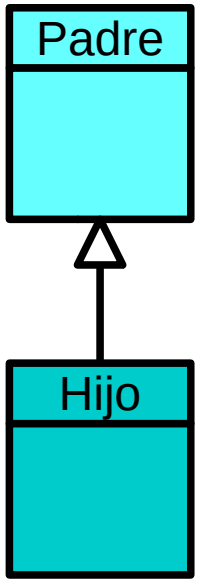
CLASE 5 - INTERFACES

Herencia simple vs herencia múltiple.
Problemas de la herencia múltiple.
Interfaces. Definición
Interfaces. Clases abstractas.
Interfaces. Qué es?
Diseño de comportamiento.
Declaración.
Herencia, composición e interfaces.
Interfaces propias del lenguaje Java.
Principio de segregación de interfaces.
Comparación entre clase abstracta e interface.
Concepto de doble envío. (Double Dispatch).
La clase Object: el método clone().
Introducción a excepciones para poder utilizar el método clone().
Clone. Funcionamiento. Actitudes frente a clone(). Estrategias de clonación.

Herencia simple vs herencia múltiple

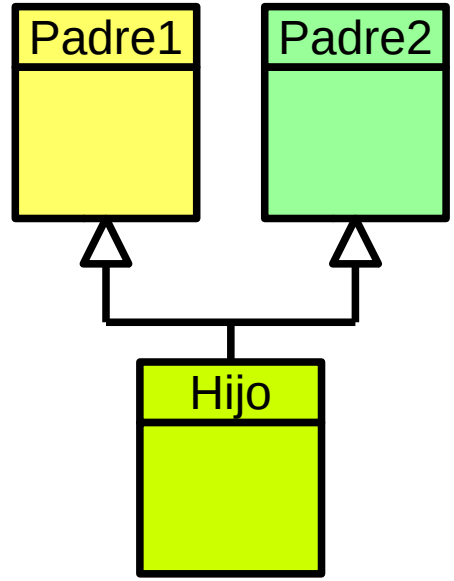
Herencia Simple

Se pueden definir nuevas clases solamente a partir de una clase inicial



Herencia Múltiple

Se pueden definir nuevas clases a partir de una o varias clases iniciales



Herencia simple vs herencia múltiple

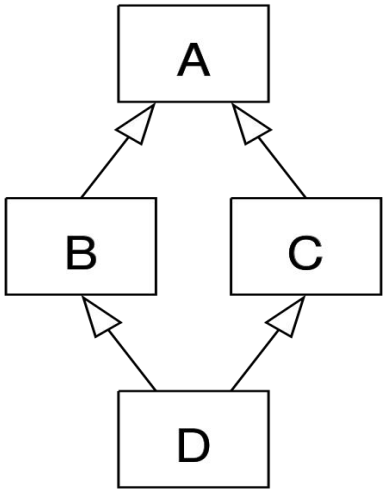
La extensión de una clase significa que la nueva clase ***no sólo*** hereda el ***contrato*** de su superclase, sino también la ***implementación*** de su superclase...

La herencia múltiple es útil cuando una nueva clase combina múltiples contratos y hereda alguna de las implementaciones de esos contratos, o todas.

Si hay más de una superclase, surgen problemas cuando el comportamiento de una superclase se hereda de más de una forma.

No todos los lenguajes implementan la herencia múltiple

Problemas de la herencia múltiple



Herencia diamante

Los problemas aparecen en la herencia de la ***implementación***

Problema del diamante

En los lenguajes de programación orientada a objetos, el problema del diamante es una ambigüedad que surge cuando dos clases B y C heredan de A, y la clase D hereda de B y C. Si un método en D llama a un método definido en A, ¿por qué clase lo hereda, B o C?

Problemas de la herencia múltiple – solución por interfaz

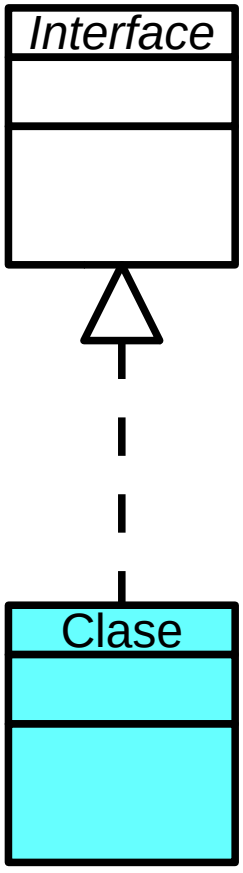
Java **no** proporciona mecanismos de herencia múltiple. Evita los problemas de implementación, pero propone otra solución mediante el uso de la INTERFACE

<< *Interface* >>

Lo bueno !

Proporcionar los medios para heredar un contrato abstracto sin heredar la implementación, permite tener los beneficios de la herencia múltiple sin que aparezcan los problemas de la herencia de implementación.
La herencia de un contrato abstracto se denomina herencia de interfaz.

Interfaces



La unidad fundamental en Java es la ***clase***, pero la unidad fundamental en el diseño orientado a objetos es el ***tipo***.

Aunque las clases definen tipos, es muy útil y potente poder ***definir un tipo sin definir una clase***.

Las interfaces definen tipos de un modo abstracto en forma de una colección de métodos o de otros tipos que conforman el contrato del tipo definido.

Las clases implementan interfaces.

Interfaces – clases abstractas

La clase que implementa una clase abstracta debe ser una subclase de la clase abstracta. Existe relación de herencia entre la clase abstracta y la clase que la implementa.

Cualquier clase puede implementar una interface, debe implementar TODOS los métodos requeridos. NO depende de donde esté ubicada la clase en la jerarquía de clases, ***NO hay relación de herencia entre la interface*** y la clase que la implementa.

Interfaces

Una interfaz (interface) es un conjunto de declaraciones de métodos (sin definición).

También puede definir constantes, que son implícitamente public, static y final, y deben siempre inicializarse en la declaración.

```
public interface NombreInterface
{
   public static final TIPO constante = valor;
   …
   public void metodo(param, param2);
   public void metodo2(param);
   …
}
```

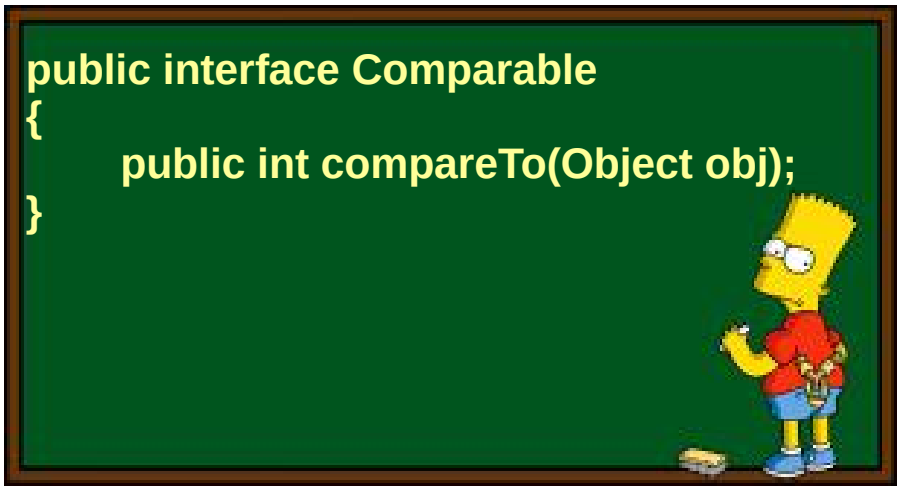
Interfaces

Una interface es parecido a una clase abstracta, pero sólo puede tener definidos **métodos abstractos y constantes** (static/final).

Ni atributos ni implementaciones de métodos.

Se crea igual que las clases, pero utilizando ***interface*** en lugar de ***class***:

EJEMPLO de interface provista por Java



Esta interfaz define un único método, que indica si el objeto receptor es menor que el proporcionado (resultado negativo), es mayor (resultado positivo) o es igual (0 como resultado).

Interfaces - Definiciones

De igual manera que las clases, las interfaces son un tipo de dato, es decir podemos declarar variables del tipo de la interface.

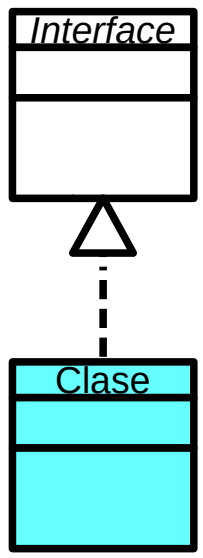
```
Mi_interfaz mi_var;
```

No es posible crear instancias de una interface.

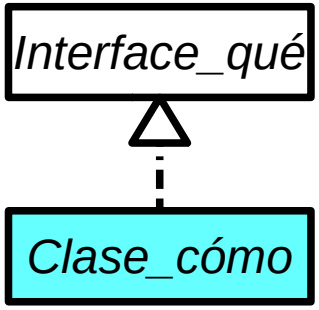
```
mi_var = new Mi_interfaz();
```

Una clase debe implementar la interface y así proveer el comportamiento necesario de los métodos declarados en la interface.

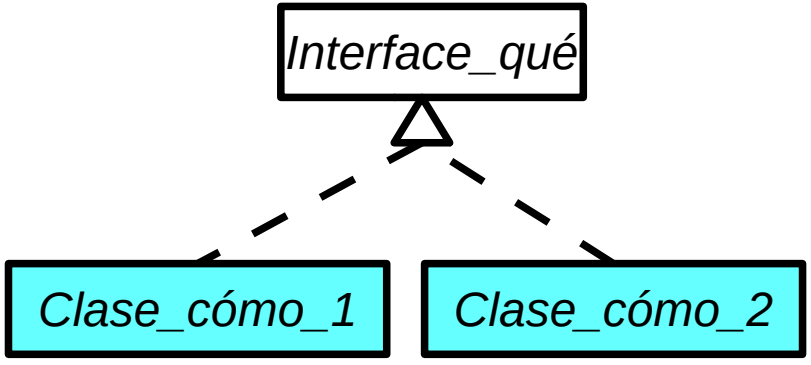
Una clase que implementa una interface adquiere las constantes de la interface.



Interfaces - Definiciones



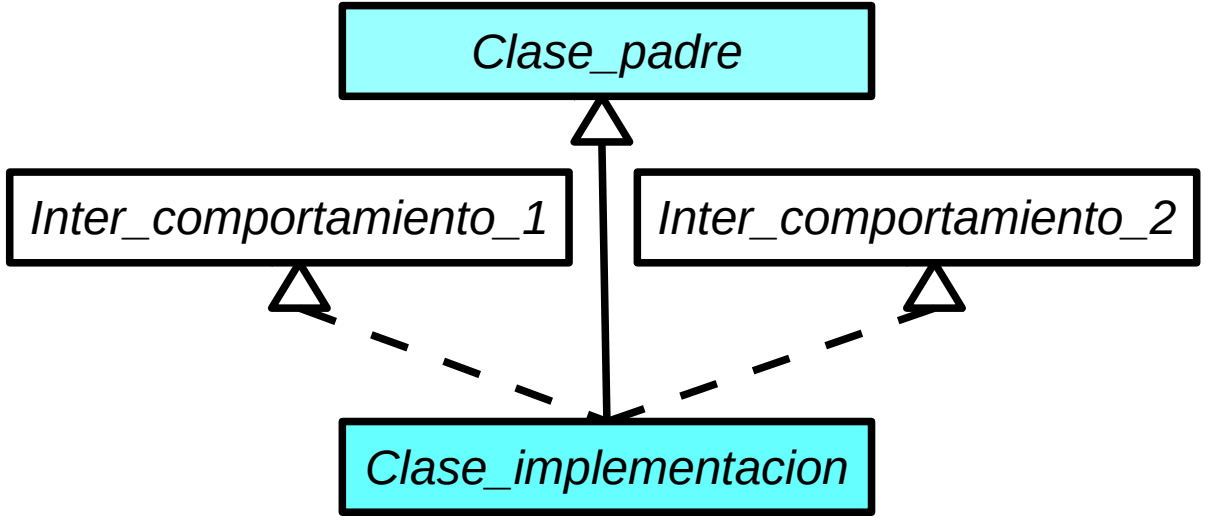
Una interface establece **qué** debe hacer la clase que la implementa, **sin especificar el *cómo***. Una instancia de dicha clase, es del tipo de la clase y de la interface. *Similar a Herencia Múltiple.*



Las interfaces permiten que objetos que no comparten la misma cadena de herencia, sean del mismo tipo en virtud de implementar la misma interface.

Interfaces - Definiciones

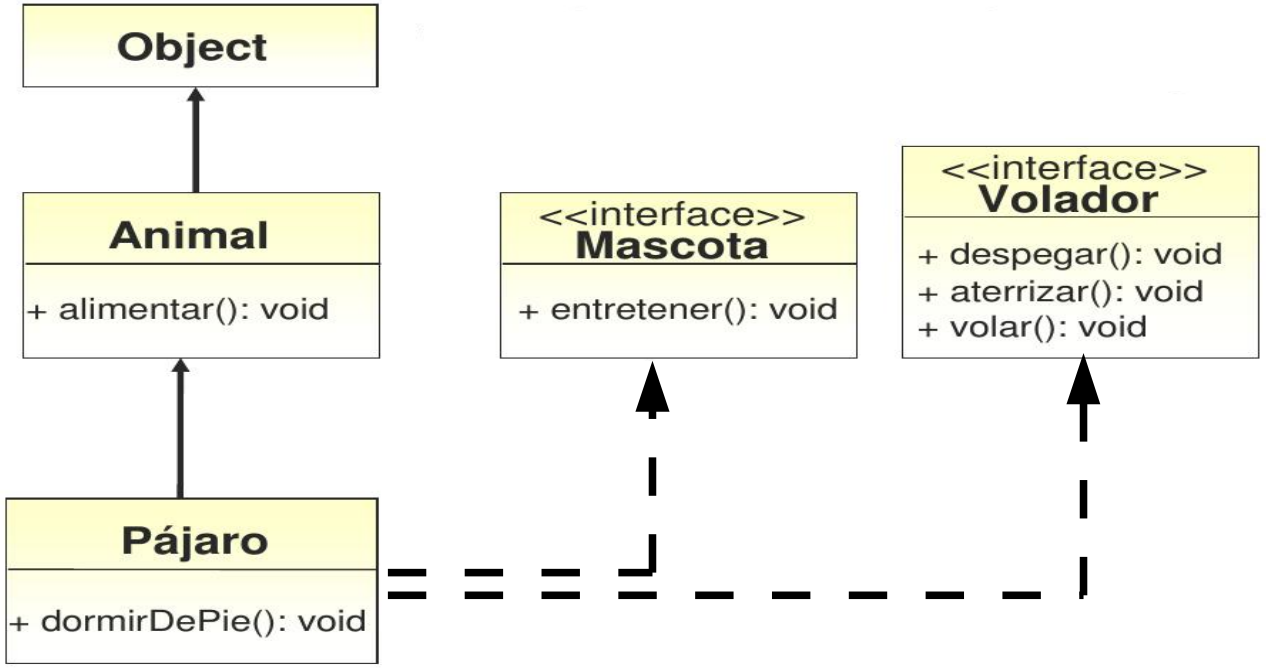
- Una interface puede extender múltiples interfaces. Por lo tanto se tiene **herencia múltiple de interfaces**.
- Las interfaces proveen una alternativa a la herencia múltiple. Las clases en JAVA pueden heredar de una única clase, pero pueden implementar múltiples interfaces.



Interfaces – diseño de comportamiento

La interfaz permite diseñar el comportamiento que luego implementará la clase correspondiente.

Como consecuencia de la implementación (por parte de una clase) de varias interfaces, se puede dar a dicha clase varios comportamientos establecidos.



Interfaces – Declaración – Ejemplo

- Para declarar una interface se utiliza la palabra clave **interface**
- Las constantes son implícitamente **public**, **static** y **final**. No es necesario escribirlo en el código.
- Los métodos son implícitamente **public** y **abstract**.
- El archivo fuente de las interfaces, al igual que el de la clases, debe tener exactamente el mismo nombre que la interface y extensión **.java**

```
package interfaces;
public interface Volador {
  long UN_SEGUNDO = 60;
  long UN_MINUTO = 3600;
  void aterrizar();
  void despegar();
  void volar();
}
```

La interface **Volador** es **public**, por lo tanto puede ser usada por cualquier clase e interface.

```
package interfaces;
public interface Prueba extends Volador, Mascota {
    void probar();
}
```

- La interface **Prueba** extiende las interfaces **Volador** y **Mascota**: herencia múltiple de interfaces.
- La interface **Prueba** hereda todas las constantes y métodos de sus superinterfaces

```
package interfaces;

interface Mascota {

    void entretener();
}
```

La interface **Mascota** es de acceso **package**, por lo tanto sólo puede ser usada por las clases e interfaces que pertenecen a su paquete.

Interfaces – Implementación – Ejemplo

```
package modelo;
import interfaces.Volador;
public class Pajaro implements Volador {
```

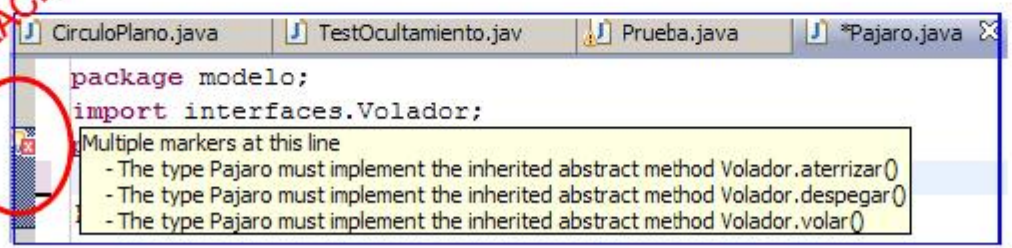
Para indicar que una clase implementa una interface se usa la palabra clave **implements**

¿A qué está obligada la clase **Pajaro**?

A implementar **TODOS** los métodos declarados en la interface **Volador**

¿Qué pasa si **Pajaro** NO implementa **TODOS** los métodos de la interface **Volador**?

ERROR DE COMPILACIÓN!!!



Se produce un error de compilación: **una clase que implementa una interface, debe implementar TODOS los métodos declarados en la interface; las constantes las hereda**.

Interfaces – Ejemplo

```
package modelo;
import interfaces.Volador;
public class Pajaro implements Volador {
public void aterrizar() {
  System.out.println("Pajaro.aterrizar()");
}

public void despegar() {
  System.out.println("Pajaro.despegar() durante "+UN_SEGUNDO);
}

public void volar() {
  System.out.println("Pajaro.volar() durante "+UN_MINUTO);
}
public void dormirDePie() {
   System.out.println("Pajaro.dormirDePie() ");
}
   ........
}
```

Más métodos

La palabra clave **implements** indica que la clase **Pajaro** implementa la interface **Volador**

La clase **Pajaro** debe implementar los 3 métodos definidos en la interface **Volador**

Las constantes **UN_MINUTO** y **UN_SEGUNDO** definidas en la interface **Volador** las hereda la clase **Pajaro** que la implementa

```
package interfaces;
public interface Volador {
  long UN_SEGUNDO = 60;
  long UN_MINUTO = 3600;
  void aterrizar();
  void despegar();
  void volar();
}
```

Interfaces – Ejemplo

Volador.java | Pajaro.java | Mascota.java | TestPaja

```
package modelo;
public class TestPajaro {
    public static void main(String[] args) {
        Pajaro p=new Pajaro();
        p.dormirDePie();
        p.despegar();
        p.aterrizar();
        p.volar();
        p.entretener();
    }
}
```

La variable **p** es de tipo **Pajaro, Volador y Mascota:**

- **Por ser Pajaro, sabe: dormirDePie()**
- **Por ser Volador, sabe: despegar(), aterrizar() y volar()**
- **Por ser Mascota, sabe: entretener()**

¿Cuál es la salida?

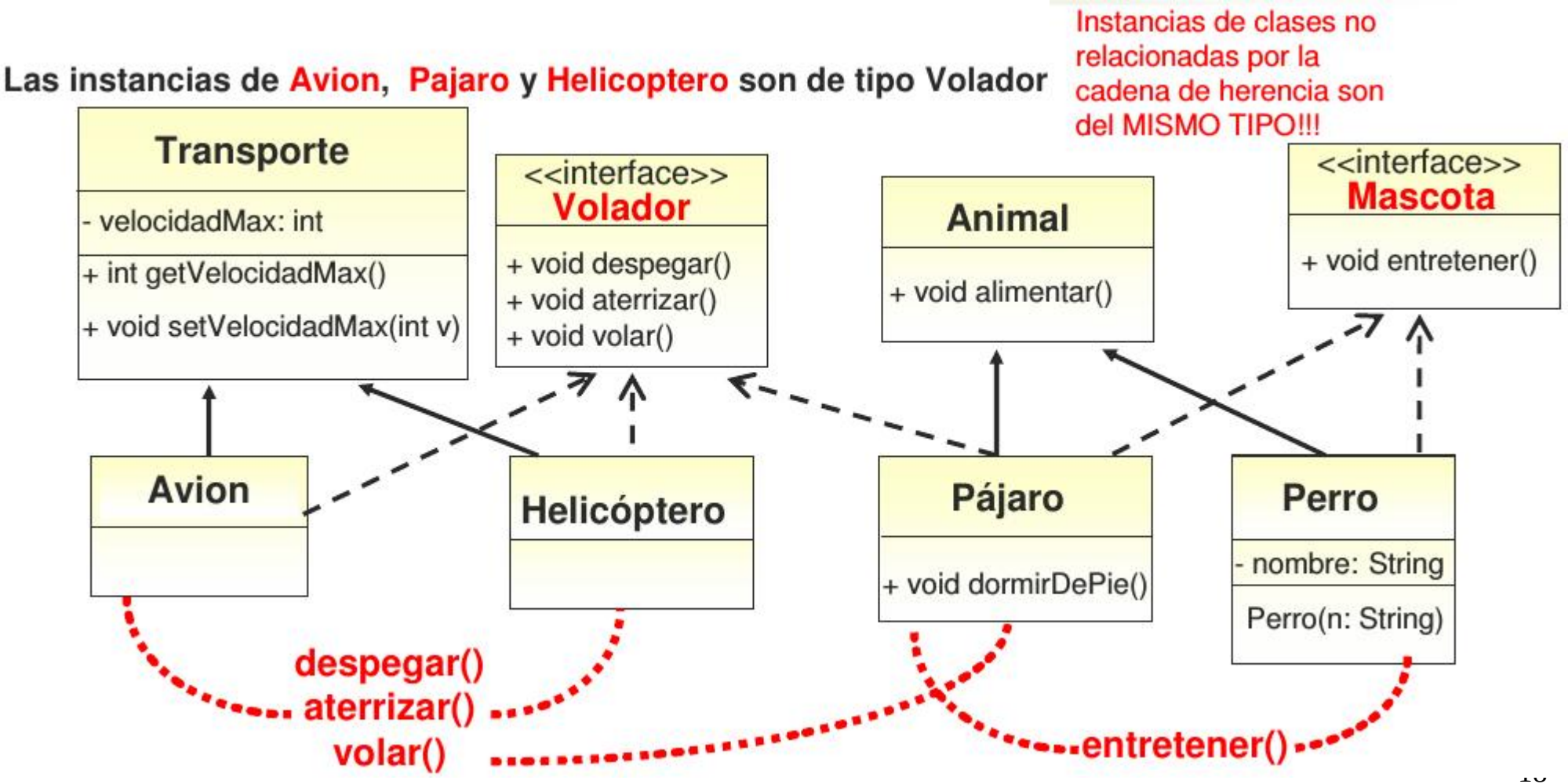
Problems | Javadoc | Declaration | Console

<terminated> TestPajaro [Java Application] C:\Archivos de p

```
Pajaro.dormirDePie()
Pajaro.despegar() durante 60
Pajaro.aterrizar()
Pajaro.volar() durante 3600
Pajaro.entrener()
```

Es un mecanismo similar a la herencia múltiple, aunque NO lo es porque de las interfaces a diferencia de las clases, NO se hereda nada.

Interfaces – Ejemplo



Interfaces – Ejemplo

Una de las formas en que JAVA implementa el **polimorfismo** es mediante **sobreescritura de métodos a través de interfaces.**

```
package test;

import modelo.Pajaro;
import modelo.Avion;
import modelo.Helicoptero;
import interfaces.Volador;

public class PruebaInterfaces {

    public static void partida(Volador v){
        v.despegar();
    }

    public static void main(String[] args) {
        Volador[] m = new Volador[3];
        m[0]= new Avion();
        m[1]= new Helicoptero();
        m[2]= new Pajaro();
        for (int j=0; j<m.length; j++)
        partida(m[j]);
    }
}
```

UPCASTING

Upcasting al tipo Volador: convierte objetos de una clase al de la interface implementada por la clases

El método despegar() es polimórfico:
El método **despegar()** que se ejecutará está determinado por el tipo real del objeto **v** (**Avion**, **Helicoptero** y **Pajaro**) y NO por el tipo de la variable, **Volador.**
En nuestro caso, más de una clase implementó la interface **Volador** y en consecuencia tenemos múltiples versiones del método **despegar()**. El **binding dinámico** resuelve a que método **despegar()** se invocará.

Declaramos un arreglo de tipo **Volador: cada objeto del arreglo debe ser de tipo Volador**

EJEMPLO: Herencia, Composición e Interfaces

Se quiere diseñar un sistema que modele el sistema solar

Modelar las características de los cuerpos celestes y sus vinculaciones.

Por un lado debemos tener en cuenta las características comunes de todos los cuerpos celestes, en nuestro caso, solamente nombre y cuerpo celeste sobre el cual orbita (ej: la tierra gira alrededor del sol); y por otro lado tener en cuenta que cada cuerpo celeste posee un conjunto particular de características que no se presentan en otros cuerpos.

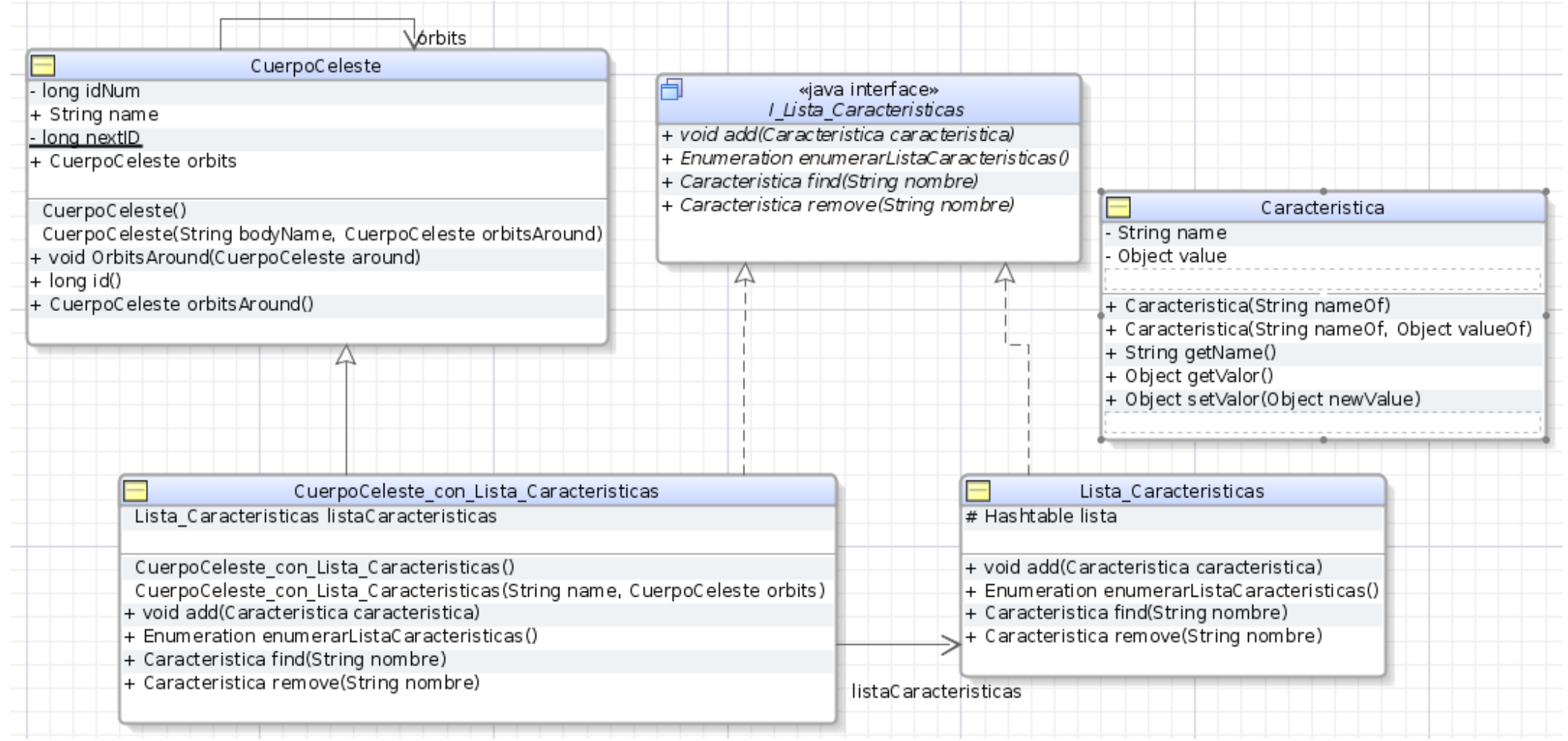
El conjunto particular de características no comunes se modelará en forma lista dinámica propia de cada cuerpo celeste.

Los elementos que componen cada lista, se definen uno a uno de forma especifica para cada cuerpo celeste. Por ejemplo, saturno tiene anillos, y la característica que lo representa será modelada mediante el binomio (<nombre>, <valor>)

Entonces tenemos los binomios:

- (“cantidad_anillos”, <un número>)
- (“distancia_promedio”, <un número>)

EJEMPLO: Herencia, Composición e Interfaces



EJEMPLO: Herencia, Composición e Interfaces

```
public class CuerpoCeleste
{
    private long idNum;
    public String name = "<unnamed>";
    public CuerpoCeleste orbits = null;

    private static long nextID = 0;

    CuerpoCeleste()
    {
        idNum = nextID++;
    }

    CuerpoCeleste(String bodyName, CuerpoCeleste orbitsAround)
    {...}

    public long id()
    {...}

    public CuerpoCeleste orbitsAround()
    {...}

    public void OrbitsAround(CuerpoCeleste around)
    {...}

}
```

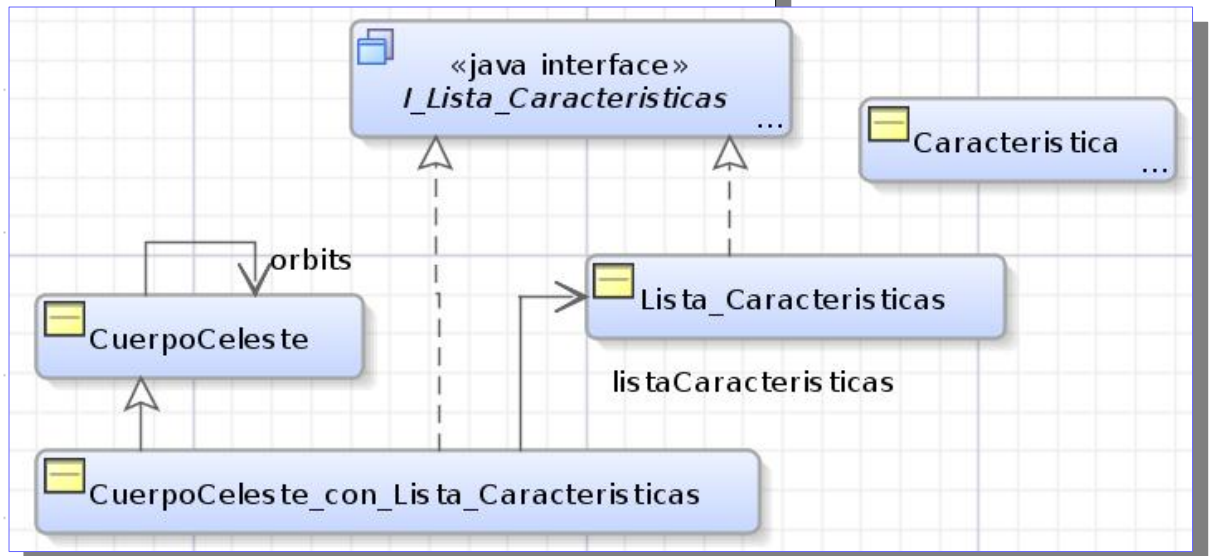
EJEMPLO: Herencia, Composición e Interfaces

```java
public interface I_Lista_Caracteristicas
{
    void add(Caracteristica caracteristica);
    Caracteristica find(String nombre);
    Caracteristica remove(String nombre);
    Enumeration enumerarListaCaracteristicas();
}
```

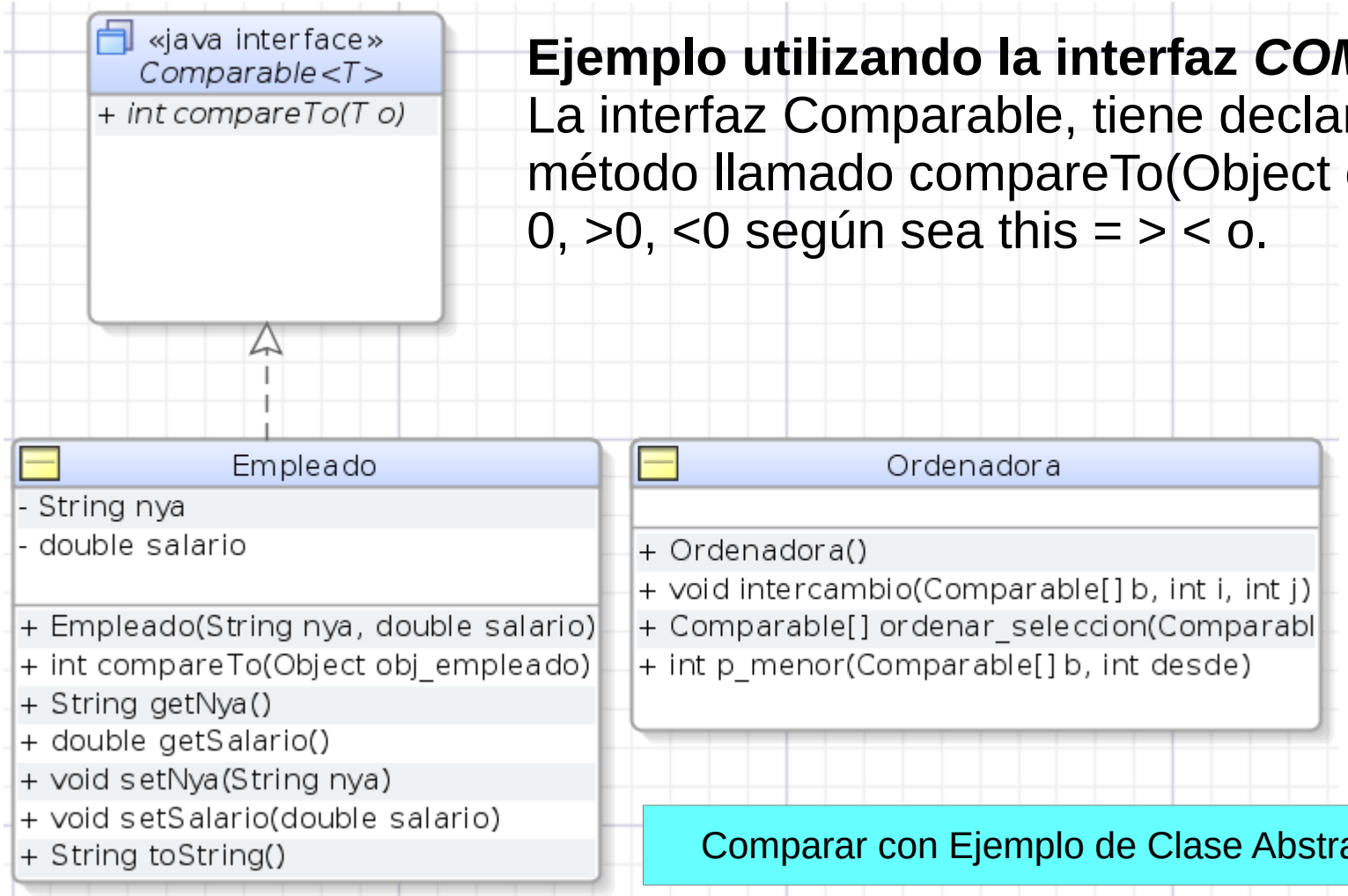
```java
public class Lista_Caracteristicas implements I_Lista_Caracteristicas
{
    protected Hashtable lista = new Hashtable();

    public void add(Caracteristica caracteristica)
    {
        lista.put(caracteristica.nameOf(), caracteristica);
    }

    public Caracteristica find(String nombre)
    {

        return (Caracteristica)lista.get(nombre);
    }

    public Caracteristica remove(String nombre)
    {
        return (Caracteristica)lista.remove(nombre);
    }

    public Enumeration enumerarListaCaracteristicas()
    {
        return lista.elements() ;
    }
}
```

EJEMPLO: Herencia, Composición e Interfaces

```
public class CuerpoCeleste_con_Lista_Caracteristicas extends CuerpoCeleste implements I_Lista_Caracteristicas
{
    Lista_Caracteristicas listaCaracteristicas = new Lista_Caracteristicas();

    CuerpoCeleste_con_Lista_Caracteristicas()
    {
        super();
    }

    CuerpoCeleste_con_Lista_Caracteristicas(String name, CuerpoCeleste orbits)
    {
        super(name, orbits);
    }

    public void add(Caracteristica caracteristica)
    {
        listaCaracteristicas.add(caracteristica);
    }

    public Caracteristica find(String nombre)
    {
        return listaCaracteristicas.find(nombre);
    }

    public Caracteristica remove(String nombre)
    {
        return listaCaracteristicas.remove(nombre);
    }

    public Enumeration enumerarListaCaracteristicas()
    {
        return listaCaracteristicas.enumerarListaCaracteristicas();
    }
}
```



Interfaces propias del lenguaje de programación



Ejemplo utilizando la interfaz *COMPARABLE*

La interfaz Comparable, tiene declarado un único método llamado compareTo(Object o), que retorna 0, >0, <0 según sea this = > < o.

Comparar con Ejemplo de Clase Abstracta, de la Clase 3

Interfaces propias del lenguaje de programación

```java
public class Empleado implements Comparable
{
    private String nya;
    private double salario;

    public Empleado(String nya, double salario)
    {...}

    @Override
    public int compareTo(Object obj_empleado)
    {
        int rta = 0;
        Empleado eb = (Empleado) obj_empleado;
        if (getSalario() < eb.getSalario())
            rta = -1;
        else if (getSalario() > eb.getSalario())
            rta = 1;
        return rta;
    }
```

```java
public class Ordenadora
{
    public Ordenadora()
    {
        super();
    }

    public int p_menor(Comparable[] b, int desde)
    {
        int N = b.length;
        Comparable menor = b[desde];
        int posmenor = desde;
        for (int i = desde; i < N; i++)
        {
            if (menor.compareTo(b[i]) > 0)
            {
                posmenor = i;
                menor = b[i];
            }
        }
        return posmenor;
    }

    public void intercambio(Comparable[] b, int i, int j)
    {...}

    public Comparable[] ordenar_seleccion(Comparable[] a)
    {...}
}
```

Algunas conclusiones

- Con el mismo criterio, se podría implementar la interfaz Comparable en cualquier otra clase y utilizar una instancia de la clase Ordenadora para ordenarlos.
- La interface imprime a la clase que la implementa un nuevo comportamiento. Implementar varias interfaces permite que la clase adquiera varios comportamientos.
- Si declaramos una variable de tipo interfaz, estaremos considerando del objeto referido (instancia de la clase que la implementa) solamente los aspectos correspondientes a la interface, por ejemplo, sean ***Desplazable, Hablador, Calculador*** interfaces que modelan distintos comportamientos que serán implementados por la clase **Robot**. Si consideramos lo siguiente:

```
Hablador robot = new Robot();
…
robot.metodoX();
```

metodoX() corresponde solamente a un método declarado en la interface ***Hablador***

El principio de segregación de interfaces

“Muchas interfaces específicas son mejores que una única más general”

“Los clientes no deberían verse forzados a depender de interfaces que no usan”

El principio de segregación de interfaces

Este principio nos habla de las interfaces **pesadas o contaminadas**, esas que tienen una gran cantidad de métodos. Si un cliente implementa esa interface pues tendrá que **declarar métodos que no necesita** y los cuales estarán evidentemente vacíos.

```
1  public interface Worker{
2      void trabajar();
3      void descansar();
4  }
5
6  public class Trabajador implements Worker {
7      public void trabajar() {
8          // Codigo de trabajar
9      }
10
11     public void descansar() {
12         // Codigo de descansar
13     }
14 }
```

El principio de segregación de interfaces

Aquí la empresa tiene trabajadores que harán operaciones como trabajar y descansar.

Bien que pasaría si ahora la empresa tiene un robot que implementa esta interface Worker, entonces el robot esta obligado a tener un método descansar() que evidentemente no usara.

```
1  public interface Worker{
2      void trabajar();
3      void descansar();
4  }
5
6  public class Trabajador implements Worker {
7      public void trabajar() {
8          // Codigo de trabajar
9      }
10
11     public void descansar() {
12         // Codigo de descansar
13     }
14 }
```

El principio de segregación de interfaces

Cuando esto sucede es porque tenemos interfaces que no son cohesivas y lo mejor es dividirlas de manera funcional, es decir vamos a agrupar los métodos según su funcionalidad, así el cliente solo implementara la interface que necesite realmente.

```java
public interface Workable{
   void trabajar();
}

public interface Pausable{
   void descansar();
}

public class Trabajador implements Workable, Pausable{
   public void trabajar() {
      // Codigo de trabajar
   }

   public void descansar() {
      // Codigo de descansar
   }
}

public class Robot implements Workable{
   public void trabajar() {
      // Codigo de trabajar
   }
}
```

Comparación entre clase abstracta e interface

Clase Abstracta

1. Contiene métodos ejecutables y métodos abstractos
2. Una clase sólo puede extender de una única clase abstracta
3. Puede tener variables de instancia, constructores y cualquiera de los tipos de visibilidad: public, private, etc.

Interface

1. No tiene código de implementación. Todos sus métodos son abstractos.
2. Una clase puede implementar *n* números de interfaces
3. No puede tener variables de instacia o constructores y sólo puede tener métodos públicos o package.

Concepto de doble envío. (Double Dispatch)

```java
/**MAL IMPLEMENTADO
 */
public class Roca
{
    public Roca()
    {
        super();
    }

    public boolean beats(GameObject object)
    {
        boolean result = false;

        if (object.getClass().getName().equals("Roca"))
            result = false;
        else if (object.getClass().getName().equals("Paper"))
            result = false;
        else if (object.getClass().getName().equals("Scissors"))
            result = true;

        return result;
    }
}
```

Como modelar una interacción cuando el resultado depende del receptor y del argumento?

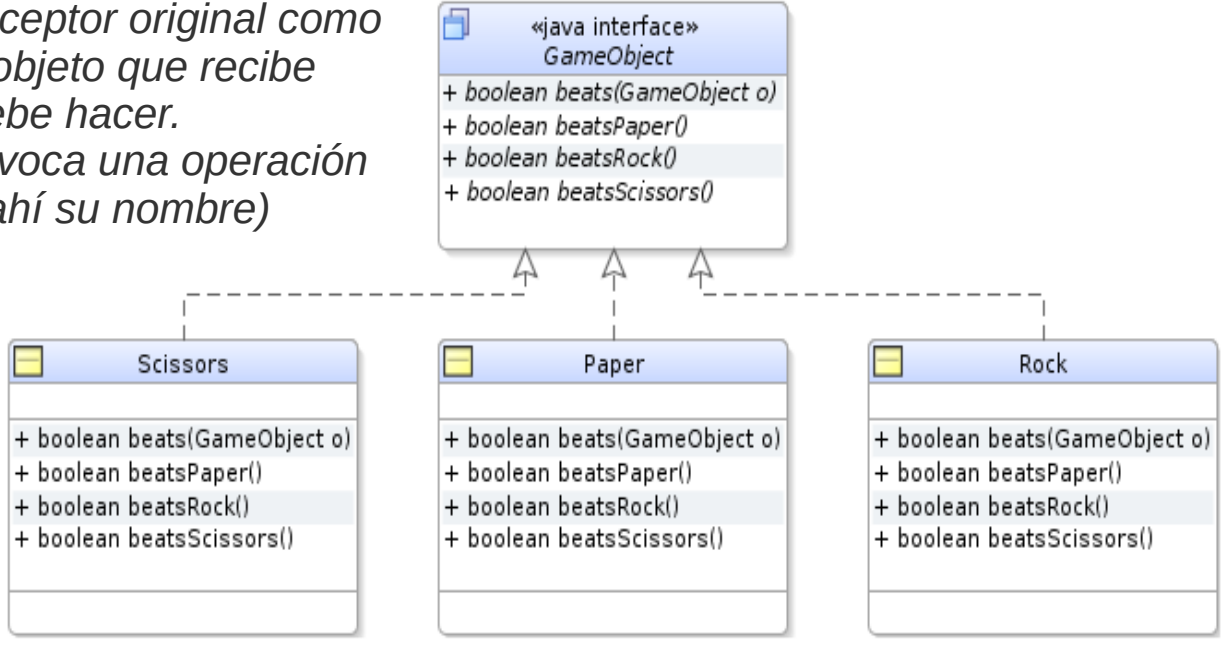
EJEMPLO:
Juego “Piedra, Papel o Tijera”
(mal diseñado)

Concepto de doble envío. (Double Dispatch)

Como modelar una interacción cuando el resultado depende del receptor y del argumento?

Solución: *en vez de escribir código que específicamente verifica la clase del parámetro, agregar nuevos métodos que tengan el mismo nombre (un método secundario) para cada clase de todos los potenciales objetos parámetros.*

Escribir el método original para simplemente llamar este nuevo método secundario, pasando el receptor original como argumento. Es responsabilidad de cada objeto que recibe este mensaje secundario conocer qué debe hacer. Típicamente, cada método secundario invoca una operación específica sobre el receptor original (de ahí su nombre)



Concepto de doble envío. (Double Dispatch)

```java
public class Rock implements GameObject
{
    /**El receptor del mensaje es una roca.
     * Se pregunta al parámetro (o) sobre rocas.
     * @param o
     * @return
     */
    public boolean beats(GameObject o)
    {
        return o.beatsRock();
    }

    /**Las rocas no ganan contra las rocas
     * @return
     */
    public boolean beatsRock()
    {
        return false;
    }

    /**Las rocas pierden con los papeles
     * @return
     */
    public boolean beatsPaper()
    {
        return false;
    }

    /**Las rocas ganan contra las tijeras
     * @return
     */
    public boolean beatsScissors()
    {
        return true;
    }
}
```

```java
public class Paper implements GameObject
{
    /**El receptor es un papel.
     * Se le pregunta al parámetro por un papel
     * @param o
     * @return
     */
    public boolean beats(GameObject o)
    {
        return o.beatsPaper();
    }

    /**Un papel le gana a
     * @return
     */
    public boolean beatsRo
    {
        return true;
    }

    public boolean beatsPa
    {
        return false;
    }

    public boolean beatsSc
    {
        return false;
    }
}
```

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

        GameObject papel = new Paper();
        GameObject tijera = new Scissors();
        GameObject piedra = new Rock();
        //suponga que el objeto piedra “golpea” a papel
        //Se ve el juego desde el punto de vista de piedra:
        boolean rta = piedra.beats(papel);
        if (rta)
            System.out.println("gana la piedra sobre el papel");
        else
            System.out.println("gana el papel sobre la piedra");
    }
}
```

Concepto de doble envío. (Double Dispatch)

piedra.beats(papel)

class Rock

```
public boolean beats(GameObject o)
{
    return o.beatsRock();
}
```

class Paper

```
public boolean beatsRock()
{
    return true;
}
```

piedra.beats() solamente puede pedir al parámetro (quien sea) que golpee contra la piedra. O sea, que ejecute **o.beatsRock()**.

En este caso, el objeto papel será quien ejecute el método **beatsRock()**.

De esta forma indirecta, se termina ejecutando el método:

papel.beatsRock()

Concepto de doble envío. (Double Dispatch)

```
/**MAL IMPLEMENTADO
 */
public class Roca
{
    public Roca()
    {
        super();
    }

    public boolean b
    {
        boolean result = false;

        if (object.getClass().getName().equals("Roca"))
            result = false;
        else if (object.getClass().getName().equals("Paper"))
            result = false;
        else if (object.getClass().getName().equals("Scissors"))
            result = true;

        return result;
```

Consecuencias: elimina declaraciones if o switch en función de la clase de un parámetro. Esto hace que el código sea más fácil de mantener. No resuelve completamente el problema de mantenimiento, pero apoya la extensión mediante la adición de métodos y no mediante la modificación de ellos.

Inconvenientes: La adición de una nueva clase de parámetro significa añadir un método secundario a ella, a menos se pueda añadir un método único a una superclase. También puede significar la adición de un método específico de clase para el objeto original (o la decisión de invocar una operación existente).

```
public class Rock implements GameObject
{
    /**El receptor del mensaje es una roca.
     * Se pregunta al parámetro (o) sobre rocas.
     * @param o
     * @return
     */
    public boolean beats(GameObject o)
    {
        return o.beatsRock();
    }

    /**Las rocas no ganan contra las rocas
     * @return
     */
    public boolean beatsRock()
    {
        return false;
    }

    /**Las rocas pierden con los papeles
     * @return
     */
    public boolean beatsPaper()
    {
        return false;
    }

    /**Las rocas ganan contra las tijeras
     * @return
     */
    public boolean beatsScissors()
    {
        return true;
    }
}
```

La clase Object: el método clone()

La clonación es el proceso de duplicación de un objeto para que en memoria existan dos objetos idénticos (cada uno con su identidad) en el mismo instante de tiempo.

Un método de clonación devuelve un nuevo objeto cuyo estado inicial es una copia del estado actual del objeto sobre el que se invoca **clone.**

Los cambios subsiguientes en ambos objetos no afectan al otro.

La clase Object: el método clone()

La clase java.lang.Object contiene una implementación ***native y protected*** del método ***clone()***. Esta implementación del método clone() determina cuánta memoria está siendo usada por el objeto a ser clonado, reserva la misma cantidad de memoria para el objeto clon, y **copia el estado del objeto original sobre el nuevo**. Y al final se devuelve un java.lang.Object el cual es la referencia al nuevo objeto (el clon).

Cómo se usa ?

```
…
Persona unaPersona = new Persona(....);
…
Persona otraPersona = (Persona)unaPersona.clone();
….
```

Problemas de utilizar la implementación por defecto del método clone()

La clase java.lang.Object contiene una implementación ***native y protected*** del método ***clone()***. Esta implementación del método clone() determina cuánta memoria está siendo usada por el objeto a ser clonado, reserva la misma cantidad de memoria para el objeto clon, y **copia el estado del objeto original sobre el nuevo**. Y al final se devuelve un java.lang.Object el cual es la referencia al nuevo objeto (el clon).

Pensemos qué consecuencias tiene la afirmación resaltada, considerando que los atributos de un objeto pueden ser tanto primitivas como objetos.

En el caso de que el atributo sea de tipo primitivo, lo que se copia es el valor que contiene.

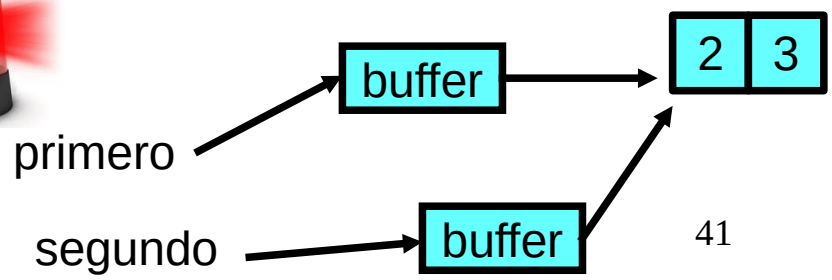
En el caso de que el atributo sea de tipo Clase:

- **Que será lo que se copia?**
- **Que implica esto, teniendo en cuenta que, luego de aplicar clone() habrá dos objetos con el mismo estado.**

clone() - Ejemplo - consecuencias

```
public class PilaEnteros implements
   Cloneable
{  private int[] buffer;
   private int tope;

   public PilaEnteros(int contenMax)
   {   buffer = new int[contenMax];
       tope = -1;
   }

   public void meter(int val)
   {   buffer[++tope] = valor;
   }

   public int sacar()
   {   return buffer[tope--];
   }
}
```


```
PilaEnteros primera = new PilaEnteros(2);

primero.meter(2);

primero.meter(9);

PilaEnteros segunda=(PilaEnteros)primero.clone();
```



Problemas de utilizar la implementación por defecto del método clone()

La clase java.lang.Object contiene una implementación ***native y protected*** del método ***clone()***. Esta implementación del método clone() determina cuánta memoria está siendo usada por el objeto a ser clonado, reserva la misma cantidad de memoria para el objeto clon, y **copia el estado del objeto original sobre el nuevo**. Y al final se devuelve un java.lang.Object el cual es la referencia al nuevo objeto (el clon).

En el caso de que el atributo sea de tipo Clase:

- **Que será lo que se copia?**
  - **Se copia la referencia del objeto contenida en el parámetro**
- **Que implica esto, teniendo en cuenta que, luego de aplicar clone() habrá dos objetos con el mismo estado.**
  - **Implica que los dos objetos tendrán atributos que referirán al mismo objeto.**

clone() - Ejemplo - solución

```
public Object clone()
{
    try
    {
        PilaEnteros nObj = (PilaEnteros)super.clone();
        nObj.buffer = (int[])buffer.clone();
        return nObj;
    }
    catch(CloneNotSupportedException e)
    {
        throw new InternalError(e.toString);
    }
}
```

Primero el método **`clone`** invoca a **`super.clone()`**

`super.clone()` puede invocar al método **`Object.clone()`** que crea un objeto del tipo correcto

Si la implementación de **`clone()`** de PilaEnteros utilizara **`new`** para crear un objeto PilaEnteros, sería incorrecto para cualquier objeto que extendiera de PilaEnteros

TRY, CATCH, FINALLY

Salto en el tiempo,
adelanto de Excepciones

CORRESPONDE AL TEMA
EXCEPCIONES

Una excepción es un evento que ocurre durante la ejecución del programa que interrumpe el flujo normal de las sentencias. O sea, algo que altera la ejecución normal.

Muchas clases de errores pueden generar excepciones -- desde problemas de hardware, como la avería de un disco duro, a los simples errores de programación, como tratar de acceder a un elemento de un array fuera de sus límites.

```
X = A / 0;
```

Espacio diferente de solución

Comprobar errores en forma clara

Código limpio

```
Metodo(...) throws Zona de propagación del error
{
    try
    {
        Zona de ejecución normal
    }
    catch( Exception e)
    {
        Zona de recuperación
    }
    finally
    {
        Zona de finalización
    }
}
```

```
todo(...) throws
          Zona de propagación del error
{
    try
    {
        Zona de ejecución normal
    }
    catch( Exception e)
    {
        Zona de recuperación
    }
    finally
    {
        Zona de finalización
    }
}
```

Dentro de la zona denominada **`try`**, se establecen las instrucciones que conforman el código “normal”, el que determina la funcionalidad de método.
Si alguna de estas instrucciones provoca un problema (excepción), no se continúa ejecutando ninguna de las siguientes instrucciones del bloque **`try`** y se provoca el lanzamiento de una excepción (creación de un objeto de tipo Exception y “lanzamiento”).

```
Metodo(...)
{
    try
    {
        Zona de ejecución normal
    }
    catch( Exception e)
    {
        Zona de recuperación
    }
    finally
    {
        Zona de finalización
    }
}
```

throws
Zona de propagación del error

El lanzamiento de un objeto de tipo Exception altera el flujo normal del programa.

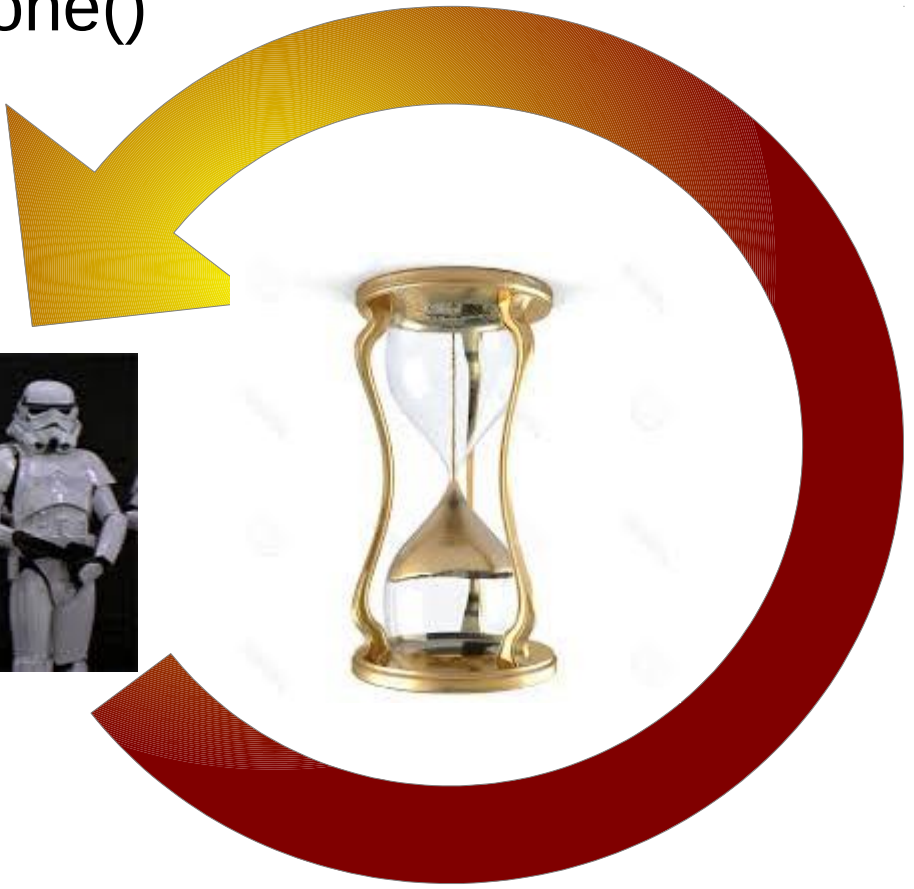
La zona **`catch`** se encarga de recibir por parámetro dicho objeto y en base a los datos aportados por el mismo, se elabora una estrategia de solución al problema ocurrido.

```
todo(...) throws
          Zona de propagación del error

try
{
        Zona de ejecución normal
}
catch( Exception e)
{
        Zona de recuperación
}
finally
{
        Zona de finalización
}
}
```

Si el cuerpo del bloque try llega a comenzar su ejecución, el bloque finally siempre se ejecutará...

- Detrás del bloque try si no se producen excepciones
- Después de un bloque catch si éste captura la excepción.
- Justo después de que se produzca la excepción si ninguna cláusula catch captura la excepción y antes de que la excepción se "***propague hacia arriba".***

Volviendo al tema clone()



clone() - Ejemplo - solución

```
public Object clone()
{
    try
    {
        PilaEnteros nObj = (PilaEnteros)super.clone();
        nObj.buffer = (int[])buffer.clone();
        return nObj;
    }
    catch(CloneNotSupportedException e)
    {
        throw new InternalError(e.toString);
    }
}
```

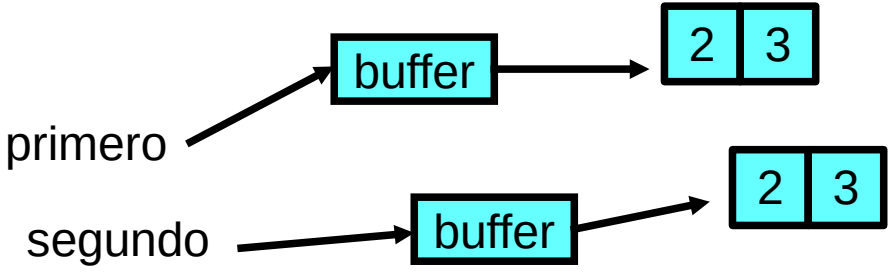
Primero el método **`clone`** invoca a **`super.clone()`**

`super.clone()` puede invocar al método **`Object.clone()`** que crea un objeto del tipo correcto

Si la implementación de **`clone()`** de PilaEnteros utilizara **`new`** para crear un objeto PilaEnteros, sería incorrecto para cualquier objeto que extendiera de PilaEnteros

clone() - Cómo funciona

```
public Object clone()
{
    try
    {
        PilaEnteros nObj = (PilaEnteros)super.clone();
        nObj.buffer = (int[])buffer.clone();
        return nObj;
    }
    catch(CloneNotSupportedException e)
    {
        throw new InternalError(e.toString);
    }
}
```

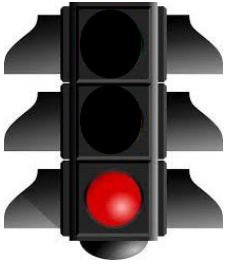


- ✔ **`Object.clone()`** inicializa todos los campos del nuevo objeto clonado asignándoles los mismos valores que los campos correspondientes del objeto que está siendo clonado.
- ✔ Sólo es necesario escribir código especial para aquellos campos en los que copiar el valor sea incorrecto.
- ✔ Los arrays pueden ser clonados.
- ✔ Si tenemos otros objetos (atributos) que no pueden ser copiados directamente, éstos deberán soportar **`clone()`** o deben tener un *constructor de copia*, como la clase String, que no soporta **`clone()`**, pero tiene un constructor de copia (depende de la versión de Java).

Actitudes frente a `clone()`:

Admitir **`clone`**. La clase implementa **`Cloneable`** y declara su método clone para que no lance excepciones

Admitir **`clone`** condicionalmente. Depende de si sus componentes pueden ser clonados. Este tipo implementará **`Cloneable`**, pero permitirá que su método clone pase cualquier excepción CloneNotSupportedException que pueda recibir de otros objetos que intenta clonar.



Prohibir **`clone`**. Una clase así no implementa **`Cloneable`** y proporciona un método clone que lanza siempre la excepción **`CloneNotSupportedExeption.`**

Estrategias de clonación

Hay tres factores a tener en cuenta:

1. La interfaz **Clonable** que debemos implementar para proporcionar un método **clone**.

2. El método **clone** implementado por la clase **Object**, que realiza un clon simple copiando todos los campos del objeto original en un nuevo objeto.

3. La excepción **CloneNotSupportedException** se puede utilizar para indicar que no se debería haber llamado a un método **clone** de una clase.

Entonces...

- Las clases que no deban admitir **`clone`** deben tener redefinido este método, el cual debe lanzar siempre la excepción **`CloneNotSupportedException.`**
- Puede declararse que todas las subclases de una clase deben admitir correctamente el método **`clone`**, redefiniendo **`clone`** de la clase con otro método que elimine la declaración de **`CloneNotSupportedException.`**
- Las subclases que implementen al método **`clone`** no podrán lanzar la excepción **`CloneNotSupportedException`**, ya que los métodos de una subclase no pueden añadir excepciones a un método.