 días de prueba para cuentas nuevas
```

# Diagrama UML de una clase abstracta

Si te fijas, en *PlantUML* siempre se ha generado un símbolo **C** al lado de las clases que hemos ido creando:



Cuando una de estas clases es abstracta, ese símbolo **C** se modificar por **A**:



```
1 @startuml
2
3     abstract class Persona {
4         - String nombre
5         - int edad
6         + Persona(String nombre, int edad)
7         + mostrarInfo()
8         + accederEvento() // Método abstracto
9     }
10
11    class Artista {
12        - String generoMusical
13        + Artista(String nombre, int edad, String generoMusical)
14        + mostrarInfo()
15        + accederEvento()
16    }
```

```

10 }
11
12 class Asistente {
13     - String entrada
14     + Asistente(String nombre, int edad, String entrada)
15     + mostrarInfo()
16     + accederEvento()
17 }
18
19 class Organizador {
20     - String rol
21     + Organizador(String nombre, int edad, String rol)
22     + mostrarInfo()
23     + accederEvento()
24 }
25
26 Persona <|-- Artista
27 Persona <|-- Asistente
28 Persona <|-- Organizador
29
30
31
32
33
34
35
36 @enduml

```



## Ejercicio

Crea el diagrama UML del ejercicio sobre la Plataforma de *Streaming*.



Obra publicada con Licencia Creative Commons Reconocimiento Compartir igual 4.0  
<http://creativecommons.org/licenses/by-sa/4.0/>

## PRACTICA 1. SISTEMA DE PAGO PARA UN E-COMMERCE



### → INTRODUCCIÓN

El profe turronero de *Lenguaje de Marcas* os ha contado tantas cosas sobre comercio online que os ha convencido: vais a crear un sitio web para estudiantes de programación que se dedique a vender cursos sobre tecnologías varias.

### → PROBLEMA A RESOLVER

Imagina que estás desarrollando el sistema de pagos para tu tienda de cursos online. La tienda acepta diferentes métodos de pago, como tarjeta de crédito, *PayPal* o *Bizum*. Cada método de pago tiene una forma distinta de procesar el pago, pero todos comparten la misma funcionalidad básica para hacerlo: un método llamado *procesarPago()*.

Realiza un programa en *Java* que implemente la lógica de la aplicación explicada anteriormente, usando POO mediante herencia, polimorfismo y abstracción.

### Construcción de las clases que van a interactuar entre ellas

La aplicación que necesitamos desarrollar constará de los siguientes elementos:

- Una clase abstracta llamada **MetodoPago**, con el siguiente método abstracto:

***void procesarPago(double importe)***

- Crea 3 subclases que extiendan y hereden de *MetodoPago*, una por cada tipo de método de pago permitido: *TarjetaCredito*, *PayPal* y *Bizum*.
- La clase **TarjetaCredito** tendrá los atributos *nro\_tarjeta* (*String* de 16 caracteres) y *tipo* (*String* que debe permitir solamente los siguientes valores: *VISA*, *MASTERCARD*, *MAESTRO*).
  - El método **procesarPago()** debe imprimir "Procesando pago de [importe] € con tarjeta de crédito *VISA*".
  - Debe haber un método más en la clase para **validarTarjeta()**, que compruebe que el *nro\_tarjeta* introducido tenga la longitud esperada y el *tipo* esté dentro de los valores permitidos.

- La clase **PayPal** tendrá los atributos *cuenta* (*String* con formato de correo electrónico “**xxx@xxx.com**”) y *saldo* (*double* por defecto 23€).
  - El método **procesarPago()** debe imprimir “Procesando pago de [importe] € con PayPal”.
  - En este caso, el método **validarPayPal()** debe comprobar que el formato del correo electrónico es correcto y que el *saldo* de la cuenta sea suficiente para realizar el pago correspondiente.
- La clase **Bizum** tendrá los atributos *telefono* (*String* de 9 caracteres) y *pin* (*int* de 6 dígitos que se generará de forma aleatoria).
  - El método **procesarPago()** debe imprimir “Procesando pago de [importe] € con Bizum”.
  - El método **validarBizum()** debe comprobar el formato del teléfono y que el pin introducido por el usuario es el correcto. **NOTA:** haz trampa e imprime el pin en cuanto se genere para poder ver cuál es y poder realizar el pago.
- Tendremos otra clase **Tienda** con un método estático:

***static void realizarPago(MetodoPago metodo, double importe)***

que invocará al método *procesarPago()* según el objeto *metodo* recibido como parámetro.

- Además, esta clase tendrá otro método estático **iniciarPago()**, donde previamente preguntará al usuario qué método de pago quiere usar para crear uno nuevo y realizar todas las validaciones correspondientes (*validarTarjeta*, *validarPayPal*, *validarBizum*) antes de ser procesado.
- Si lo necesitas, crea más métodos auxiliares para modularizar el código todo lo que puedas.
- Por último, crea un programa principal **AppEcommerce** que implemente toda la lógica.

```
public class AppEcommerce {
    public static void main(String[] args) {
        Tienda.iniciarPago();
    }
}
```

## Ejemplo de ejecución:

```
¿Qué método de pago quieras usar? [Bizum,PayPal,Tarjeta]:  
Tarjeta  
Introduce los datos de tu tarjeta:  
Número (16 digitos):  
1234  
tipo (VISA o MASTERCARD):  
VISA  
Validando tarjeta...  
Los datos de tu tarjeta no son correctos.
```

```
"C:\Program Files\Java\jdk-23\bin\java.exe" "-javaagent:C:\Program Files\J  
¿Qué método de pago quieras usar? [Bizum,PayPal,Tarjeta]:  
Tarjeta  
Introduce los datos de tu tarjeta:  
Número (16 dígitos):  
1234567812345678  
tipo (VISA o MASTERCARD):  
VISA  
Validando tarjeta...  
Introduce el importe a pagar:  
67  
Procesando pago de 67.0 con tarjeta de crédito VISA  
Pago aceptado. Muchas gracias.
```

## → REALIZACIÓN DE LA PRÁCTICA

Sigue los siguientes pasos para realizar la práctica. ¡Ve guardando tu trabajo de vez en cuando para evitar que se borre el avance si se cierra el editor de textos u ocurre cualquier problema en tu equipo!

1. Programa en Java la aplicación requerida
2. Diagrama UML



ENTREGA

**REALIZA UN INFORME EN PDF CON LA INFO GENERADA Y LOS PASOS SEGUIDOS PARA REALIZAR ESTA PRÁCTICA. EXPLICA TU CÓDIGO. SÚBELO TODO A LA TAREA DE AULES DISPONIBLE.**

**ADEMÁS, PEGA LA URL DE TU PROYECTO EN GITHUB.**

## 6.4. Interfaces

### 6.4. Interfaces

Si las clases abstractas se comportaban como un "molde", las interfaces van un poco más allá: son un "contrato" que las clases que implementen una interfaz deben cumplir. O dicho de otra forma, **una interfaz se implementa en una clase para que esta sepa qué tiene que hacer, pero no el cómo.**

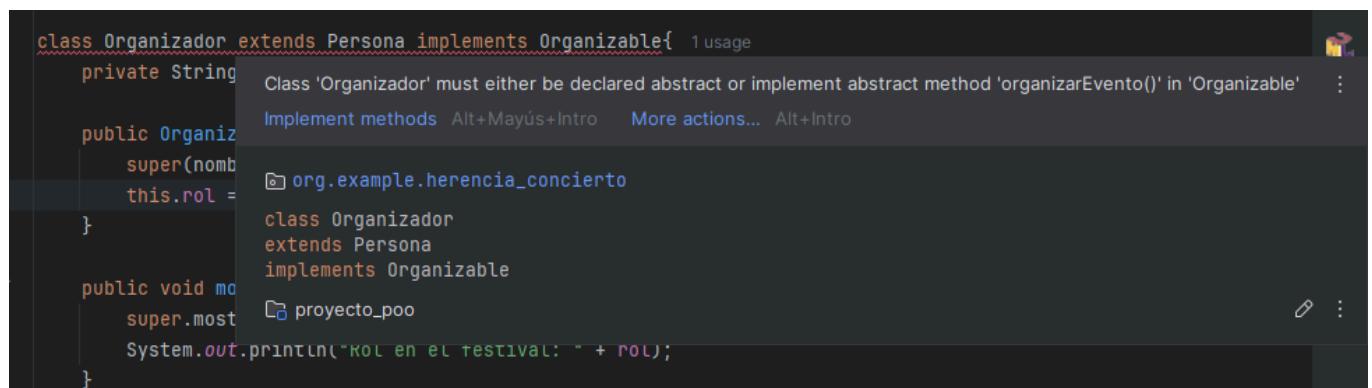
Para crear una interfaz, si seguimos con el ejemplo del *Festival*, escribiremos una interfaz con el siguiente formato:

```
1 //interfaz para quienes organizan el evento
2 public interface Organizable {
3     void organizarEvento();
4 }
```

#### ⚠️ FÍJATE:

- Los métodos son implícitamente públicos y abstractos SIN INDICARLO.
- No puede tener constructores, ya que no se pueden instanciar directamente. En cuanto a los atributos o variables, sólo podría tener constantes estáticas y finales (*public static final*).

La forma de implementar una interfaz es usando la palabra reservada *implements*, de la siguiente manera:



The screenshot shows a Java code editor with the following code:

```
class Organizador extends Persona implements Organizable{
    private String nom;
    public Organizador(String nom) {
        super(nom);
        this.nom = nom;
    }
    public void organizarEvento() {
        super.mostear();
        System.out.println("ROL en el festival: " + nom);
    }
}
```

A tooltip from the IDE indicates that the class 'Organizador' must either be declared abstract or implement the abstract method 'organizarEvento()' in 'Organizable'. This is a common IDE feature for提醒开发者实现接口的方法。

Igual que nos pasaba con los métodos abstractos, nada más indicar en la clase que queremos usar la interfaz nos mostrará el **ERROR** y nos obligará a implementar los métodos

que dice el "contrato", es decir, los métodos que tenga la interfaz.

```
1 class Organizador extends Persona implements Organizable {  
2  
3     private String rol; // Ejemplo: "Seguridad", "Producción", "Prensa"  
4  
5     public Organizador(String nombre, int edad, String rol) {  
6         super(nombre, edad);  
7         this.rol = rol;  
8     }  
9  
10    public void mostrarInfo() {  
11        super.mostrarInfo();  
12        System.out.println("Rol en el festival: " + rol);  
13    }  
14  
15    public void accederEvento(){  
16        System.out.println("Accediendo como Organizador: Coordinando el  
17    }  
18  
19    public void organizarEvento() {  
20        System.out.println("Organizando...");  
21    }  
22}  
23}
```

## Herencia múltiple

Como ves, la clase *Organizador* hasta ahora tiene dos referencias: por un lado, hereda las propiedades de una clase abstracta como lo es *Persona*, y por otra, implementa los métodos de la interfaz *Organizable* que acabamos de crear. **Este es uno de los principales usos de las interfaces: permitir la "herencia" múltiple**, porque como ya te habrás dado cuenta, no se permite heredar (o "extender") clases hijas de varias superclases:

A screenshot of an IDE showing a Java code editor. The code defines a class `Organizador` that extends `Persona` and `Concierto`. A tooltip is displayed over the inheritance line, stating "Class cannot extend multiple classes" with a "2 usages" link. Below the code editor, there's a message "public void mostrarInfo() { 3 usages".

pero en cambio, sí podemos implementar varias interfaces. Por ejemplo, si creamos otra interfaz llamada **Promocionable** como la siguiente:

```
1 public interface Promocionable {  
2  
3     void promocionar();  
4  
5 }
```

Podríamos implementarla también desde la clase *Organizador* de la siguiente manera:

```
class Organizador extends Persona implements Organizable, Promocionable  
private String rol; // Ejemplo: "Seguridad", "Producción", "Prensa"  
  
public Organizador(String nombre, int edad, String rol) {  
    super(nombre, edad);  
    this.rol = rol;  
}  
  
public void mostrarInfo() {  
    super.mostrarInfo();  
    System.out.println("Rol en el festival: " + rol);  
}  
  
public void accederEvento(){  
    System.out.println("Accediendo como Organizador: Coordinando el");  
}  
  
public void organizarEvento() {  
    System.out.println("Organizando...");  
}
```

```

23     public void promocionar() {
24         System.out.println("Promocionando...");
25     }
26 }
27 }
```

Clases abstractas	Interfaces
Se usa para compartir comportamiento común entre clases relacionadas.	Se usa para definir un "contrato" que múltiples clases deben cumplir.
Puede tener métodos abstractos (sin cuerpo) y métodos tradicionales (con cuerpo).	Desde Java 8 puede tener métodos abstractos, métodos default (con cuerpo) y métodos estáticos, aunque será raro que los lleguemos a implementar.
Puede tener atributos	Sólo puede tener constantes estáticas y finales (public static final).
Puede tener constructores, aunque no se pueden instanciar directamente.	No puede tener constructores.
Una clase sólo puede extender una clase abstracta (herencia simple).	Una clase puede implementar múltiples interfaces (herencia múltiple).
Los métodos pueden tener cualquier modificador de acceso (public, protected, private).	Los métodos son implícitamente públicos (aunque desde Java 9 se permiten métodos privados para ayuda interna).

## ¿Cuándo usar cuál?



- Usaremos clase abstractas si hay una relación fuerte entre clases y se comparten atributos y código común. Ejemplo: *Animal* → *Perro, Gato*.
- Usaremos interfaces cuando necesitemos implementar herencia múltiple y definir comportamientos que pueden aplicarse a clases no relacionadas. Ejemplo: *Volador* → *Pájaro, Avión, Superhéroe*.

## Polimorfismo cuando usamos interfaces (instanceOf)

Como ya venimos haciendo desde que conocemos el polimorfismo, uno de los usos principales es el de poder guardar en una misma estructura (normalmente *arrays*) varios tipos de subclases de la siguiente manera:

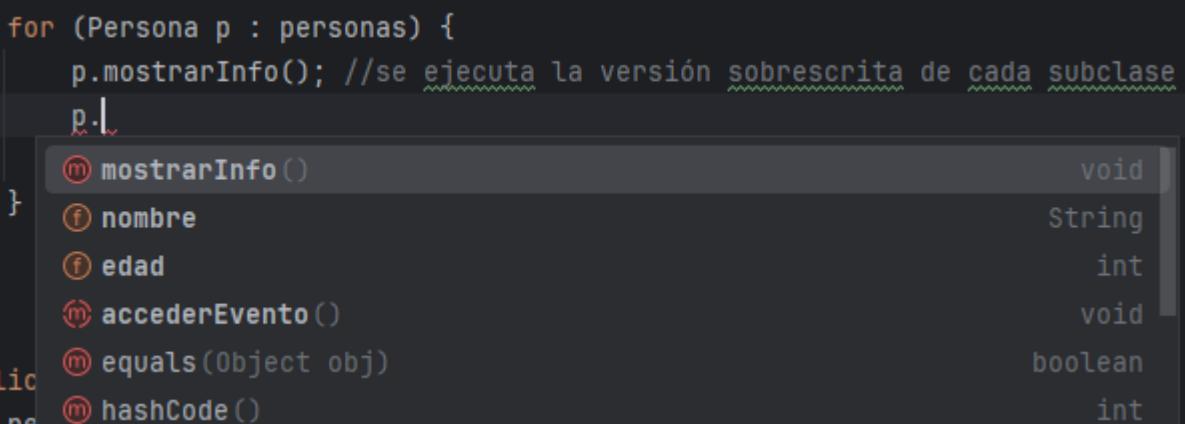
```
1 | Persona[] personas = {  
2 |     new Asistente("Lucas", 18, "VIP"),  
3 |     new Artista("Sofía", 25, "Rock Alternativo"),  
4 |     new Organizador("Martín", 30, "Producción")  
5 | };
```

Con esta estructura, llamábamos a métodos que compartían todas las subclases, como por ejemplo `mostrarInfo()`:

```
1 | for (Persona p : personas) {  
2 |     p.mostrarInfo(); //se ejecuta la versión sobrescrita de cada  
3 | }
```

Pero, ¿qué pasaría si el método a invocar no existiera en todas las "hijas"? Es el caso en que nos encontramos ahora mismo si quisiéramos usar alguno de los que hemos implementado sólo en la clase *Organizador*, usando las interfaces *Promocionable* y *Organizable*.

Como ves, si intentamos acceder a los métodos de la interfaz, no nos está permitido:



The screenshot shows a Java code editor with the following code:

```
for (Persona p : personas) {  
    p.mostrarInfo(); //se ejecuta la versión sobrescrita de cada subclase  
    p. |  
}
```

A code completion dropdown menu is open at the cursor position, showing the following options:

- (m) mostrarInfo() void
- (f) nombre String
- (f) edad int
- (m) accederEvento() void
- (m) equals(Object obj) boolean
- (m) hashCode() int

Para poder invocarlos, deberemos usar la condición `instanceof` con la finalidad de asegurarnos de que un objeto usa una interfaz, y cuando lo hayamos corroborado, entonces

hacer uso de esos métodos (nunca antes):

```
1 |     for (Persona p : personas) {  
2 |         p.mostrarInfo(); //se ejecuta la versión sobrescrita de cada  
3 |  
4 |         if (p instanceof Organizable) {  
5 |             Organizable org = (Organizable) p;  
6 |             org.organizarEvento();  
7 |         }  
8 |     }
```

⚠ Por lo tanto, cuando hagamos polimorfismo se debe verificar la implementación de una interfaz con `instanceof` ANTES de intentar acceder a un método propio de dicha interfaz.

```
1 |     if (objeto instanceof Tipo) {  
2 |         //entonces podemos convertirlo (cast) y usarlo como Tipo  
3 |     }
```

# Ejercicio

---



## Ejercicio: Dispositivos Inteligentes

---

Vamos a administrar dispositivos inteligentes que podríamos tener en nuestra casa.

1. Crea una clase abstracta llamada ***Dispositivo***:

- Atributos: nombre (*String*) y estado (*boolean*, encendido/apagado).
- Métodos:
  - *encender()*: abstracto, cada dispositivo lo implementa de forma diferente.
  - *apagar()*: apaga el dispositivo (*estado = false*) y muestra un mensaje "*nombre + apagado.*". En caso de que se intente volver a apagar y ya esté en estado apagado, se debe mostrar "*nombre + ya está apagado.*".
  - *mostrarEstado()*: muestra el estado actual (encendido o apagado).

2. Crea una interfaz llamada ***ControlRemoto*** que contenga un método *sincronizar()*, para dispositivos que usan control remoto.

3. Otras clases:

- ***Televisor***: extiende *Dispositivo* e implementa *ControlRemoto*.
  - *encender()*: imprime "Encendiendo televisor...". Si ya está encendido, se debe mostrar "El televisor ya está encendido."
  - *sincronizar()*: imprime "Sincronizando televisor con control remoto...".
- ***ParlanteInteligente***: extiende *Dispositivo*.
  - *encender()*: imprime "Activando parlante con comando de voz...". Si ya está encendido, se debe mostrar "El parlante ya está encendido."
- ***AireAcondicionado***: extiende *Dispositivo* e implementa *ControlRemoto*.
  - *encender()*: imprime "Encendiendo aire acondicionado...". Si ya está encendido, se debe mostrar "El aire acondicionado ya está encendido."
  - *sincronizar()*: imprime "Sincronizando aire acondicionado con control remoto...".

4. Programa principal:

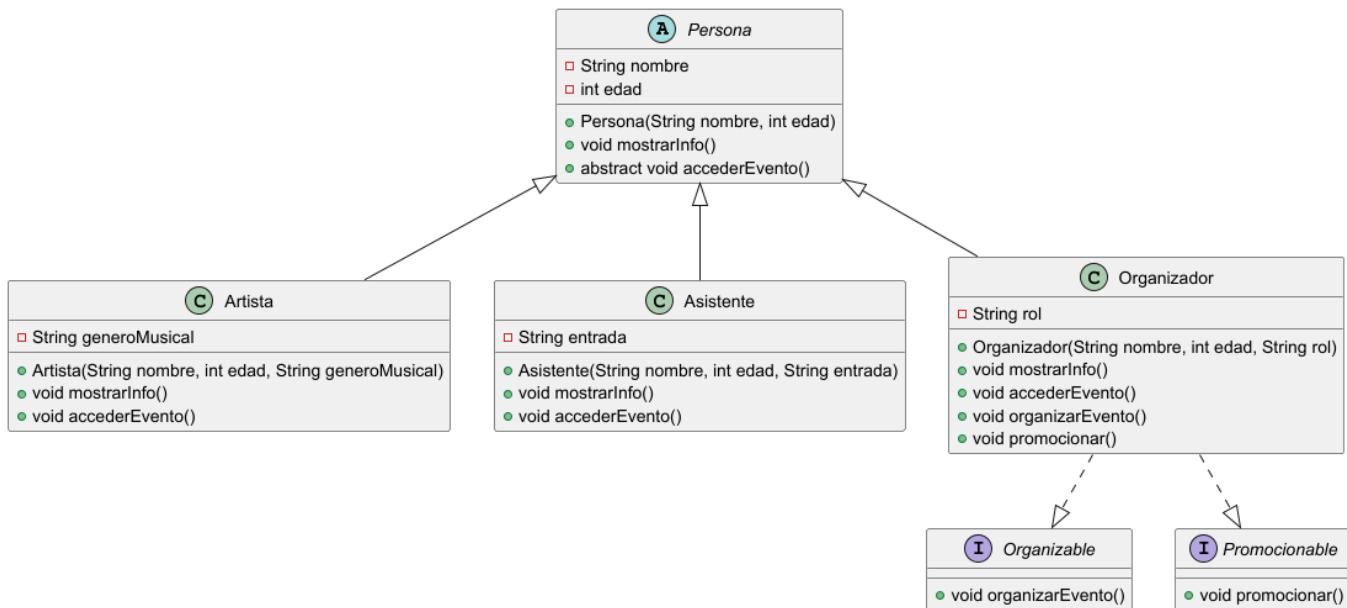
- Crea una lista de dispositivos.
- Recorre la lista para:
  - Encender cada dispositivo.
  - Sincronizar en caso de que un dispositivo implemente *ControlRemoto*.
  - Mostrar el estado de cada dispositivo.

- Apagar cada dispositivo.
-

# Diagrama UML con elementos de tipo interface

---

La forma de representar una implementación de una interfaz es similar a las relaciones de herencia y abstracción anteriores. En este caso, la línea de la flecha que une a la clase con su interfaz será discontinua y la interfaz como tal se representa con el símbolo "I".



## Ejercicio

---

Realiza el diagrama UML del ejercicio sobre Dispositivos Inteligentes realizado anteriormente.

---

# Clases anónimas

---

Las clases anónimas en *Java* son como clases normales, pero sin nombre. Se usan cuando necesitamos una clase "de usar y tirar", generalmente para implementar una interfaz o extender una clase existente rápidamente. **Haremos uso de ellas cuando necesitemos una funcionalidad específica y no queremos crear una clase separada sólo para eso.** Por ejemplo, al usar *listeners* (escuchas) en interfaces gráficas o para pequeñas tareas puntuales.

Imagina que tienes una interfaz llamada ***Saludo*** así:

```
1 | interface Saludo {  
2 |     void decirHola();  
3 | }
```

Podrías hacer esto con una clase normal:

```
1 | class SaludoEspanol implements Saludo {  
2 |     public void decirHola() {  
3 |         System.out.println("¡Hola!");  
4 |     }  
5 | }
```

Y luego usarla así:

```
1 | Saludo saludo = new SaludoEspanol();  
2 | saludo.decirHola();
```

Pero con una clase anónima, nos ahorraremos crear ***SaludoEspanol*** y lo podemos implementar en una sola línea:

```
1 | Saludo saludo = new Saludo() {  
2 |     @Override  
3 |     public void decirHola() {  
4 |         System.out.println("¡Hola!");  
5 |     }  
6 | }
```

```
6 | };  
7 | saludo.dicirHola();
```

¡¿Qué ha pasado ahí!

- Pues para empezar, esta clase no tiene nombre: no hay **class SaludoEspanol**. La clase se ha creado e implementado **Saludo** directamente en el momento de la declaración.
- Se ha usado una sola vez: esto es ideal si no vamos a reutilizar nunca más esa implementación.
- Hemos sobreescrito el método **decirHola()** ahí mismo.

**NOTA:** Fíjate que es lo mismo que nos recomienda hacer *IntelliJ IDEA* cuando intentamos instanciar un objeto del tipo de una clase abstracta:

```
Persona prueba = new Persona(nombre: "Pepe", edad: 50) {  
    @Override 1 usage  
    public void accederEvento() {  
  
    }  
};
```

---

Las clases anónimas en Java también se pueden usar sin interfaz, extendiendo una clase existente. Básicamente, podemos crear una versión "rápida" de una clase y modificar su comportamiento sobre la marcha.

Imagina que tienes una clase **Vehículo** así:

```
1 | class Vehiculo {  
2 |     void hacerSonido() {  
3 |         System.out.println("Brrrrrrrr");  
4 |     }  
5 | }
```

Si quisieras crear una versión específica (por ejemplo, para un tren), podrías hacerlo con una clase anónima así:

```
1 | Vehiculo tren = new Vehiculo() {  
2 |     @Override  
3 |     void hacerSonido() {  
4 |         System.out.println("Chucu chucu");  
5 |     }  
6 | };
```

```
7 | tren.hacerSonido();
```

No hay una clase como Tren, sólo se usa aquí y ya.

---

## ¿Cuándo no usarlas?

- Cuando necesitamos usar la misma lógica en varios lugares (mejor crear una clase normal).
- Si la lógica a implementar es muy compleja se vuelve confuso (mejor hacerlo más claro con una clase normal).

# Mejora con clases anónimas el proyecto de Dispositivos Inteligentes

---

Utilizaremos **clases anónimas** para implementar:

- *Dispositivo* para un nuevo *Proyector*. Al encenderse, debemos mostrar "Encendiendo proyector con ajuste automático de brillo...", En caso de que ya esté encendido, mostraremos "El proyector ya está encendido."
- *ControlRemoto* en el *Proyector* con una sincronización única. Mostraremos "Sincronizando proyector con control remoto de presentación...".
- *Dispositivo* en un *Horno Inteligente* con un método de encendido especial. Mostraremos "Calentando horno con ajuste automático de temperatura...", y en caso de que ya esté encendido, "El horno ya está encendido."

Añadiremos estos nuevos dispositivos en la lista y ejecutaremos sus métodos como en el ejercicio anterior, teniendo en cuenta que el proyector tiene su propia sincronización especial.

---



Obra publicada con Licencia Creative Commons Reconocimiento Compartir igual 4.0  
<http://creativecommons.org/licenses/by-sa/4.0/>

## BONUS. Enums

### BONUS. Tipos enumerados (enum)

En Java, las clases tipo **enum** (enumeración) se utilizan para definir un conjunto fijo de valores. Es decir, **definen tipos de datos que sólo pueden tener un conjunto específico de valores**, como por ejemplo los días de la semana (*LUNES, MARTES, MIÉRCOLES, JUEVES, VIERNES*), meses del año (*ENERO, FEBRERO, MARZO, ABRIL,...*), los estados de un pedido (*RECIBIDO, PROCESANDO, COMPLETADO, ENTREGADO*), los tipos de entradas en un concierto (*GENERAL, PISTA, VIP*), los roles de un organizador (*SEGURIDAD, PRENSA, PROMOTOR*), etc.

```
1 | public enum identificadorValores {  
2 |     //posibles valores  
3 | }
```

Ejemplo para los días de la semana:

```
1 | public enum DiaSemana {  
2 |     LUNES,  
3 |     MARTES,  
4 |     MIÉRCOLES,  
5 |     JUEVES,  
6 |     VIERNES,  
7 |     SÁBADO,  
8 |     DOMINGO;  
9 | }
```

⚠️ **A tener en cuenta:** no podemos crear objetos del tipo enumerado con la sentencia **new**. Cada uno de estos valores (*LUNES, MARTES, etc.*) ya son instancias de *DiaSemana*.

Para usar el conjunto de días de la semana desde un programa principal *main...*

```

1 public class AppTipoEnumerados {
2
3     public static void main(String[] args) {
4
5         DiasSemana diaActual = DiasSemana.JUEVES;
6         System.out.println("Hoy es " + diaActual);
7
8     }
9 }
```

```
"C:\Program Files\Java\jdk-23\bin\java.exe
Hoy es JUEVES
```

Además, tenemos disponible el **método `.values()`**, que devuelve un *array* con todos los valores del *enum* en el orden en que son declarados:

```

1 public class AppTipoEnumerados {
2
3     public static void main(String[] args) {
4
5         DiasSemana diaActual = DiasSemana.JUEVES;
6         System.out.println("Hoy es " + diaActual);
7
8         for (DiasSemana dia : DiasSemana.values()){
9             System.out.print(dia + " ");
10        }
11    }
12 }
```

```
"C:\Program Files\Java\jdk-23\bin\java.exe" "-javaagent:C:\Pro
Hoy es JUEVES
LUNES MARTES MIÉRCOLES JUEVES VIERNES SÁBADO DOMINGO
Process finished with exit code 0
```

## Métodos y constructores en clases de tipo *enum*

Las clases tipo *enum* también pueden tener métodos y constructores definidos. Por ejemplo, con una clase *enum* como la siguiente que contiene las meses del año:

```

1 public enum Mes {
2
3     ENERO(31),
4     FEBRERO(28),
5     MARZO(31),
6     ABRIL(30),
7     MAYO(31),
8     JUNIO(30),
9     JULIO(31),
10    AGOSTO(31),
11    SEPTIEMBRE(30),
12    OCTUBRE(31),
13    NOVIEMBRE(30),
14    DICIEMBRE(31);
15
16    private final int dias;
17
18    Mes(int dias) {
19        this.dias = dias;
20    }
21
22}

```

La clase **Mes** define 12 constantes de enumeración que representan los meses del año. Además, la clase tiene un campo *días* que se utiliza para almacenar el número de días en cada mes. La clase también tiene un constructor que se utiliza para inicializar el campo *días*.

También se podría añadir un método *getDias()* para obtener el número de días en un mes específico:

```

1 public enum Mes {
2     ENERO(31),
3     FEBRERO(28),
4     MARZO(31),
5     ABRIL(30),
6     MAYO(31),
7     JUNIO(30),
8     JULIO(31),
9     AGOSTO(31),
10    SEPTIEMBRE(30),
11    OCTUBRE(31),
12    NOVIEMBRE(30),
13
14}

```

```
15 |     DICIEMBRE(31);  
16 |  
17 |     private final int dias;  
18 |  
19 |     Mes(int dias) {  
20 |         this.dias = dias;  
21 |     }  
22 |  
23 |     public int getDias() {  
24 |         return dias;  
25 |     }  
26 | }
```

Para probar las nuevas funcionalidades, añadiremos la siguiente lógica a nuestro programa principal:

```
1 | public class AppTipoEnumerados {  
2 |  
3 |     public static void main(String[] args) {  
4 |  
5 |         DiasSemana diaActual = DiasSemana.JUEVES;  
6 |         System.out.println("Hoy es " + diaActual);  
7 |  
8 |         for (DiasSemana dia : DiasSemana.values()){  
9 |             System.out.print(dia + " ");  
10 |        }  
11 |  
12 |        System.out.println();  
13 |  
14 |        Mes mesActual = Mes.ENERO;  
15 |        System.out.println(mesActual + " tiene " + mesActual.getDias());  
16 |    }  
17 |}  
18 |}  
19 |}
```

```
"C:\Program Files\Java\jdk-23\bin\java.exe" "-javaagent:C:\Program Fil
Hoy es JUEVES
LUNES MARTES MIÉRCOLES JUEVES VIERNES SÁBADO DOMINGO
ENERO tiene 31 días
```

## De consola a *enum*

En ocasiones, puede que debamos recibir información por consola y almacenarla como tipo enumerado. Para ello, deberemos guardar en un *String* un valor compatible con algún valor de tipo enum en cuestión, y luego hacer una conversión.

Haremos dicha conversión a través del método *valueOf(String cadena)* que contienen todas las clases de tipo *enum*. Este método recibirá un *String* con un posible valor de tipo enumerado.

```
public class AppTipoEnumerados {

    static Scanner teclado = new Scanner(System.in);

    public static void main(String[] args) {

        DiasSemana diaActual = DiasSemana.JUEVES;
        System.out.println("Hoy es " + diaActual);

        for (DiasSemana dia : DiasSemana.values()){
            System.out.print(dia + " ");
        }

        System.out.println();

        Mes mesActual = Mes.ENERO;
        System.out.println(mesActual + " tiene " + mesActual.getDias());

        System.out.println();

        System.out.println("Dime un mes " + Arrays.toString(Mes.values()));
        String mes = teclado.next();
        Mes mes_enum = Mes.valueOf(mes); // mes → deberá ser un tipo en
        System.out.println(mes_enum + " tiene " + mesEnum.getDias() + "
```

```
    }
}
```

```
"C:\Program Files\Java\jdk-23\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2024.2.2\lib\idea_rt.jar=5334,C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2024.2.2\bin"
Hoy es JUEVES
LUNES MARTES MIÉRCOLES JUEVES VIERNES SÁBADO DOMINGO
ENERO tiene 31 días

Dime un mes [ENERO, FEBRERO, MARZO, ABRIL, MAYO, JUNIO, JULIO, AGOSTO, SEPTIEMBRE, OCTUBRE, NOVIEMBRE, DICIEMBRE]
FEBRERO
FEBRERO tiene 28 días
```

**OJO.** Si el valor del *String* no es válido, se producirá un error en ejecución. Además, deberemos tener en cuenta que se diferencia entre mayúsculas y minúsculas.

```
"C:\Program Files\Java\jdk-23\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2024.2.2\lib\idea_rt.jar=5334,C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2024.2.2\bin"
Hoy es JUEVES
LUNES MARTES MIÉRCOLES JUEVES VIERNES SÁBADO DOMINGO
ENERO tiene 31 días

Dime un mes [ENERO, FEBRERO, MARZO, ABRIL, MAYO, JUNIO, JULIO, AGOSTO, SEPTIEMBRE, OCTUBRE, NOVIEMBRE, DICIEMBRE]
enero
Exception in thread "main" java.lang.IllegalArgumentException Create breakpoint : No enum constant org.example.enums.Mes.enero
  at java.base/java.lang.Enum.valueOf(Enum.java:293)
  at org.example.enums.Mes.valueOf(Mes.java:3)
  at org.example.enums.AppTipoEnumerados.main(AppTipoEnumerados.java:28)
```

## Otros métodos útiles en los *enum*

- *ordinal()*: devuelve la posición (índice) de la constante buscada en el *enum*.
- *name()*: devuelve el nombre de la constante como *String*.

```
package org.example.enums;

import java.util.Arrays;
import java.util.Scanner;

public class AppTipoEnumerados {

    static Scanner teclado = new Scanner(System.in);

    public static void main(String[] args) {

        DiasSemana diaActual = DiasSemana.JUEVES;
        System.out.println("Hoy es " + diaActual);

        for (DiasSemana dia : DiasSemana.values()){
            System.out.print(dia + " ");
        }
    }
}
```

```

17     }
18
19     System.out.println();
20
21     Mes mesActual = Mes.ENERO;
22     System.out.println(mesActual + " tiene " + mesActual.getDias());
23
24     System.out.println();
25
26     System.out.println("Dime un mes " + Arrays.toString(Mes.values()));
27     String mes = teclado.next();
28     Mes mes_enum = Mes.valueOf(mes); // mes → deberá ser un tipo enum
29     System.out.println(mes_enum + " tiene " + mes_enum.getDias() + " días");
30
31     System.out.println();
32     int posicion = diaActual.ordinal();
33     System.out.println("Posición para " + diaActual + ": " + posicion);
34     String valor = mesActual.name();
35     System.out.println("Valor del enum para " + mesActual + ": " + valor);
36
37 }
38 }
```

```

"C:\Program Files\Java\jdk-23\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2024.2.2\lib\idea_rt.jar"
Hoy es JUEVES
LUNES MARTES MIÉRCOLES JUEVES VIERNES SÁBADO DOMINGO
ENERO tiene 31 días

Dime un mes [ENERO, FEBRERO, MARZO, ABRIL, MAYO, JUNIO, JULIO, AGOSTO, SEPTIEMBRE, OCTUBRE, NOVIEMBRE, DICIEMBRE]
MAYO
MAYO tiene 31 días

Posición para JUEVES: 3
Valor del enum para ENERO: ENERO
```

## Enums en el proyecto del Festival

---

Podríamos usar *enum* para definir roles de organizadores o tipos de entrada en nuestro *Festival* de la siguiente manera:

```
1 | public enum RolOrganizador {  
2 |     SEGURIDAD, PRODUCCION, PRENSA;  
3 | }
```

```
1 | public enum TipoEntrada {  
2 |     GENERAL, PISTA, VIP;  
3 | }
```

Y podríamos usarlo así en las clases *Organizador* y *Asistente*:

```
package org.example.herencia_concierto;  
  
class Organizador extends Persona implements Organizable, Promocionable  
    private RolOrganizador rol; // Ejemplo: "Seguridad", "Producción",  
  
    public Organizador(String nombre, int edad, RolOrganizador rol) {  
        super(nombre, edad);  
        this.rol = rol;  
    }  
  
    public void mostrarInfo() {  
        super.mostrarInfo();  
        System.out.println("Rol en el festival: " + rol);  
    }  
  
    public void accederEvento(){  
        System.out.println("Accediendo como Organizador: Coordinando el  
    }  
  
    @Override  
    public void organizarEvento() {  
        System.out.println("Organizando...");  
    }
```

```
25     }
26
27     @Override
28     public void promocionar() {
29         System.out.println("Promocionando...");
30     }
}
```

```
1 package org.example.herencia_concierto;
2
3 class Asistente extends Persona {
4     private TipoEntrada entrada; //Tipo de entrada (General, VIP, etc.)
5
6     public Asistente(String nombre, int edad, TipoEntrada entrada) {
7         super(nombre, edad); // Llamamos al constructor de la clase Persona
8         this.entrada = entrada;
9     }
10
11    @Override
12    public void mostrarInfo() {
13        super.mostrarInfo(); //Llamamos al método de la clase Persona
14        System.out.println("Tipo de entrada: " + entrada);
15    }
16
17    public void accederEvento(){
18        System.out.println("Accediendo como Asistente: Buscando asiento");
19    }
20}
```

En el *main*, a la hora de crear los objetos, lo haremos así:

```
1 public class Concierto {
2     public static void main(String[] args) {
3
4         Asistente asistente = new Asistente("Lucas", 18, TipoEntrada.GENERAL);
5         Organizador organizador = new Organizador("Martín", 30, RolOrganizador.ORGANIZADOR);
6
7     }
}
```

Para añadir un poco de "chicha", vamos a asignar un precio a cada entrada:

```
1 public enum TipoEntrada {  
2     GENERAL(50), PISTA(75), VIP(250);  
3  
4     private int precio;  
5  
6     TipoEntrada(int precio) {  
7         this.precio = precio;  
8     }  
9  
10    public int getPrecio() {  
11        return precio;  
12    }  
13}  
14}
```

Y desde el *main*, vamos a imprimir cada tipo de entrada disponibles:

```
1 public class Concierto {  
2     public static void main(String[] args) {  
3  
4         Persona asistente = new Asistente("Lucas", 18, TipoEntrada.GENE  
5         Persona organizador = new Organizador("Martín", 30, RolOrganiza  
6  
7         for (TipoEntrada entrada : TipoEntrada.values()){  
8             System.out.println("Entrada: " + entrada + " con precio " +  
9                 )  
10            }  
11        }
```

```
"C:\Program Files\Java\jdk-23\bin\java.exe" "-javaagent:  
Entrada: GENERAL con precio 50€  
Entrada: PISTA con precio 75€  
Entrada: VIP con precio 250€
```

# Ejercicios

---



## Ejercicio 1

---

Simula un semáforo usando una clase *enum* con:

- Crea un *enum* llamado *Semaforo* con valores secuenciales: *AMARILLO*, *ROJO*, *VERDE*.
- Añade un método *public Semaforo siguiente()* que devuelva el siguiente color del semáforo.
- Simula una secuencia de 10 cambios de semáforo en un *main()*.

```
"C:\Program Files\Java\jdk-
ROJO
VERDE
AMARILLO
ROJO
VERDE
AMARILLO
ROJO
VERDE
AMARILLO
ROJO
```



## Ejercicio 2

---

Representa con *enum* niveles de dificultad en un juego.

- Define un *enum Dificultad* con valores: **FACIL**, **MEDIO**, **DIFICIL**, **EXPERTO**.
- Asigna un multiplicador de puntuación a cada nivel: **FACILx2**, **MEDIOx4**, **DIFICILx8**, **EXPERTOx10**.
- Crea un método *getMultiplicador()* para obtener el valor del multiplicador asociado.
- Simula una partida donde un usuario elige la dificultad y su puntuación final (aleatoriedad) se multiplica según el nivel elegido.

```
"C:\Program Files\Java\jdk-23\bin\java.exe" "-javaagent:C:\Pro  
Introduce la dificultad [FACIL, MEDIO, DIFICIL, EXPERTO]:  
EXPERTO  
Puntuación obtenida=4812  
Puntuación final=48120
```



Obra publicada con Licencia Creative Commons Reconocimiento Compartir igual 4.0  
<<http://creativecommons.org/licenses/by-sa/4.0/>>

## 6.5. Excepciones personalizadas

### 6.5. Excepciones personalizadas

Las excepciones en Java son eventos anómalos que pueden ocurrir durante la ejecución de un programa y que alteran el flujo normal de ejecución. Como ya hemos visto durante el curso, estos eventos representan situaciones inesperadas o errores que deben ser manejados de manera adecuada para garantizar que el programa continúe ejecutándose y evitar interrupciones.

Este manejo de excepciones en una aplicación es muy importante para una buena experiencia de usuario, ya que permite que la aplicación continúe funcionando a pesar de errores predecibles y proporciona información útil al usuario para corregir el problema.

#### Uso de *throw* para lanzar excepciones manualmente

```
1 public class AppExcepciones {  
2  
3     public static void main(String[] args) {  
4  
5         int edad = 10;  
6  
7         if (edad < 18) {  
8             throw new IllegalArgumentException("Debes ser mayor de edad");  
9         }  
10    }  
11 }  
12  
13 }
```

```
"C:\Program Files\Java\jdk-23\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition  
Exception in thread "main" java.lang.IllegalArgumentException: Debes ser mayor de edad.  
at org.example.excepciones.AppExcepciones.main(AppExcepciones.java:11)  
  
Process finished with exit code 1
```

## Creación de Excepciones personalizadas

Hasta ahora, hemos visto excepciones típicas como *NullPointerException*, *ArrayIndexOutOfBoundsException*, *ArithmeticException*... Pero es posible que un programa requiera una excepción específica que todavía no existe. Para crearla, podemos usar una clase nueva que herede de *Exception* o *RuntimeException*:

```
1 | public class MiExcepcion extends Exception {  
2 |     public MiExcepcion(String mensaje) {  
3 |         super(mensaje);  
4 |     }  
5 | }
```

Y podemos lanzarla desde cualquier método, siempre y cuando incluyamos en la cabecera la palabra "**throws**" y el nombre de la excepción que puede darse dentro de él. En nuestro caso, *MiExcepcion*:

```
1 | public class AppExcepciones {  
2 |  
3 |     public static void main(String[] args) throws MiExcepcion {  
4 |  
5 |         int valor = -15;  
6 |         if (valor < 0) {  
7 |             throw new MiExcepcion("El valor no puede ser negativo.");  
8 |         }  
9 |     }  
10 | }  
11 | }  
12 | }
```

```

3 ▶ public class AppExcepciones {
4   ⚡
5 ▶     public static void main(String[] args) throws MiExpcion {} ←
6
7     int valor = -15;
8     if (valor < 0) {
9       throw new MiExpcion(mensaje: "El valor no puede ser negativo.");
10    }
11
12 }
13
14 }
```

```
"C:\Program Files\Java\jdk-23\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2024
Exception in thread "main" org.example.excepciones.MiExpcion Create breakpoint : El valor no puede ser negativo.
at org.example.excepciones.AppExcepciones.main(AppExcepciones.java:15)

Process finished with exit code 1
```

Esta forma de crear excepciones nos vale para controlar varios tipos de errores, ya que la excepción lanzará el mensaje que nosotros le pasemos. Pero si por algún motivo quisiéramos tener excepciones separadas, una para cada tipo de error, podríamos no pasar ningún mensaje como parámetro y asignarlo cuando llamemos al constructor de *MiExpcion* directamente:

```

1 | public class MiExpcion extends Exception {
2 |   public MiExpcion() {
3 |     super("El valor no puede ser negativo.");
4 |   }
5 | }
```

Y desde el método que provoca la excepción, la llamada no enviaría ningún mensaje:

```

1 | public class AppExcepciones {
2 |
3 |   public static void main(String[] args) throws MiExpcion {
4 |
5 |     int valor = -15;
6 |     if (valor < 0) {
7 |       throw new MiExpcion();
8 |     }
9 |
10    }
11
12 }
```

```
}
```

---

Vale... ¿y por qué si lanzo una excepción que ya existe no pongo *throws* en la cabecera y si lanzo una personalizada sí?

La diferencia radica en si la excepción es comprobada (*Checked*) o no comprobada (*Unchecked*). Cuando hemos creado la excepción personalizada hemos heredado de *Exception*, lo que la convierte en una *Checked Exception* y obliga a usar *throws* (o controlarla con un *try-catch*). Este tipo de excepciones son las que no podemos controlar, como acceso a bases de datos caídas (*SQLException*) u otros tipos de errores que como programadores sabemos que pueden pasar pero no sabemos cuándo.

Por otro lado, las *Unchecked Exceptions* heredan de *RuntimeException* (errores que pasan durante la ejecución del programa y que normalmente provoca el usuario, como introducir un formato erróneo de algún dato) y no requieren *throws* en la cabecera del método, porque el compilador no obliga a manejarlas (pero es más que recomendable).

Por lo tanto, **si extendemos *Exception* (*Checked Exception*) debemos usar *throws* en la firma del método. Si extendemos de *RuntimeException* (*Unchecked Exception*) no necesitamos *throws*.**

# Excepciones en nuestro Festival

Para familiarizarnos un poco más con las excepciones personalizadas, vamos a añadir a nuestra clase *Persona* la posibilidad de lanzar una nueva excepción *EdadValidaException* que sea capaz de saltar si intentamos añadir a alguien menor de edad al evento del Festival.

Para ello, crearemos una clase nueva *EdadValidaException* que herede de *Exception*:

```
1 public class EdadValidaException extends Exception {  
2  
3     public EdadValidaException(){  
4         super("Debe ser mayor de edad");  
5     }  
6  
7 }
```

Y desde la clase *Persona*, modificaremos el constructor actual para que la edad se asigne a través del *set()*. Además, recuerda que como el *set* es probable que devuelva una excepción, deberemos incorporar en la cabecera del constructor *throws EdadValidaException*:

```
public Persona(String nombre, int edad) throws EdadValidaException { 3 usages  
    this.nombre = nombre;  
    setEdad(edad);  
}
```

El *set* quedará de la siguiente manera:

```
1 public void setEdad(int edad) throws EdadValidaException {  
2     if (edad<18) throw new EdadValidaException();  
3     this.edad = edad;  
4 }
```

A raíz de este cambio, empezarán a "chillarnos" las subclases *Asistente*, *Artista* y *Organizador* que heredan el constructor de *Persona*. Deberemos añadir en la cabecera del constructor de cada una de ellas la coletilla *throws EdadValidaException*:

```

public Asistente(String nombre, int edad, TipoEntrada entrada) throws EdadValidaException { 1 usage
    super(nombre, edad); // Llamamos al constructor de la clase Persona
    this.entrada = entrada;
}

public Artista(String nombre, int edad, String generoMusical) throws EdadValidaException { 1 usage
    super(nombre, edad);
    this.generoMusical = generoMusical;
}

public Organizador(String nombre, int edad, RolOrganizador rol) throws EdadValidaException {
    super(nombre, edad);
    this.rol = rol;
}

```

Ya estamos listos. Vamos a forzar la excepción desde nuestro programa principal...

```

1 public class Concierto {
2     public static void main(String[] args) throws EdadValidaException {
3
4         Persona artista = new Artista("Sofía", 15, "Rock Alternativo");
5         Persona asistente = new Asistente("Lucas", 18, TipoEntrada.ENTRADA);
6     }
7 }

```

```

"C:\Program Files\Java\jdk-23\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2024.2.
Exception in thread "main" org.example.herencia_concierto.EdadValidaException Create breakpoint : Debe ser mayor de edad
    at org.example.herencia_concierto.Persona.setEdad(Persona.java:19)
    at org.example.herencia_concierto.Persona.<init>(Persona.java:10)
    at org.example.herencia_concierto.Artista.<init>(Artista.java:8)
    at org.example.herencia_concierto.Concierto.main(Concierto.java:7)

```

Comprobamos que funciona, pero en nuestra aplicación no nos interesa obtener un error, solamente que nos informe del error cometido. Por lo tanto, controlaremos nuestra propia excepción con un *try-catch*.

```

public class Concierto {
    public static void main(String[] args) {

        try{
            Persona artista = new Artista("Sofía", 15, "Rock Alternativo");
            Persona asistente = new Asistente("Lucas", 18, TipoEntrada.ENTRADA);
        } catch (EdadValidaException e){
            System.out.println(e.getMessage());
        }
    }
}

```

```
10 |     }
11 | }
```

```
"C:\Program Files\Java\jdk-23\bin\java.exe" "-javaagent:..\..\lib\dvwa-agent.jar"
Debe ser mayor de edad
Entrada: GENERAL con precio 50€
Entrada: PISTA con precio 75€
Entrada: VIP con precio 250€
```

**NOTA:** cuando se controla la excepción de esta manera no es necesario indicar *throws EdadValidaException* en la cabecera del método que vaya a provocarla.



## Ejercicio

---

Modifica de nuevo las clases del paquete del *Festival* para que la nueva excepción personalizada herede de *RuntimeException* en lugar de *Expcion*.

---

# Ejercicios

---



## Simulación de Cajero Automático

---

Vamos a simular un cajero automático con manejo de excepciones personalizadas, permitiendo realizar depósitos, retiros y consultar saldo.

1. Crea una clase **CuentaBancaria** con:

- Un atributo **saldo** privado.
- Métodos para ingresar dinero, retirar dinero y consultar saldo.
- Maneja una excepción **SaldoInsuficienteException** si se intenta retirar más de lo disponible.
- Maneja una excepción **LímiteDiarioException** si se intenta retirar más de 600€.
- Maneja una excepción **DepositoMaximoException** si se intenta ingresar más de 3.000€.
- Para realizar cualquier operación, se debe controlar que el **importe** sea mayor que 0.

2. Crea un programa principal que permita al usuario interactuar con el cajero usando un menú.

```
==== Cajero Automático ===
```

- [1]. Consultar saldo
- [2]. Ingresar dinero
- [3]. Retirar dinero
- [X]. Salir

```
Selecciona una opción: 1
```

```
Saldo disponible: 500.0€
```

```
==== Cajero Automático ===
```

- [1]. Consultar saldo
- [2]. Ingresar dinero
- [3]. Retirar dinero
- [X]. Salir

```
Seleccione una opción: 2
```

```
Introduce el importe a ingresar: 100
```

```
Dinero ingresado con éxito. Saldo actual: 600.0€
```

```
==== Cajero Automático ===
```

- [1]. Consultar saldo
- [2]. Ingresar dinero
- [3]. Retirar dinero
- [X]. Salir

```
Selecciona una opción: X
```



Obra publicada con Licencia Creative Commons Reconocimiento Compartir igual 4.0  
<http://creativecommons.org/licenses/by-sa/4.0/>

## PRÁCTICA 2. SISTEMA DE MODERNIZACIÓN PARA EL MUTXAMEL FC VS REAL MADRID

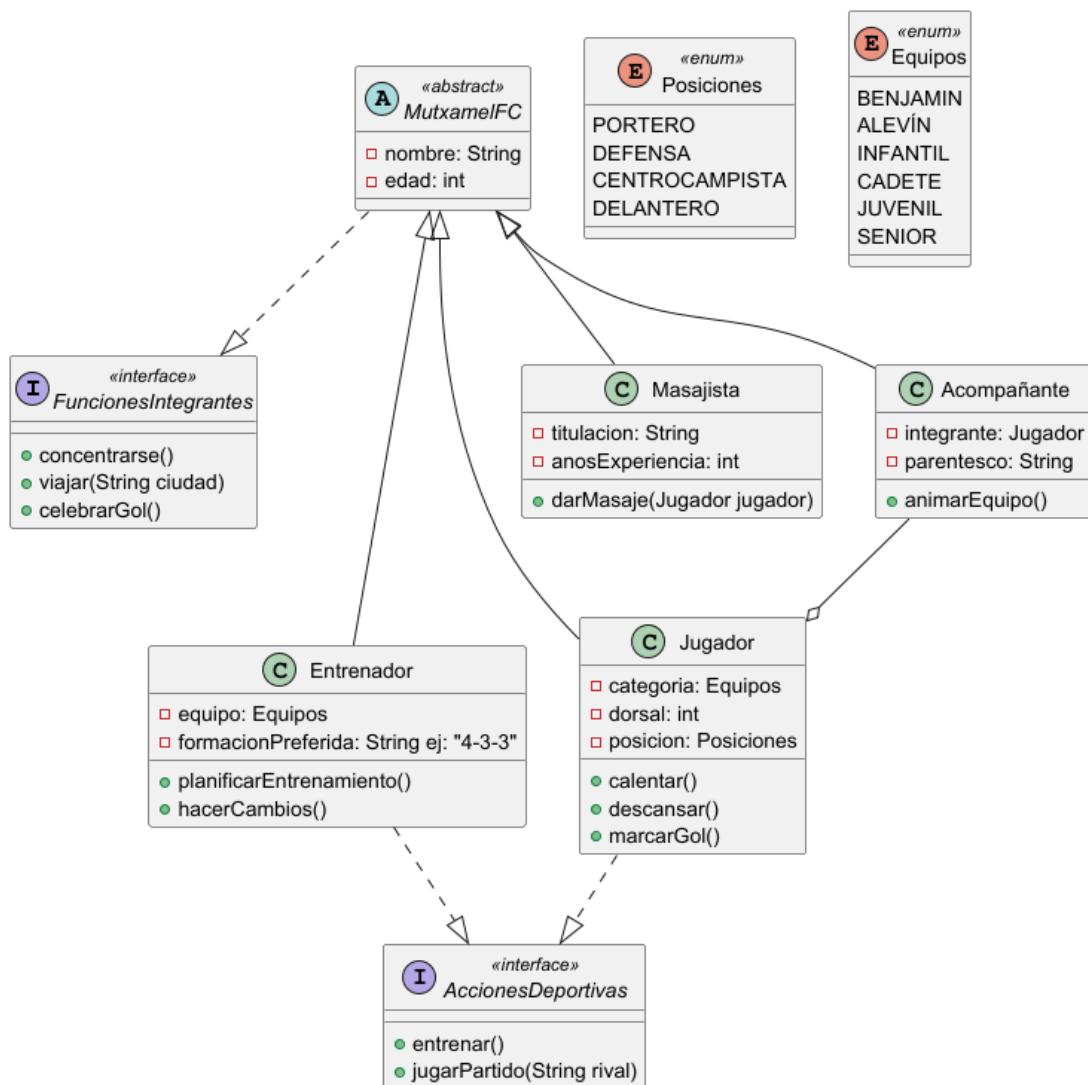


### → INTRODUCCIÓN

El azar ha hecho que al MUTXAMEL FC le toque el Real Madrid en las primeras rondas de la Copa del Rey. Por eso, el club ha emitido un comunicado para contar que va a empezar un proceso de modernización en cuanto a su organización para que no parezca que sea un “equipillo” cualquiera, y ver si así la visita al Bernabéu hace que Florentino se interese un poco por ellos.

### → PROBLEMA A RESOLVER

Vamos a desarrollar un software de mantenimiento del club. Para ello, un analista ha definido cómo sería la estructura de la aplicación a implementar:



Realiza un programa en *Java* que implemente la lógica de la aplicación dada, usando *POO*.

### Condiciones para la construcción de las clases y otras cosas a tener en cuenta

- Inventa (con sentido) el contenido de los métodos a implementar. Por ejemplo, *concentrarse()* podría imprimir “*nombre concentrándose para el partido...*”, *viajar(String ciudad)* podría imprimir “*Viajando a Madrid...*” o *celebrarGol()* mostraría “*Gooooooooooooool*”.
- Debes crear **excepciones personalizadas** para:
  - Si se intenta crear o modificar a un jugador en un mismo *equipo* con un *dorsal* que ya tiene asignado otro jugador.
  - Si se intenta crear o modificar a un entrenador y la *formacionPreferida* no tiene el formato correcto: **N-N-N**.
- Crea un programa principal **AppMutxamelFC** que simule toda la lógica. Usa la siguiente guía de instrucciones:

```
public class AppMutxamelFC {  
  
    public static void main(String[] args) {  
  
        // crear varios jugadores para el equipo SENIOR  
        // crear al entrenador del equipo SENIOR  
        // crear a los masajistas del club  
        // crear a algún acompañante para un par de jugadores  
        // concentrarse()  
        // entrenar()  
        // darMasaje() a algún jugador  
        // viajar() a Madrid  
        // planificarEntrenamiento()  
        // entrenar()  
        // descansar()  
        // calentar()  
        // jugarPartido()  
        // animarEquipo()  
        // hacerCambios()  
        // marcarGol()  
        // celebrarGol()  
        // darMasaje() a varios jugadores  
        // viajar() a Mutxamel  
        // descansar()  
  
    }  
}
```

Ten en cuenta que cada una de las instrucciones han de realizarse por todos aquellos integrantes del equipo que sean capaces de hacerlo.

- Por último, crea una **AppMantenimiento** para dar funcionalidad a un menú como el siguiente:

```
==== App de mantenimiento del MUTXAMEL FC ====
```

```
[1]. Mantenimiento de jugadores
    Dentro podremos añadir jugadores, modificar datos y añadir acompañantes (sólo seniors).
[2]. Mantenimiento de entrenadores (añadir-modificar-salir)
    Dentro podremos añadir entrenadores y modificar sus datos.
[3]. Mantenimiento masajistas (añadir-modificar datos-salir)
    Dentro podremos añadir masajistas y modificar sus datos.
[4]. Consultar equipos
    Dentro se deben listar los tipos de equipos del club y elegir uno.
[X]. Salir
```

```
=====
Selecciona una opción -->
```

En cada una de las subpantallas debe haber una opción de **[X]. Volver al menú principal**.

**NOTA:** Cada pantalla de mantenimiento debe gestionar una lista diferente de jugadores, acompañantes, entrenadores y masajistas.

#### Ejemplo de ejecución:

Por ejemplo, la subpantalla de *Mantenimiento de jugadores*:

```
==== Mantenimiento de Jugadores ===

[1]. Añadir nuevo jugador
[2]. Modificar datos de jugador existente
[3]. Crear acompañantes (sólo seniors)
[X]. Volver al menú principal

=====
Selecciona una opción --> 2
```

```
==== Mantenimiento de Jugadores. Modificar datos de jugador existente ===
```

¿De qué jugador quieras hacer cambios?

```
[0, Nombre: Carlos, Edad: 14, Categoría: INFANTIL, Dorsal: 14, Posición: MEDIOCENTRO]
[1, Nombre: Ismael, Edad: 20, Categoría: SENIOR, Dorsal: 9, Posición: DELANTERO]
[2, Nombre: Carla, Edad: 5, Categoría: BENJAMIN, Dorsal: 20, Posición: DEFENSA]
[3, Nombre: Luis, Edad: 25, Categoría: SENIOR, Dorsal: 7, Posición: DELANTERO]
```

```
=====
Selecciona una opción --> 1
```

```
== Mantenimiento de Jugadores. Modificar datos de jugador existente ==
```

```
Modificando jugador: [Nombre: Ismael, Edad: 20, Categoría: SENIOR, Dorsal: 9, Posición: DELANTERO]
```

```
¿Quéquieres modificar? [nombre,edad,categoría,dorsal,posición]:
```

```
=====
```

```
Selecciona una opción --> dorsal
```

```
== Mantenimiento de Jugadores. Modificar datos de jugador existente ==
```

```
Modificando jugador: [Nombre: Ismael, Edad: 20, Categoría: SENIOR, Dorsal: 9, Posición: DELANTERO]
```

```
=====
```

```
Nuevo dorsal --> 7
```

```
¡Lo siento! Ese dorsal ya está ocupado por un jugador del mismo equipo (SENIOR).
```

## → REALIZACIÓN DE LA PRÁCTICA

Sigue los siguientes pasos para realizar la práctica. ¡Ve guardando tu trabajo de vez en cuando para evitar que se borre el avance si se cierra el editor de textos u ocurre cualquier problema en tu equipo!

1. Programa en Java la aplicación requerida
2. Sube un vídeo ejecutando tu aplicación de mantenimiento, replicando pruebas y explicando todos los comportamientos que hayas implementado.



### ENTREGA

**REALIZA UN INFORME EN PDF CON LA INFO GENERADA Y LOS PASOS SEGUIDOS PARA REALIZAR ESTA PRÁCTICA. EXPLICA TU CÓDIGO. SÚBELO TODO A LA TAREA DE AULES DISPONIBLE.**

**ADEMÁS, PEGA LA URL DE TU PROYECTO EN GITHUB.**

## Programación

### EXAMEN TEMA 6 – POO avanzada

1. (1,5p) Dada la siguiente interfaz correspondiente a la app de una tienda de ropa que todavía está en desarrollo:

```
public interface GestionInventario {  
    void verificarStock();  
}
```

Una de las trabajadoras necesita urgentemente usar el método abstracto ***verificarStock()***, ya que su jefe le ha pedido que actualice el inventario. Pero como la app no está terminada, todavía no hay ninguna clase (*.java*) que implemente la interfaz, y por lo tanto, esa opción no está disponible a través del menú de los trabajadores.

El jefe ha hablado con el informático que les está desarrollando la app, y este le ha dado una solución temporal para que de momento puedan lanzar el método que se necesita.

Indica sobre el siguiente *main* la posible solución que habrá implementado el informático.

```
public class App {  
    public static void main(String[] args) {  
  
    }  
}
```

# Programación

## EXAMEN PRÁCTICO TEMA 6 – POO avanzada

(26/02/2025)



**LEE ATENTAMENTE LAS SIGUIENTES INSTRUCCIONES ANTES DE EMPEZAR:**



- **Recopila en un documento de texto las evidencias de todo el examen. Guárdalo de vez en cuando para no perder el avance de tu trabajo.**
- Cuando termines, **pásalo a PDF y sube el documento creado a la entrega de AULES.**

### PARTE 1: Configuración del entorno (0,5p)

1. Crea un nuevo repositorio llamado “EXAMEN\_UD6\_[nombre]” desde *SourceTree*. El repositorio debe crearse en local y tener su espejo en remoto, por lo tanto, sincronízalo con *GitHub*.

**Pega a continuación la URL a tu nuevo repositorio de GitHub:**

2. Crea un nuevo proyecto Java (*Maven*) con *IntelliJ* -o el IDE que utilices- dentro del repositorio que acabas de crear. Llámalo “EXAMEN UD6”.
3. Crea en el proyecto un paquete nuevo llamado “pizzaexpres”.

**Sincroniza los cambios en tu repositorio remoto.**

### PARTE 2: Resolución de problemas



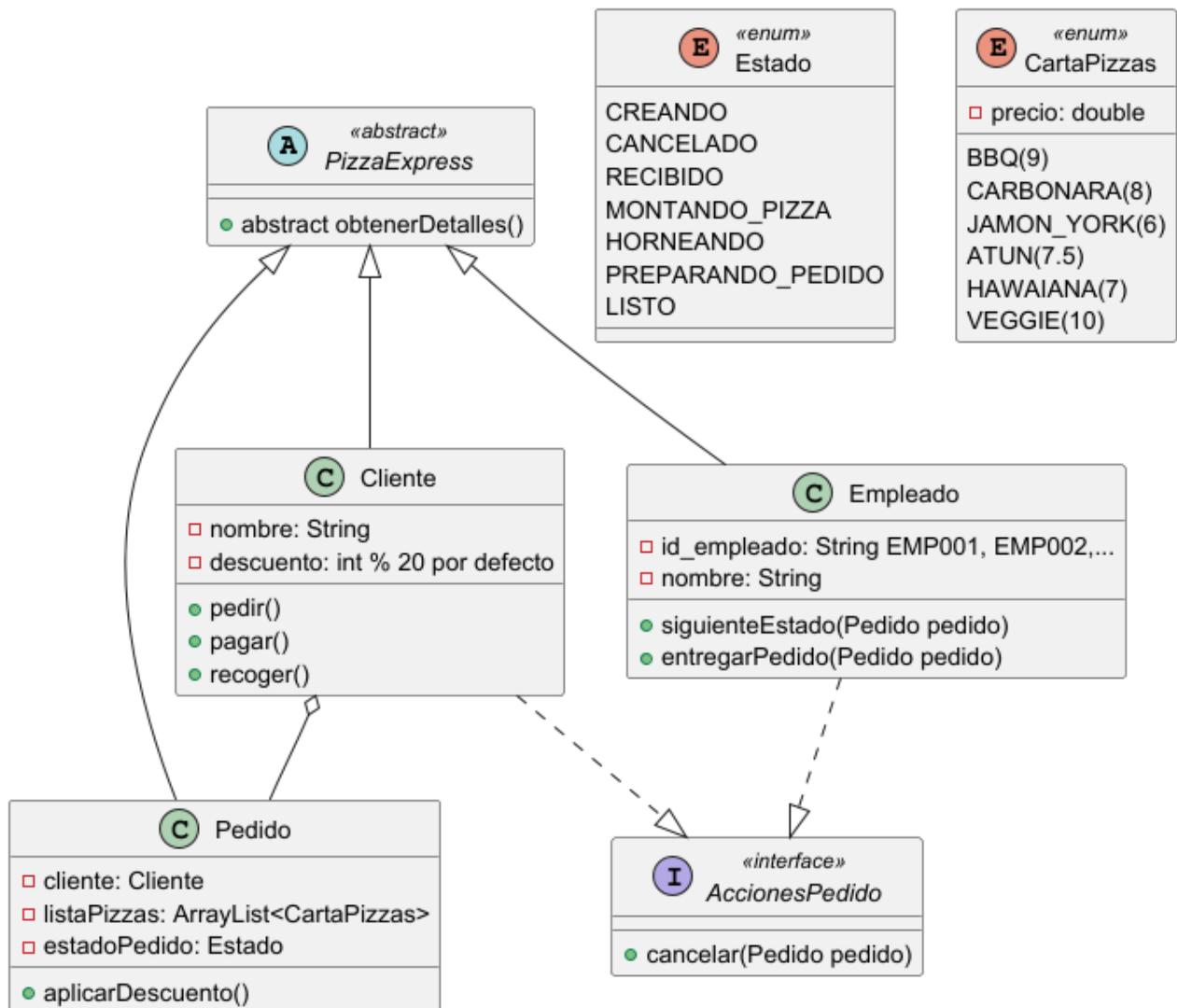
Programa en *Java* la solución a los siguientes ejercicios. Usa el proyecto que te acabas de crear en el apartado anterior.

**Si no has conseguido crearlo correctamente, utiliza alguno de los proyectos que ya tenías para los ejercicios de clase y pega la URL de GitHub del repositorio al que vas a subir los cambios.**

## 1. (7p) App para modernizar la pizzería *PizzaExpress* de Mutxamel

La pizzería más famosa de Mutxamel se ha visto afectada por las ofertas irresistibles de *Domino's Pizza*. Por eso, quieren sacar su propia app para realizar pedidos y obtener descuentos para intentar recuperar a algunos de sus clientes.

El diseño de la app que ha pensado uno de sus trabajadores que estudia 1º DAW en el IES MUTXAMEL es el siguiente:



Realiza un proyecto en Java que implemente la lógica de la aplicación dada, usando POO.

Condiciones para la construcción de las clases y otras cosas a tener en cuenta

- Debes crear una **excepción personalizada** para controlar si un empleado intenta entregar un pedido que todavía no está LISTO.

- Crea un programa principal **AppPizzaExpress** que simule toda la lógica. Usa la siguiente guía de instrucciones:

```
public class AppPizzaExpress {
    public static void main(String[] args) {

        //crear 2 empleados e imprimir info obtenerDetalles()
        //iniciar pedido preguntando nombre
        //responder un nombre
        //crear new Cliente con dicho nombre
        //crear new Pedido para el Cliente creado y estado = CREANDO
        //pedir() preguntar qué le apetece hoy al Cliente y mostrar carta
de pizzas
        //elegir pizza - si existe en la carta, añadir al ArrayList del
pedido. Si no, saltará como inválido por no estar incluida enum
        //decir precio acumulado del pedido y preguntar si se quieren
añadir más pizzas al pedido (S/N)
//\n
        //modificar estado del Pedido a RECIBIDO
        //mostrar "Pedido RECIBIDO" (Estado). Total pedido: importe.
Descuento a aplicar: 20%. Total a pagar: importe-descuento
        //mostrar "Pasa por caja para pagar y recoger tu pedido cuando esté
LISTO. Muchas gracias nombre"
        //avanzar estado a MONTANDO_PIZZA e imprimir
        //avanzar estado a HORNEANDO e imprimir
        //intento de entregar() pedido por alguno de los empleados
        //avanzar estado a PREPARANDO_PEDIDO e imprimir
        //avanzar estado a LISTO e imprimir
        //pagar() Pedido
        //entregar() Pedido
        //recoger()
    }
}
```

Simula también el caso en que se añade más de una pizza. Debe seguir la misma estructura, pero después del paso:

```
//decir precio acumulado del pedido y preguntar si se quieren añadir más
pizzas al pedido (S/N)
```

si se elige “S”, se debe volver a mostrar la carta y dar opción de añadir otra pizza al cliente.

**Ve a ver las distintas salidas de ejemplo.**

## 2. (1p) Pruebas.

Lanza tu programa y explica las pruebas que vas haciendo conforme vas implementando código.

### Salida de ejemplo 1: camino feliz con una sola pizza en el pedido

```
"C:\Program Files\Java\jdk-23\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2023.2.1\lib\javassist-agent.jar" -Dfile.encoding=UTF-8
*** BIENVENIDO A LA PIZZERÍA POPEYE DE MUTXAMEL ***
Empleado EMP001: Carlos
Empleado EMP002: Sabrina
Haz tu pedido...¿cómo te llamas?
Patri
¿Qué te apetece hoy, Patri?
=====CARTA=====
BBQ: 9.0€
CARBONARA: 8.0€
JAMON_YORK: 6.0€
ATUN: 7.5€
HAWAIANA: 7.0€
VEGGIE: 10.0€
=====
veggie
El precio actual de la cuenta es de 10.0€. ¿Quieres añadir otra pizza a tu pedido? [S/N]
n
Pedido RECIBIDO. Total pedido: 10.0€.
Descuento a aplicar: 20%. Total importe a pagar: 8.0€
Pasa por caja para pagar y recoger tu pedido cuando esté listo. Muchas gracias Patri.
MONTANDO_PIZZA...
HORNEANDO...
El pedido todavía no está listo para entregarse. Estado: HORNEANDO
```

```
PREPARANDO_PEDIDO...
LISTO!!
Patri realizando pago en caja...
Entregando pedido a Patri
Pedido recogido por Patri

Process finished with exit code 0
```

### Salida de ejemplo 2: camino feliz con varias pizzas en el pedido

```
*** BIENVENIDO A LA PIZZERÍA POPEYE DE MUTXAMEL ***
Empleado EMP001: Carlos
Empleado EMP002: Sabrina
Haz tu pedido...¿cómo te llamas?
Patri
¿Qué te apetece hoy, Patri?
=====CARTA=====
BBQ: 9.0€
CARBONARA: 8.0€
JAMON_YORK: 6.0€
ATUN: 7.5€
HAWAIANA: 7.0€
VEGGIE: 10.0€
=====
veggie
```

```
El precio actual de la cuenta es de 10.0€. ¿Quieres añadir otra pizza a tu pedido? [S/N]
s
¿Qué te apetece hoy, Patri?
=====CARTA=====
BBQ: 9.0€
CARBONARA: 8.0€
JAMON_YORK: 6.0€
ATUN: 7.5€
```

```
HAWAIANA: 7.0€
VEGGIE: 10.0€
=====
atun
El precio actual de la cuenta es de 17.5€. ¿Quieres añadir otra pizza a tu pedido? [S/N]
n
Pedido RECIBIDO. Total pedido: 17.5€.
Descuento a aplicar: 20%. Total importe a pagar: 14.0€
Pasa por caja para pagar y recoger tu pedido cuando esté listo. Muchas gracias Patri.
MONTANDO_PIZZA...
HORNEANDO...
INTENTO DE ENTREGA. El pedido todavía no está listo para entregarse. Estado: HORNEANDO
PREPARANDO_PEDIDO...
LISTO!!
Patri realizando pago en caja...
Entregando pedido a Patri
Pedido recogido por Patri

Process finished with exit code 0
```

### Salida de ejemplo 3: introducimos pizza que no existe y camino feliz con una pizza

```
*** BIENVENIDO A LA PIZZERÍA POPEYE DE MUTXAMEL ***
Empleado EMP001: Carlos
Empleado EMP002: Sabrina
Haz tu pedido...¿cómo te llamas?
Patricia
¿Qué te apetece hoy, Patricia?
=====CARTA=====
BBQ: 9.0€
CARBONARA: 8.0€
JAMON_YORK: 6.0€
ATUN: 7.5€
HAWAIANA: 7.0€
VEGGIE: 10.0€
=====
holá
El producto escogido no está disponible.
El precio actual de la cuenta es de 0.0€. ¿Quieres añadir otra pizza a tu pedido? [S/N]
s
¿Qué te apetece hoy, Patricia?
```

```

=====CARTA=====
BBQ: 9.0€
CARBONARA: 8.0€
JAMON_YORK: 6.0€

ATUN: 7.5€
HAWAIANA: 7.0€
VEGGIE: 10.0€
=====
veggie
El precio actual de la cuenta es de 10.0€. ¿Quieres añadir otra pizza a tu pedido? [S/N]
n
Pedido RECIBIDO. Total pedido: 10.0€.
Descuento a aplicar: 20%. Total importe a pagar: 8.0€
Pasa por caja para pagar y recoger tu pedido cuando esté listo. Muchas gracias Patricia.
MONTANDO_PIZZA...
HORNEANDO...
INTENTO DE ENTREGA. El pedido todavía no está listo para entregarse. Estado: HORNEANDO
PREPARANDO_PEDIDO...
LISTO!!
Patricia realizando pago en caja...
Entregando pedido a Patricia
Pedido recogido por Patricia

Process finished with exit code 0

```

#### Salida de ejemplo 4: introducimos pizza que no existe y terminamos el pedido sin ningún producto añadido

```

"C:\Program Files\Java\jdk-23\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2023.2.1\lib\idea_rt.jar=5434,C:\Program Files\JetBrains\IntelliJ IDEA 2023.2.1\bin"
*** BIENVENIDO A LA PIZZERÍA POPEYE DE MUTXAMEL ***
Empleado EMP001: Carlos
Empleado EMP002: Sabrina
Haz tu pedido...¿cómo te llamas?
Patricia
¿Qué te apetece hoy, Patricia?
=====CARTA=====
BBQ: 9.0€
CARBONARA: 8.0€
JAMON_YORK: 6.0€
ATUN: 7.5€
HAWAIANA: 7.0€
VEGGIE: 10.0€
=====
hola
El producto escogido no está disponible.
El precio actual de la cuenta es de 0.0€. ¿Quieres añadir otra pizza a tu pedido? [S/N]
n
NO has añadido ningún producto. Pedido finalizado.

Process finished with exit code 0

```

# Programación

## EXAMEN PRÁCTICO TEMA 6 – POO avanzada

(27/02/2025)



**LEE ATENTAMENTE LAS SIGUIENTES INSTRUCCIONES ANTES DE EMPEZAR:**



- **Recopila en un documento de texto las evidencias de todo el examen. GUÁRDALO DE VEZ EN CUANDO PARA NO PERDER EL AVANCE DE TU TRABAJO.**
- Cuando termines, **pásalo a PDF y sube el documento creado a la entrega de AULES.**

### PARTE 1: Configuración del entorno (0,5p)

1. Crea un nuevo repositorio llamado “EXAMEN\_UD6\_[nombre]” desde SourceTree. El repositorio debe crearse en local y tener su espejo en remoto, por lo tanto, sincronízalo con GitHub.

**Pega a continuación la URL a tu nuevo repositorio de GitHub:**

2. Crea un nuevo proyecto Java (Maven) con IntelliJ -o el IDE que utilices- dentro del repositorio que acabas de crear. Llámalo “EXAMEN UD6”.
3. Crea en el proyecto un paquete nuevo llamado “mutxaAwards”.

**Sincroniza los cambios en tu repositorio remoto.**

### PARTE 2: Resolución de problemas

#### *MutxaAwards*



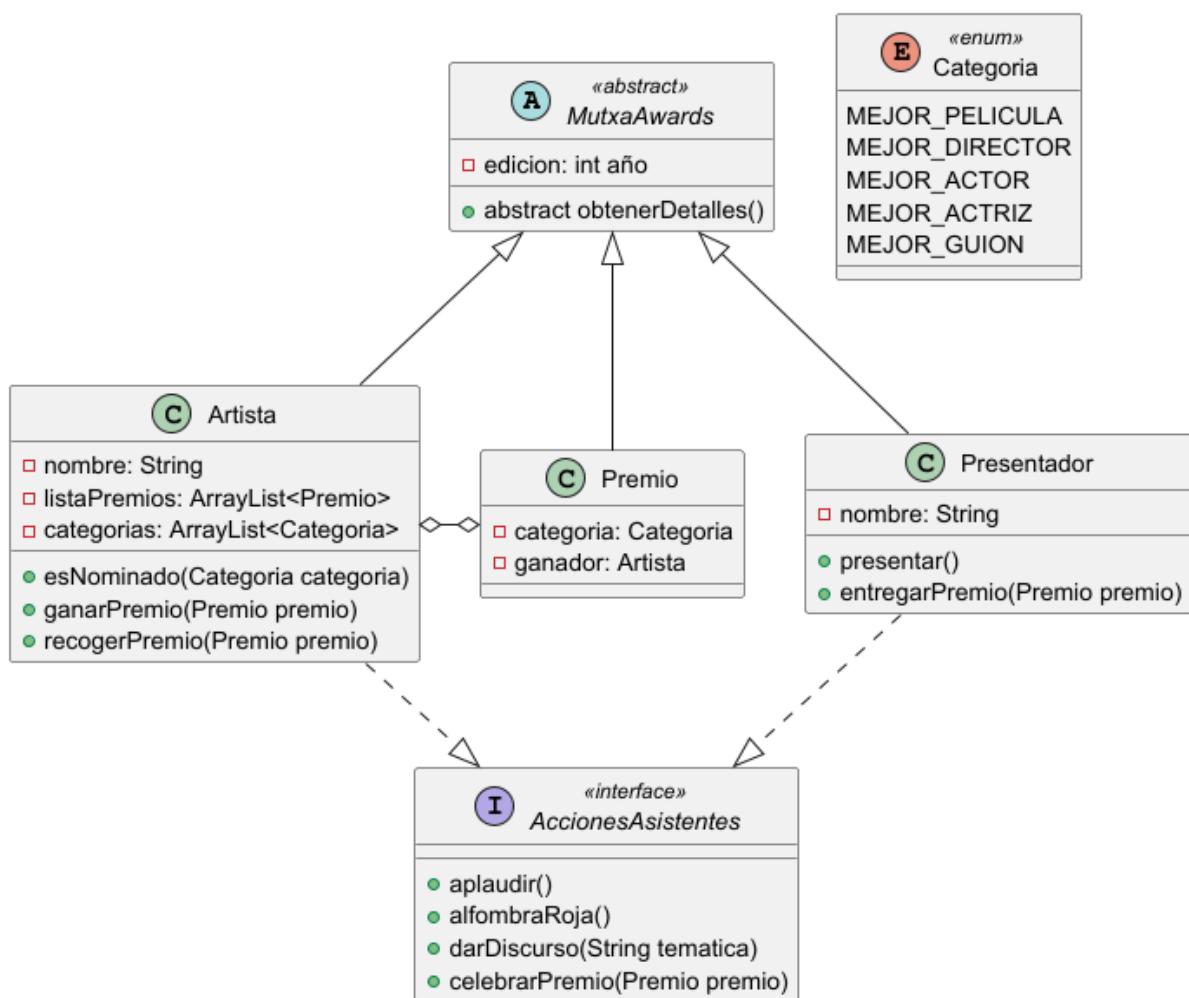
Programa en Java la solución al siguiente problema. Usa el proyecto que te acabas de crear en el apartado anterior.

**Si no has conseguido crearlo correctamente, utiliza alguno de los proyectos que ya tenías para los ejercicios de clase y pega la URL de GitHub del repositorio al que vas a subir los cambios.**

## 1. (7p) App para gestionar los premios cinematográficos de Mutxamel

El Ayuntamiento de Mutxamel ha decidido reconocer a sus vecin@s más artistas su trabajo. Por eso, a partir de 2025, va a organizar unos premios cinematográficos al estilo de los *Goya* o los *Oscars* (salvando las distancias).

Para poder llevarlos a cabo, va a ser necesaria una app que gestione todo lo que ocurre durante la ceremonia. A uno de los alumnos de 1º DAM que está haciendo sus prácticas en el mismo Ayuntamiento se le ha ocurrido el siguiente diseño para implementar la aplicación:



Realiza un proyecto en Java que implemente la lógica de la aplicación dada, usando POO.

### Condiciones para la construcción de las clases y otras cosas a tener en cuenta

- Debes crear una **excepción personalizada** para que no se le permita a un Artista recoger un premio que no ha ganado.
- Crea un programa principal **AppMutxaAwards** que simule toda la lógica de una gala. Usa la siguiente guía de instrucciones:

```

public class AppMutxaAwards {

    public static void main(String[] args) {

        // crear 1 presentador para la edición de 2025 y obtenerDetalles()
        // crear 5 artistas para la edición de 2025
        // todos los asistentes pasan por la alfombra roja
        // todos los asistentes aplauden
        // el presentador empieza a presentar la gala
        // el presentador da un discurso "para solidarizarse con los
afectados de la DANA"
        // se crea el premio (new) con la categoría MEJOR_ACTOR
        // 4 de los artistas son nominados a la categoría MEJOR_ACTOR y
actualizan su lista de categorías
        // 1 de los artistas nominados gana el premio y actualiza su lista
de premios ganados
        // se actualiza el atributo ganador en el premio MEJOR_ACTOR
        // todos los asistentes aplauden y celebran el premio
        // 1 de los artistas que estaba nominado intenta recoger el premio
aunque no lo haya ganado
        // el artista ganador recoge el premio
        // el presentador entrega el premio
        // el artista ganador realiza un discurso "sobre el esfuerzo
realizado para rodar la película"
        // todos los asistentes aplauden
        // el presentador da un discurso "para introducir el siguiente
premio: MEJOR_PELICULA"
        // se crea el premio con la categoría MEJOR_PELICULA
        // 4 de los artistas son nominados a la categoría MEJOR_PELICULA y
actualizan su lista de categorías
        // 1 de los artistas nominados gana el premio y actualiza su lista
de premios ganados
        // se actualiza el ganador en el premio MEJOR_PELICULA
        // todos los asistentes celebran el premio
        // el artista ganador recoge el premio
        // el presentador entrega el premio
        // el artista ganador realiza un discurso "sobre lo duro que ha
sido conseguir financiación"
        // el presentador da un discurso "para despedir la gala"
        // mostrarDetalles() de cada artista después de la gala
    }
}

```

## 2. (1p) Pruebas.

Lanza tu programa y explica las pruebas que vas haciendo conforme vas implementando código.

### Ejecución de ejemplo (sin parte amarilla)

```
"C:\Program Files\Java\jdk-23\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community  
Presentador: Roberto Leal  
-----  
Presentador Roberto Leal pasando por la alfombra roja...  
Artista Javier Bardem pasando por la alfombra roja...  
Artista Antonio Banderas pasando por la alfombra roja...  
Artista Maxi Iglesias pasando por la alfombra roja...  
Artista Úrsula Corberó pasando por la alfombra roja...  
Artista Hiba Abouk pasando por la alfombra roja...
```

```
-----  
Presentador Roberto Leal aplaudiendo... PLAS PLAS PLAS  
El asistente Javier Bardem aplaudiendo... PLAS PLAS PLAS  
El asistente Antonio Banderas aplaudiendo... PLAS PLAS PLAS  
El asistente Maxi Iglesias aplaudiendo... PLAS PLAS PLAS  
El asistente Úrsula Corberó aplaudiendo... PLAS PLAS PLAS  
El asistente Hiba Abouk aplaudiendo... PLAS PLAS PLAS  
-----  
Roberto Leal presentando la gala...  
El presentador da un discurso para solidarizarse con los afectados de la DANA  
-----  
Introduciendo premio MEJOR_ACTOR  
-----  
Artista Javier Bardem nominado a MEJOR_ACTOR  
Artista Maxi Iglesias nominado a MEJOR_ACTOR  
Artista Antonio Banderas nominado a MEJOR_ACTOR  
Artista Hiba Abouk nominado a MEJOR_ACTOR  
-----  
Artista Maxi Iglesias ha ganado el premio MEJOR_ACTOR
```

```
-----  
Presentador Roberto Leal aplaudiendo... PLAS PLAS PLAS  
El asistente Javier Bardem aplaudiendo... PLAS PLAS PLAS  
El asistente Antonio Banderas aplaudiendo... PLAS PLAS PLAS  
El asistente Maxi Iglesias aplaudiendo... PLAS PLAS PLAS  
El asistente Úrsula Corberó aplaudiendo... PLAS PLAS PLAS  
El asistente Hiba Abouk aplaudiendo... PLAS PLAS PLAS  
-----  
El presentador Roberto Leal está celebrando el premio MEJOR_ACTOR otorgado a Maxi Iglesias  
El artista Javier Bardem está celebrando el premio MEJOR_ACTOR otorgado a Maxi Iglesias  
El artista Antonio Banderas está celebrando el premio MEJOR_ACTOR otorgado a Maxi Iglesias  
El artista Maxi Iglesias está celebrando el premio MEJOR_ACTOR otorgado a Maxi Iglesias  
El artista Úrsula Corberó está celebrando el premio MEJOR_ACTOR otorgado a Maxi Iglesias  
El artista Hiba Abouk está celebrando el premio MEJOR_ACTOR otorgado a Maxi Iglesias  
-----  
El artista Antonio Banderas es un tramposo y ha intentado recoger el premio, pero no ha ganado.  
-----  
El artista Maxi Iglesias sube a recoger el premio MEJOR_ACTOR  
El presentador Roberto Leal está entregando el premio MEJOR_ACTOR a Maxi Iglesias  
El artista Maxi Iglesias está dando un discurso sobre el esfuerzo realizado para rodar la película
```

```
-----  
El presentador da un discurso para despedir la gala  
-----  
Artista: Javier Bardem, con 1 nominaciones a : [MEJOR_ACTOR] y 0 premios ganados  
Artista: Antonio Banderas, con 1 nominaciones a : [MEJOR_ACTOR] y 0 premios ganados  
Artista: Maxi Iglesias, con 1 nominaciones a : [MEJOR_ACTOR] y 1 premios ganados MEJOR_ACTOR  
Artista: Úrsula Corberó, con 0 nominaciones a : [] y 0 premios ganados  
Artista: Hiba Abouk, con 1 nominaciones a : [MEJOR_ACTOR] y 0 premios ganados  
  
Process finished with exit code 0
```



