Ejercicio 2

Ejercicio 2.1

Codigo original:

```
public class EmpleadoTemporario {

      public String nombre;

      public String apellido;

      public double sueldoBasico = 0;

      public double horasTrabajadas = 0;

      public int cantidadHijos = 0;

      // ......

   public double sueldo() {

   return this.sueldoBasico

   (this.horasTrabajadas * 500)

   (this.cantidadHijos * 1000)

   (this.sueldoBasico * 0.13);

   }
}

public class EmpleadoPlanta {

    public String nombre;

    public String apellido;

    public double sueldoBasico = 0;

    public int cantidadHijos = 0;
```

```
    // ......
    public double sueldo() {
        return this.sueldoBasico
+ (this.cantidadHijos * 2000)
- (this.sueldoBasico * 0.13);
    }
}

public class EmpleadoPasante {
    public String nombre;
    public String apellido;
    public double sueldoBasico = 0;
    // ......
    public double sueldo() {
        return this.sueldoBasico - (this.sueldoBasico * 0.13);
    }
}
```

(1) Code Smell: Codigo duplicado

```
Las clases EmpleadoTemporario, EmpleadoPlanta y EmpleadoPasante comparten
atributos y metodo
```

Refactoring: Crear superclase Empleado

Solucion:

```
public class Empleado () {}

public class EmpleadoTemporario extends Empleado {

    public String nombre;

    public String apellido;

    public double sueldoBasico = 0;

    public double horasTrabajadas = 0;

    public int cantidadHijos = 0;

    // ......

 public double sueldo() {

 return this.sueldoBasico

 (this.horasTrabajadas * 500)

 (this.cantidadHijos * 1000)

 (this.sueldoBasico * 0.13);

 }

}


public class EmpleadoPlanta extends Empleado {

  public String nombre;

  public String apellido;

  public double sueldoBasico = 0;

  public int cantidadHijos = 0;

  // ......

  public double sueldo() {

      return this.sueldoBasico
```

```
+ (this.cantidadHijos * 2000)

- (this.sueldoBasico * 0.13);

  }
}

public class EmpleadoPasante extends Empleado {

  public String nombre;

  public String apellido;

  public double sueldoBasico = 0;

  // ......

  public double sueldo() {

      return this.sueldoBasico - (this.sueldoBasico * 0.13);

  }
}
```

(2) Code Smell: Codigo duplicado

Se siguen repitiendo los atributos y metodos

Refactoring: Subir atributos comunes

```
public class Empleado () {}
    public String nombre;
    public String apellido;
    public double sueldoBasico = 0 ;

public class EmpleadoTemporario extends Empleado {

    public double horasTrabajadas = 0;

    public int cantidadHijos = 0;

    // ......

 public double sueldo() {
```

```
 return this.sueldoBasico

 (this.horasTrabajadas * 500)

 (this.cantidadHijos * 1000)

 (this.sueldoBasico * 0.13);

 }
}

public class EmpleadoPlanta extends Empleado {

  public int cantidadHijos = 0;

  // ......

  public double sueldo() {

      return this.sueldoBasico

+ (this.cantidadHijos * 2000)

- (this.sueldoBasico * 0.13);

  }
}
public class EmpleadoPasante extends Empleado {
  // ......
  public double sueldo() {

      return this.sueldoBasico - (this.sueldoBasico * 0.13);
  }
}
```

(3) Code Smell: Rompe el encapsulamiento del objeto

Al tener los atributos publicos en todas las clases, permitimos que desde afuera de la clase, puedan modificar la estructura interna.

Refactoring: Atributos con acceso privado

```
public class Empleado () {}
    private String nombre;
```

```
    private String apellido;
    private double sueldoBasico = 0 ;

public class EmpleadoTemporario extends Empleado {
    private double horasTrabajadas = 0;
    private int cantidadHijos = 0;
    // ......
 public double sueldo() {
 return this.sueldoBasico
 (this.horasTrabajadas * 500)
 (this.cantidadHijos * 1000)
 (this.sueldoBasico * 0.13);
 }
}
public class EmpleadoPlanta extends Empleado {
    private int cantidadHijos = 0;
  // ......
  public double sueldo() {

      return this.sueldoBasico

+ (this.cantidadHijos * 2000)

- (this.sueldoBasico * 0.13);
  }
}

public class EmpleadoPasante extends Empleado {

  // ......

  public double sueldo() {

      return this.sueldoBasico - (this.sueldoBasico * 0.13);

  }

}
```

(4)Code Smell: Falta de polimorfismo

Refactoring: Crear metodo abstracto en clase Empleado

```
public class Empleado () {}
```

```
    private String nombre;
    private String apellido;
    private double sueldoBasico = 0 ;
    public abstract double  sueldo();
public class EmpleadoTemporario extends Empleado {
    private double horasTrabajadas = 0;
    private int cantidadHijos = 0;
    // ......
@Override
 public double sueldo() {
 return this.sueldoBasico
 (this.horasTrabajadas * 500)
 (this.cantidadHijos * 1000)
 (this.sueldoBasico * 0.13);
 }
}
public class EmpleadoPlanta extends Empleado{
    private int cantidadHijos = 0;
  // ......
  @Override
  public double sueldo() {
      return this.sueldoBasico
+ (this.cantidadHijos * 2000)
- (this.sueldoBasico * 0.13);
```

```
  }
}


public class EmpleadoPasante  extends Empleado {
  // ......
  @Override
  public double sueldo() {
      return this.sueldoBasico - (this.sueldoBasico * 0.13);
  }
}
```

(5) Code Smell: nombre metodo abstracto poco autoexplicativo

Refactoring: Renombrar metodo

```
public class Empleado () {}
    private String nombre;
    private String apellido;
    private double sueldoBasico = 0 ;


    public abstract double calcularSueldo ();


public class EmpleadoTemporario extends Empleado {
    private double horasTrabajadas = 0;
    private int cantidadHijos = 0;
    // ......
```

```
@Override
 public double calcularSueldo() {
 return this.sueldoBasico
 (this.horasTrabajadas * 500)
 (this.cantidadHijos * 1000)
 (this.sueldoBasico * 0.13);
 }
}


public class EmpleadoPlanta extends Empleado{
    private int cantidadHijos = 0;
  // ......
  @Override
  public double calcularSueldo() {
      return this.sueldoBasico
+ (this.cantidadHijos * 2000)
- (this.sueldoBasico * 0.13);
  }
}


public class EmpleadoPasante  extends Empleado {
  // ......
  @Override
```

```
  public double calcularSueldo() {

      return this.sueldoBasico - (this.sueldoBasico * 0.13);

  }

}
```

Ejercicio 2.2

Codigo original

```
public class Juego {

  // ......
  public void incrementar(Jugador j) {
      j.puntuacion = j.puntuacion + 100;
  }
  public void decrementar(Jugador j) {
      j.puntuacion = j.puntuacion - 50;
  }
public class Jugador {
  public String nombre;
  public String apellido;
  public int puntuacion = 0;
}
}
```

(1) Code smell: Clase inapropiada

La clase juego accede demasiado a los detalles internos de otra clase

Refactoring: mover metodos, eliminar clase

```
public class Jugador {

  public String nombre;

  public String apellido;

  public int puntuacion = 0;
```

```
  public void incrementar(Jugador j) {
    j.puntuacion = j.puntuacion + 100;
}
public void decrementar(Jugador j) {
    j.puntuacion = j.puntuacion - 50;
}
}
```

(2) Code smell: Rompe el encapsulamiento

La clase Jugador tiene atributos con visibilidad publica, por lo que permite que se puedan modificar sus atributos desde fuera de la clase.

Refactoring: Cambiar el modificador de acceso de los atributos, agregar metodos setters y getters

```
public class Jugador {
  private String nombre;
  private String apellido;
  private int puntuacion = 0;

  public void incrementar(Jugador j) {
    j.puntuacion = j.puntuacion + 100;
}
public void decrementar(Jugador j) {
    j.puntuacion = j.puntuacion - 50;
```

```
}

public String getNombre() {
  return this.nombre;
}

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

public String getApellido() {
  return this.apellido;
}

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

public int getPuntuacion() {
  return this.puntuacion;
}

public void setPuntuacion(int puntuacion) {
```

```
  this.puntuacion = puntuacion;
}
}
```

(4) Code smell: Nombre de metodo poco autoexplicativo

Refactoring: Renombrar metodos

```
public class Jugador {
  private String nombre;
  private String apellido;
  private int puntuacion = 0;

  public void incrementarPuntuacionEn100(Jugador j) {
    j.puntuacion = j.puntuacion + 100;
}
public void decrementarPuntuacionEn50(Jugador j) {
    j.puntuacion = j.puntuacion - 50;
}
public String getNombre() {
  return this.nombre;
}

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

```
}


public String getApellido() {

  return this.apellido;

}


public void setApellido(String apellido) {

  this.apellido = apellido;

}


public int getPuntuacion() {

  return this.puntuacion;

}


public void setPuntuacion(int puntuacion) {

  this.puntuacion = puntuacion;

}

}
```

2.3

Codigo original

```
/**
* Retorna los últimos N posts que no pertenecen al usuario user
*/
```

```java
public List<Post> ultimosPosts(Usuario user, int cantidad) {

    List<Post> postsOtrosUsuarios = new ArrayList<Post>();

    for (Post post : this.posts) {

        if (!post.getUsuario().equals(user)) {

            postsOtrosUsuarios.add(post);

        }

    }

   // ordena los posts por fecha

   for (int i = 0; i < postsOtrosUsuarios.size(); i++) {

       int masNuevo = i;

       for(int j= i +1; j < postsOtrosUsuarios.size(); j++) {

           if (postsOtrosUsuarios.get(j).getFecha().isAfter(

     postsOtrosUsuarios.get(masNuevo).getFecha())) {

              masNuevo = j;

           }

       }

      Post unPost =
postsOtrosUsuarios.set(i,postsOtrosUsuarios.get(masNuevo));

      postsOtrosUsuarios.set(masNuevo, unPost);

   }

    List<Post> ultimosPosts = new ArrayList<Post>();

    int index = 0;

    Iterator<Post> postIterator = postsOtrosUsuarios.iterator();

    while (postIterator.hasNext() &&  index < cantidad) {
```

```
        ultimosPosts.add(postIterator.next());
    }
    return ultimosPosts;
}
```

2.3

Ejercicio original

```
/**
* Retorna los últimos N posts que no pertenecen al usuario user
*/
public List<Post> ultimosPosts(Usuario user, int cantidad) {
  List<Post> postsOtrosUsuarios = new ArrayList<Post>();
  for (Post post : this.posts) {
      if (!post.getUsuario().equals(user)) {
          postsOtrosUsuarios.add(post);
      }
  }
 // ordena los posts por fecha
 for (int i = 0; i < postsOtrosUsuarios.size(); i++) {
     int masNuevo = i;
     for(int j= i +1; j < postsOtrosUsuarios.size(); j++) {
         if (postsOtrosUsuarios.get(j).getFecha().isAfter(
   postsOtrosUsuarios.get(masNuevo).getFecha())) {
             masNuevo = j;
```

```
            }
        }
      Post unPost = postsOtrosUsuarios.set(i,postsOtrosUsuarios.get(masNuevo));
      postsOtrosUsuarios.set(masNuevo, unPost);
  }
    List<Post> ultimosPosts = new ArrayList<Post>();
    int index = 0;
    Iterator<Post> postIterator = postsOtrosUsuarios.iterator();
    while (postIterator.hasNext() &&  index < cantidad) {
        ultimosPosts.add(postIterator.next());
    }
    return ultimosPosts;
}
```

(1) Code smell: Metodo largo

El metodo es largo, y tiene mas de una responsabilidad. Se deben separar las responsabilidades

Refactoring: Separar responsabilidades

```
public List<Post> ultimosPosts(Usuario user, int cantidad) {
    List<Post> postsOtrosUsuarios = this.obtenerPostDeOtrosUsuarios();
    this.ordenarPostPorFecha(user);



    List<Post> ultimosPosts = new ArrayList<Post>();
    filtrarPrimerosPostSegunCantidad(cantidad,postsOtrosUsuarios,ultimosPosts);
```

```
  return ultimosPosts;
}

private List<Post> obtenerPostDeOtrosUsuarios(Usuario user)
{
  List<Post> listaPostOtrosUsuarios = new ArrayList<Post>();
  for (Post post : this.posts) {
    if (!post.getUsuario().equals(user)) {
      listaPostOtrosUsuarios.add(post);
    }
  }
  return listaPostOtrosUsuarios;
}

private void ordenarPostPorFecha(List<Post> posts)
{
  // ordena los posts por fecha
  for (int i = 0; i < posts.size(); i++) {
    int masNuevo = i;
    for(int j= i +1; j < posts.size(); j++) {
        if (posts.get(j).getFecha().isAfter(
          posts.get(masNuevo).getFecha())) {
```

```
            masNuevo = j;
          }
      }
    Post unPost = posts.set(i,posts.get(masNuevo));
    posts.set(masNuevo, unPost);
  }
}

private void filtrarPrimerosPostSegunCantidad(int cantiddad, List<Post>
postsOtrosUsuarios, List<Post> ultimosPosts>)
{
  int index = 0;
  Iterator<Post> postIterator = postsOtrosUsuarios.iterator();
  while (postIterator.hasNext() &&  index < cantidad) {
      ultimosPosts.add(postIterator.next());
  }
}
```

(2)Code smell: Reinventando la rueda en los metododos: 'obtenerPostDeOtrosUsuarios' 'ordenarPostPorFecha' y 'filtrarPrimerosPostSegunCantidad'

Refactoring: Reemplazar for por stream()

```
public List<Post> ultimosPosts(Usuario user, int cantidad) {
  List<Post> postsOtrosUsuarios = this.obtenerPostDeOtrosUsuarios();
```

```
  this.ordenarPostPorFecha(user);



  List<Post> ultimosPosts = new ArrayList<Post>();

  filtrarPrimerosPostSegunCantidad(cantidad,postsOtrosUsuarios,ultimosPosts);


  return ultimosPosts;

}


private List<Post> obtenerPostDeOtrosUsuarios(Usuario user)

{

  return this.posts.stream()

  .filter(post -> !post.getUsuario.equals(user))

  .collect(Collectors.toList());

}


private void ordenarPostPorFecha(List<Post> posts)

{

  posts.sort((post1, post2) -> post2.getFecha().compareTo(post1.getFecha()));

}


private void filtrarPrimerosPostSegunCantidad(int cantiddad, List<Post>
postsOtrosUsuarios, List<Post> ultimosPosts>)

{
```

```
  postsOtrosUsuarios.stream()

  .limit(cantidad)  // Limita el número de elementos a la cantidad
especificada

  .forEach(ultimosPosts::add);  // Agrega cada elemento al listado
ultimosPosts

}
```

2.4

Codigo original

```
public class Producto {

  private String nombre;

  private double precio;

  public double getPrecio() {

      return this.precio;

  }

}



public class ItemCarrito {

  private Producto producto;

  private int cantidad;

  public Producto getProducto() {

      return this.producto;

  }

  public int getCantidad() {

      return this.cantidad;
```

```
  }


}


public class Carrito {
  private List<ItemCarrito> items;
  public double total() {
return this.items.stream()
.mapToDouble(item ->
item.getProducto().getPrecio() * item.getCantidad())
.sum();
  }
}
```

(1) Code Smell: Envidia de atributos

En el metodo total dentro de la clase carrito, nos traemos atributos que no nos pertenecen para realizar un calculo que no es responsabilidad de la clase Carrito

Refactoring: Move method, crear metodo calcularPrecio dentro de la clase itemCarrito

```
public class Producto {
  private String nombre;
  private double precio;
  public double getPrecio() {
      return this.precio;
```

```
  }
}


public class ItemCarrito {
  private Producto producto;
  private int cantidad;
  public Producto getProducto() {
      return this.producto;
  }
  public int getCantidad() {
      return this.cantidad;
  }

  public double calcularPrecio()
  {
    return producto.getPrecio() * this.cantidad;
  }

}

public class Carrito {
  private List<ItemCarrito> items;
  public double total() {
```

```
    return this.items.stream()
    .mapToDouble(item -> item.calcularPrecio();
    .sum();
      }
}
```

(2) Code Smell: Nombre del metodo poco autoexplicativo

En la clase carrito, el nombre del metodo total, no explica cual es su funcionalidad

Refactoring: Renombrar metodo

```
public class Producto {
  private String nombre;
  private double precio;
  public double getPrecio() {
      return this.precio;
  }
}


public class ItemCarrito {
  private Producto producto;
  private int cantidad;
  public Producto getProducto() {
      return this.producto;
  }
  public int getCantidad() {
```

```
      return this.cantidad;
  }


  public double calcularPrecio()
  {
    return producto.getPrecio() * this.cantidad;
  }


}


public class Carrito {
  private List<ItemCarrito> items;
  public double calcularPrecioTotal() {
  return this.items.stream()
  .mapToDouble(item -> item.calcularPrecio()
  .sum();
    }
}
```

2.5

Codigo original

```
public class Supermercado {

  public void notificarPedido(long nroPedido, Cliente cliente) {
```

```
    String notificacion = MessageFormat.format("Estimado cliente, se le
informa que hemos recibido su pedido con número {0}, el cual será enviado a la
dirección {1}", new Object[] { nroPedido, cliente.getDireccionFormateada() });


    // lo imprimimos en pantalla, podría ser un mail, SMS, etc..

   System.out.println(notificacion);

 }

}



public class Cliente {

  public String getDireccionFormateada() {

 return

   this.direccion.getLocalidad() + ", " +

   this.direccion.getCalle() + ", " +

   this.direccion.getNumero() + ", " +

   this.direccion.getDepartamento()

     ;

}



public class Direccion {

  public String localidad;

  public String calle;

  public String numero;

  public String departamento;
```

```
}
```

(1) Code Smells: Envidia de atributos

La clase Cliente, accede a los atributos de la clase Direccion para realizar un formato de direccion con sus datos, cuando en realidad esto debe ser responsabilidad de la clase direccion

Refactoring: Mover metodo a clase Direccion.

```
public class Supermercado {



  public void notificarPedido(long nroPedido, Cliente cliente) {

    String notificacion = "Estimado cliente, se le informa que hemos recibido
su pedido con numero "

    + nroPedido + ", el cual sera enviado a la direccion "

    + cliente.getDireccionCompleta();




    // lo imprimimos en pantalla, podría ser un mail, SMS, etc..

   System.out.println(notificacion);

 }

}



public class Cliente {

  private Direccion direccion;



  public String getDireccionCompleta()
```

```
  {
    return this.direccion.getDireccionFormateada();
  }
}



public class Direccion {
   public String localidad;
   public String calle;
   public String numero;
   public String departamento;

   public String getDireccionFormateada()
   {
    return this.localidad + " "+this.calle+ " "+ this.numero+" "+
this.departamento;
   }
  }
```

(2) Code Smells: Rompe el encapsulamiento

La clase Direccion al tener sus atributos publicos rompe el encapsulamiento, permitiendo que modifiquen sus atributos desde fuera de su clase.

Refactoring: Cambiar acceso a los atributos

```
  public class Supermercado {
```

```
    public void notificarPedido(long nroPedido, Cliente cliente) {
      String notificacion = "Estimado cliente, se le informa que hemos
recibido su pedido con numero "
      + nroPedido + ", el cual sera enviado a la direccion "
      + cliente.getDireccionCompleta();
      // lo imprimimos en pantalla, podría ser un mail, SMS, etc..
     System.out.println(notificacion);
   }
  }


  public class Cliente {
    private Direccion direccion;
    public String getDireccionCompleta()
    {
      return this.direccion.getDireccionFormateada();
    }
  }


  public class Direccion {
     private String localidad;
     private String calle;
     private String numero;
     private String departamento;
```

```
     public String getDireccionFormateada()

     {

      return this.localidad + " "+this.calle+ " "+ this.numero+" "+
this.departamento;

     }

    }
```

2.6

Codigo original

```
public class Usuario {

  String tipoSubscripcion;

  // ...



  public void setTipoSubscripcion(String unTipo) {

    this.tipoSubscripcion = unTipo;

  }

  public double calcularCostoPelicula(Pelicula pelicula) {

    double costo = 0;

    if (tipoSubscripcion=="Basico") {

      costo = pelicula.getCosto() + pelicula.calcularCargoExtraPorEstreno();

    }

    else if (tipoSubscripcion== "Familia") {

      costo = (pelicula.getCosto() + pelicula.calcularCargoExtraPorEstreno())
* 0.90;
```

```
    }
    else if (tipoSubscripcion=="Plus") {
      costo = pelicula.getCosto();
    }
    else if (tipoSubscripcion=="Premium") {
      costo = pelicula.getCosto() * 0.75;
    }
    return costo;
  }
}


public class Pelicula {
  LocalDate fechaEstreno;
  // ...

  public double getCosto() {
    return this.costo;
  }
  public double calcularCargoExtraPorEstreno(){
// Si la Película se estrenó 30 días antes de la fecha actual, retorna un
cargo de 0$, caso contrario, retorna un cargo extra de 300$
   return (ChronoUnit.DAYS.between(this.fechaEstreno, LocalDate.now()) ) > 30
? 0 : 300;
  }
```

```
}
```

(1) Code Smell: Switch Statement / falta de polimorfismo

En la clase Usuario, el metodo calcularCostoPelicula esta lleno de if-else preguntando por un tipo de subscripcion, es un mal olor de que hay algo que esta mal, se debe aplicar polimorfismo

Refactoring: Reemplazar condicionales por polimorfismo

```
public class Usuario {

  private Subscripcion tipoSubscripcion;

  // ...


  public void setTipoSubscripcion(Subscripcion unTipo) {

    this.tipoSubscripcion = unTipo;

  }

  public double calcularCostoPelicula(Pelicula pelicula) {

    return tipoSubscripcion.calcularCostoPelicula(pelicula);

  }


  public interface Subscripcion {

    public double calcularCostoPelicula(Pelicula pelicula);

}


public class SubscripcionBasica implements Subscripcion{

  @Override

  public double calcularCostoPelicula(Pelicula pelicula) {
```

```
        return pelicula.getCosto() + pelicula.calcularCargoExtraPorEstreno();
    }
}


public class SubscripcionFamilia implements Subscripcion{
    @Override
    public double calcularCostoPelicula(Pelicula pelicula) {
        return (pelicula.getCosto() + pelicula.calcularCargoExtraPorEstreno()) *
0.90;
    }
}


public class SubscripcionPlus implements Subscripcion{
    @Override
    public double calcularCostoPelicula(Pelicula pelicula) {
        return pelicula.getCosto();
    }
}


public class SubscripcionPremium implements Subscripcion{
    @Override
    public double calcularCostoPelicula(Pelicula pelicula) {
        return pelicula.getCosto() * 0.75;
```

```
  }

}



public class Pelicula {

  private LocalDate fechaEstreno;

  // ...



  public double getCosto() {

    return this.costo;

  }

  public double calcularCargoExtraPorEstreno(){

// Si la Película se estrenó 30 días antes de la fecha actual, retorna un
cargo de 0$, caso contrario, retorna un cargo extra de 300$

   return (ChronoUnit.DAYS.between(this.fechaEstreno, LocalDate.now()) ) > 30
? 0 : 300;

  }

}

}
```

(2) Code Smell: Envidia de atributos

Los metodos calcularCostoPelicula en las subclases subscripcionBasica y subscripcionFamilo ejecutan logica que le pertenece a la clase Pelicula.

```
public class Usuario {

  // ...

  private Subscripcion tipoSubscripcion;
```

```
  public void setTipoSubscripcion(Subscripcion unTipo) {
      this.tipoSubscripcion = unTipo;
  }

  public double calcularCostoPelicula(Pelicula pelicula) {
      return tipoSubscripcion.calcularCostoPelicula(pelicula);
  }
}

public interface Subscripcion {
  public double calcularCostoPelicula(Pelicula pelicula);
}

public class SubscripcionBasica implements Subscripcion{
  @Override
  public double calcularCostoPelicula(Pelicula pelicula) {
      return pelicula.calcularCostoConCargoExtra();
  }
}

public class SubscripcionFamilia implements Subscripcion{
  @Override
```

```
  public double calcularCostoPelicula(Pelicula pelicula) {
      return pelicula.calcularCostoConCargoExtra() * 0.90;
  }
}

public class SubscripcionPlus implements Subscripcion{
  @Override
  public double calcularCostoPelicula(Pelicula pelicula) {
      return pelicula.getCosto();
  }
}

public class SubscripcionPremium implements Subscripcion{
  @Override
  public double calcularCostoPelicula(Pelicula pelicula) {
      return pelicula.getCosto() * 0.75;
  }
}

public class Pelicula {
  private LocalDate fechaEstreno;
  // ...
```

```
  public double getCosto() {

    return this.costo;

  }

  public double calcularCostoConCargoExtra() {

      return this.getCosto() + this.calcularCargoExtraPorEstreno();

  }



  public double calcularCargoExtraPorEstreno(){

// Si la Película se estrenó 30 días antes de la fecha actual, retorna un
cargo de 0$, caso contrario, retorna un cargo extra de 300$

   return (ChronoUnit.DAYS.between(this.fechaEstreno, LocalDate.now()) ) > 30
? 0 : 300;

  }

}
```