Ministerio de Producción
Presidencia de la Nación

Ministerio de Educación y Deportes

Subsecretaría de Servicios Tecnológicos y Productivos

Programa 111 mil
VOS PODÉS SER UNO.

Interfaces

Programación

- Interfaces: Definición e implementación
- Interfaces como tipos de datos
- Interfaces comparadas con clases abstractas
- La inteface **Comparable**
- Ejercicios prácticos en PC

Interfaces: Definición

- Como hemos visto hasta ahora, Java provee 2 formas de definir clases para modelar tanto tipos de datos en sí como su comportamiento dentro de una aplicación: **clases concretas** y **clases abstractas**
- Repasemos sus características:

	Puede contener métodos abstractos?	**Puede contener métodos concretos?**	**Puede contener atributos?**	**Puede heredar de...**
Clase concreta	No	Sí	Sí	Otras clases concretas o abstractas
Clase abstracta	Sí	Sí	Sí	Otras clases concretas o abstractas

Interfaces: Definición

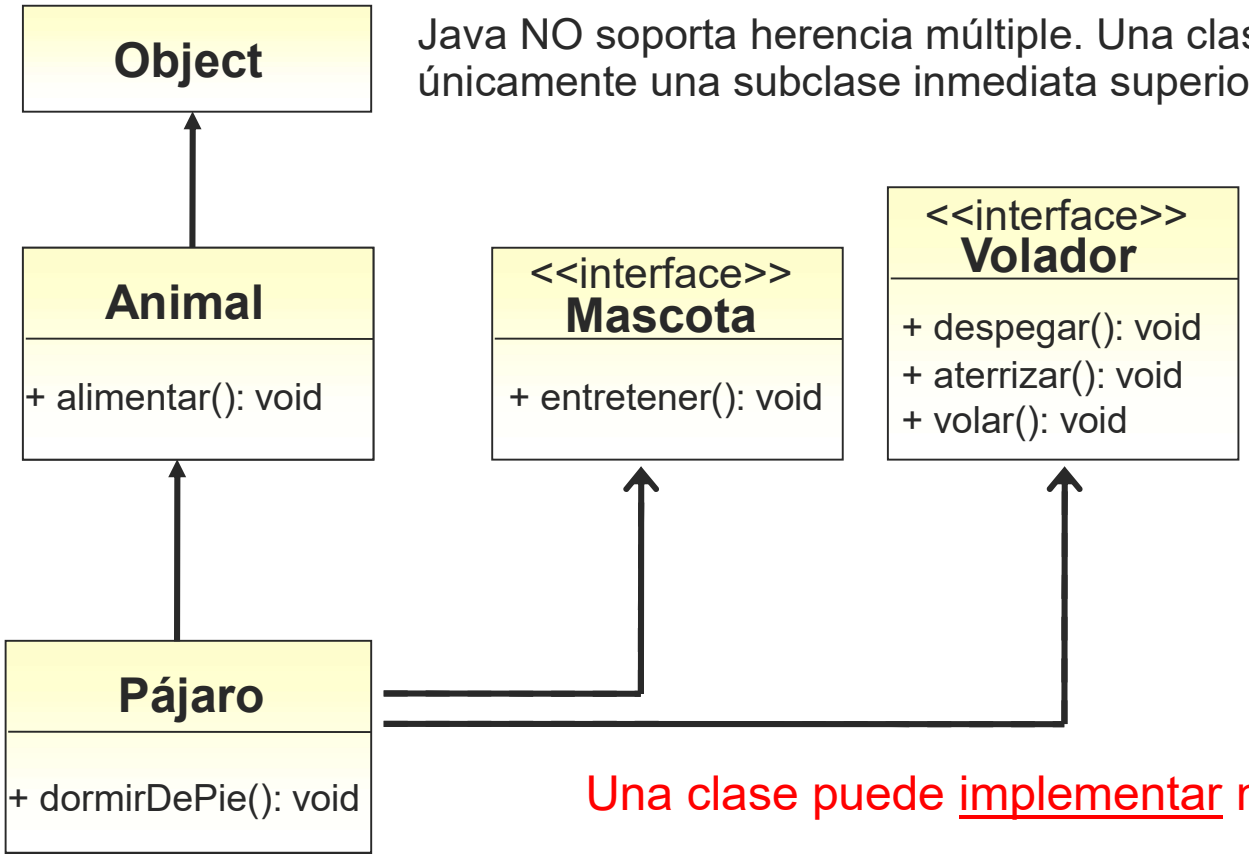
- Un tercer mecanismo adicional a las clases concretas y abstractas provisto por Java son las **interfaces**, las cuales son **completamente abstractas**
- Comparado con las clases concretas y las clases abstractas, las interfaces:

	Puede contener métodos abstractos?	**Puede contener métodos concretos?**	**Puede contener atributos?**	**Puede heredar de...**
Clase concreta	No	Sí	Sí	Otras clases concretas o abstractas
Clase abstracta	Sí	Sí	Sí	Otras clases concretas o abstractas
Interfaces	**Sí**	**No**	**No**	**Otras interfaces únicamente**

- La clase que implementa una clase abstracta **debe ser** una subclase de la clase abstracta. Hay 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

Interfaces: Definición

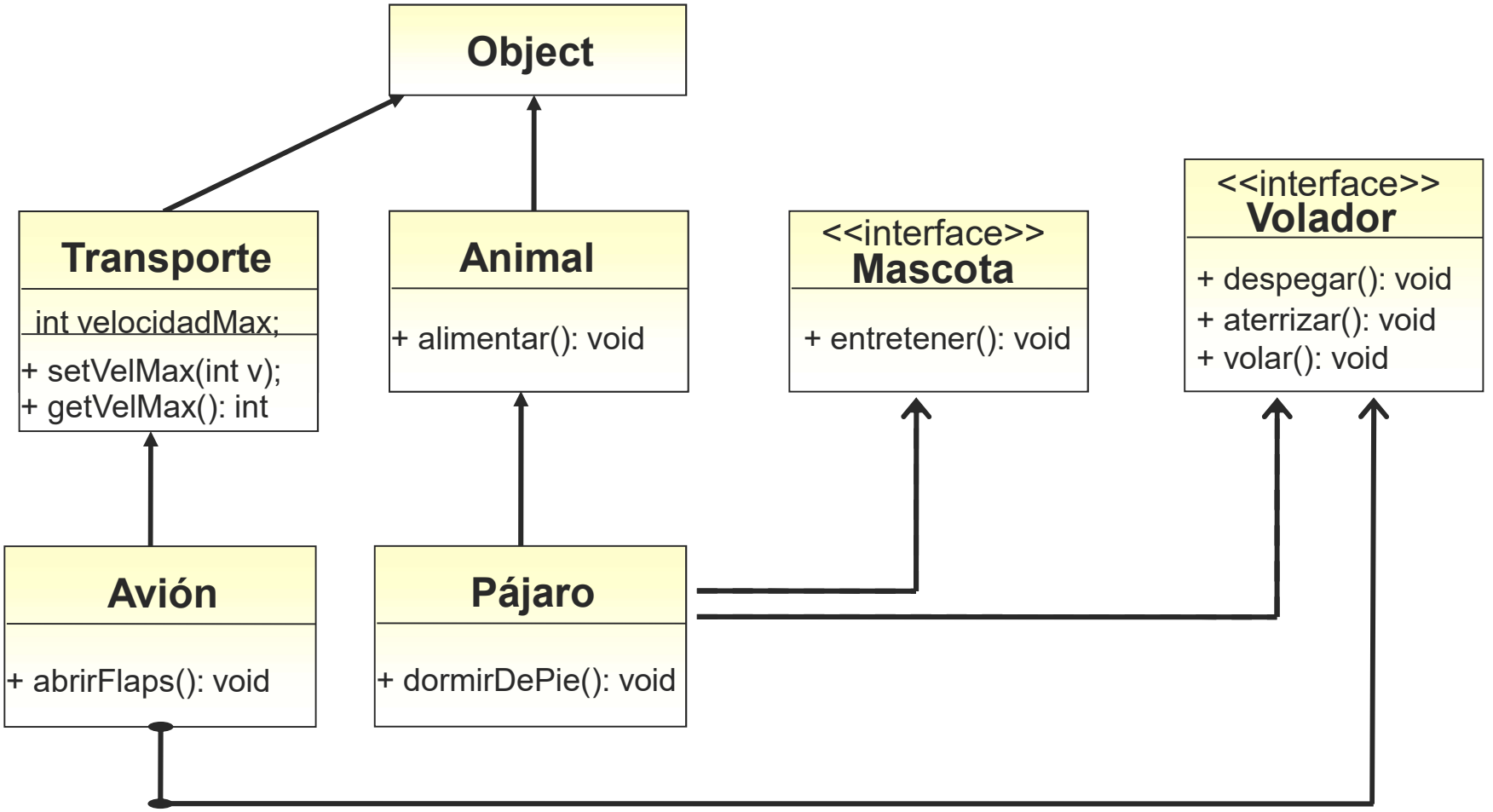
¿Puede Pájaro ser subclase de Animal, Mascota y Voladores? **NO!!!**



Java NO soporta herencia múltiple. Una clase puede tener únicamente una subclase inmediata superior

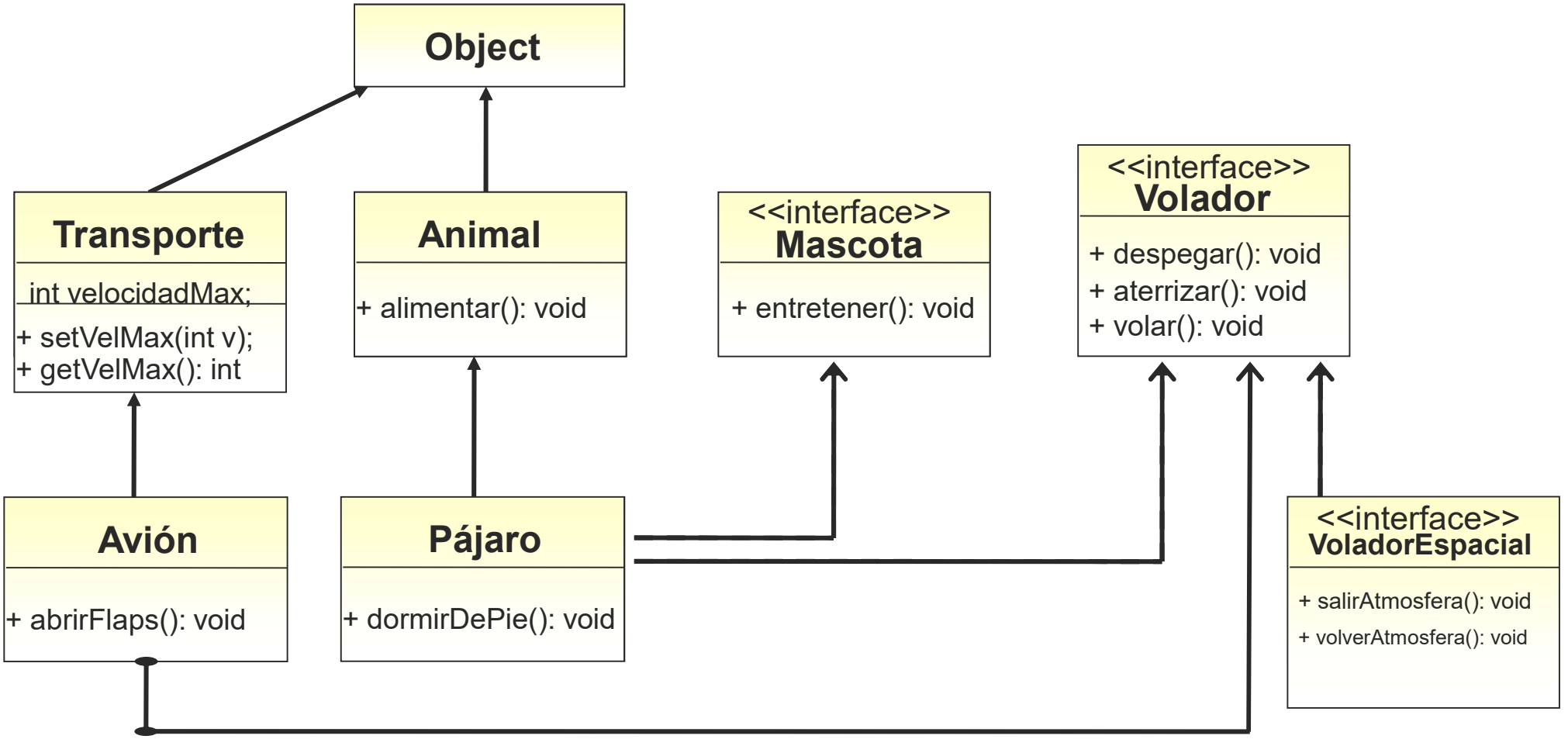
Una clase puede implementar múltiples interfaces

Interfaces: Definición



Una interface puede ser implementada por múltiples clases

Interfaces: Definición



Una interface puede ser implementada por múltiples clases **o interfaces**

Interfaces: Declaración

- Para declarar una interface se utiliza la palabra clave **interface**

```
package interfaces;
public interface Volador {
     long UN_SEGUNDO = 1;
     long UN_MINUTO = 60;
     void aterrizar();
     void despegar();
     void volar();
}
```

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

```
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

- Las constantes son implícitamente **public**, **static** y **final**. No es necesario escribirlo en el código
- Los métodos son implícitamente **public** y **abstract**. Tampoco es necesario escribirlo en el código

Interfaces: Uso

- Luego de declarar la interface, definimos la clase que la implementa utilizando la palabra clave **implements**:

```
package interfaces;
public class Avion extends Vehiculo implements Volador {
     void Avion(){…} // constructor
     void abrirFlaps();
     void despegar(){...};
     void aterrizar(){...};
     void volar(){...};
}
```

- **Avion** heredará todas las constantes de **Volador**
- También es posible la herencia entre interfaces mediante la palabra clave **extends**. Por ejemplo, podemos definir una interface **MascotaVoladora** que extiende las interfaces **Volador** y **Mascota:**

```
package interfaces;
public interface MascotaVoladora extends Volador, Mascota {
          void enjaular();
}
```

- **MascotaVoladora** heredará todas las constantes y métodos abstractos de sus superinterfaces (**Volador**, **Mascota**)

Interfaces: Uso

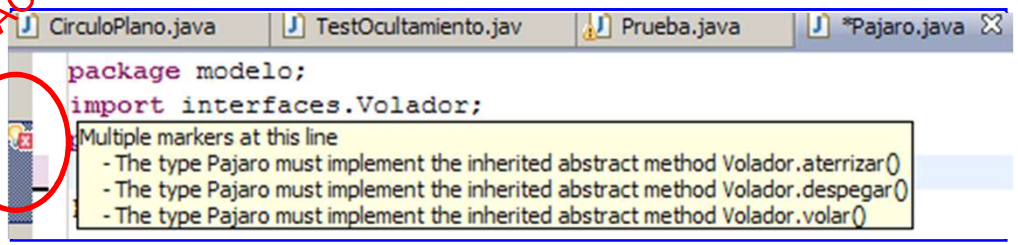
```
package modelo;
import interfaces.Volador;
public class Pajaro implements Volador {
}
```

¿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!!!



Interfaces: Implementación parcial de métodos

- Como alternativa a implementar todos los métodos, es posible implementar parte de los métodos de la superinterface y declarar la clase como abstracta:

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

- Considerando que la clase **Volador** define los métodos abstractos **despegar()**, **volar()** y **aterrizar()**, ¿qué opciones existen para generar versiones abstractas de la clase **Pajaro**?

implementar 1 método solamente (despegar, volar, o bien aterrizar)
implementar pares de métodos
implementar los 3 implicaría que la clase **Pajaro** sea concreta!

Interfaces: Implementación total parcial de métodos

```
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() ");
}
   ........
}
```

Otros 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** para ser concreta

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 = 1;
     long UN_MINUTO = 60;
     void aterrizar();
     void despegar();
     void volar();
}
```

Interfaces, accesors y paquetes

¿Puede la clase **Pajaro** implementar la interface Mascota? **NO!!!**

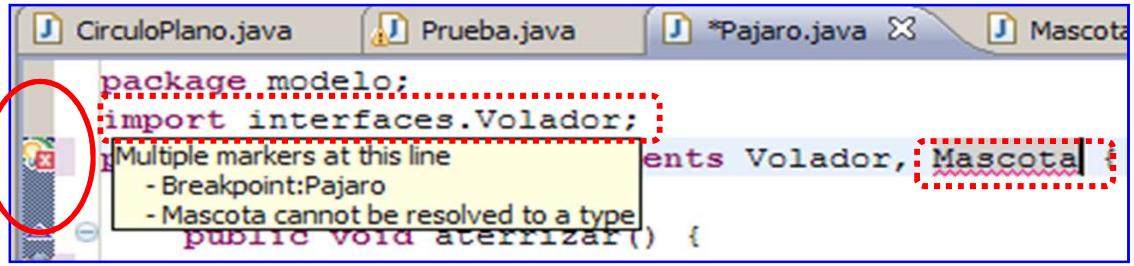
```
package modelo;
import interfaces.Volador;
public class Pajaro implements Volador {
   ...
}
```

```
package interfaces;

interface Mascota {

    void entretener();
}
```

ERROR DE COMPILACIÓN



- Se produce un error de compilación: **Mascota** tiene alcance **package** y **Pajaro** se encuentra en un paquete distinto (interfaces). **Pajaro NO tiene acceso a la interface Mascota. La interface Mascota SOLO puede se implementada por clases de su mismo paquete (interfaces)**

Interfaces, accesors y paquetes

- ¿Qué modificación debemos realizar en la interface **Mascota** para que pueda ser implementada por cualquier clase, de cualquier paquete?

```
package interfaces;

public interface Mascota {

    void entretener();
}
```

Transformarla en una interface **pública**

Pajaro debe implementar TODOS los métodos definidos en **Volador** y **Mascota**

Volador.java | Pajaro.java | Mascota.java | TestPajaro.java | 42

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

    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 entretener() {
        System.out.println("Pajaro.entrener()");
    }
    public void dormirDePie() {
        System.out.println("Pajaro.dormirDePie()");
    }
}
```

Interfaces: Invocación

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?

Pajaro.dormirDePie()

Pajaro.despegar() durante 1

Pajaro.aterrizar()

Pajaro.volar() durante 60

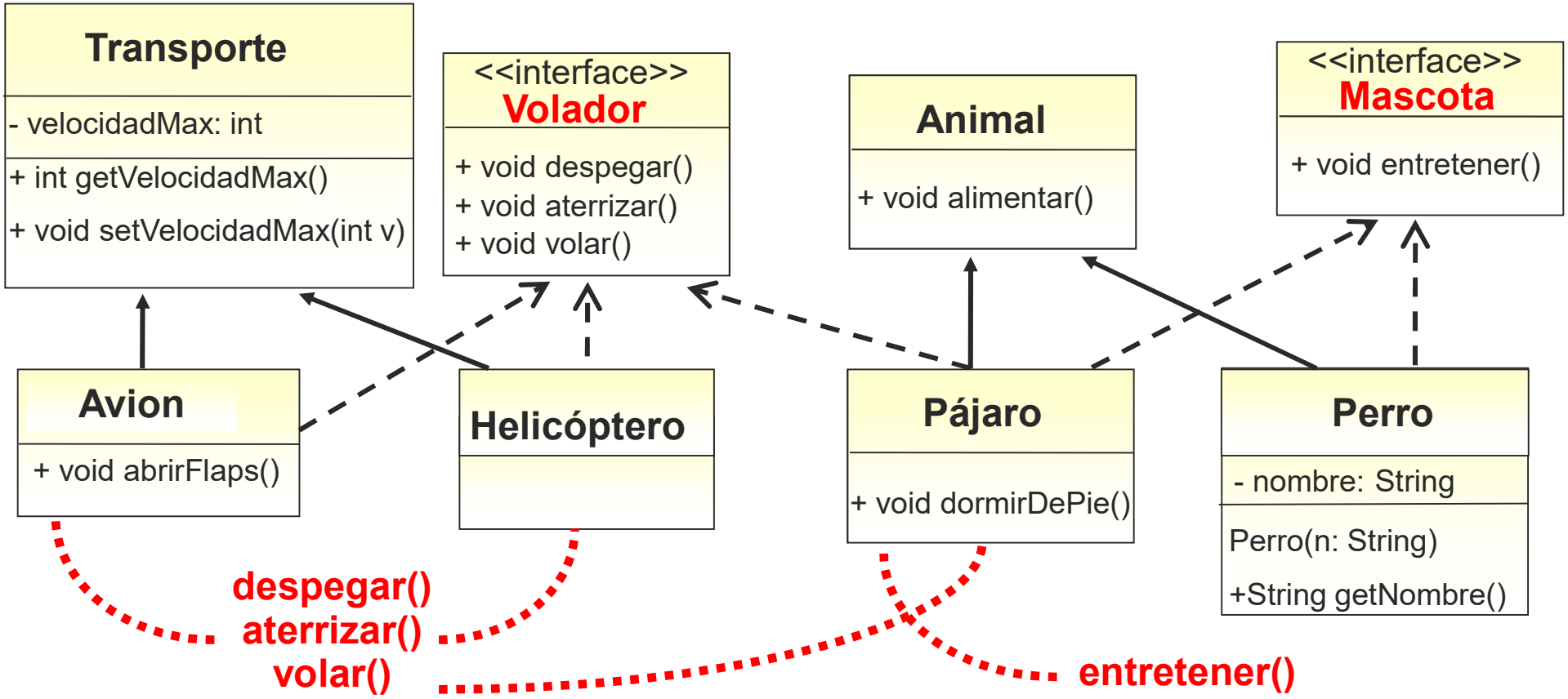
Pajaro.entretener()

Es un mecanismo similar a la herencia múltiple, aunque NO lo es porque de las interfaces a diferencia de las clases abstractas, NO existe la posibilidad de heredar métodos concretos

Interfaces: Tipificación

Las instancias de Avion, Pajaro y Helicoptero son de tipo Volador

Instancias de clases no relacionadas por la cadena de herencia son del MISMO TIPO!!!



Interfaces: Tipificación

Pajaro.java | Transporte.java | *Avion.java | Animal.java

```
package modelo;
import interfaces.Mascota;
import interfaces.Volador;
public class Pajaro extends Animal implements Volador, Mascota {

    public void aterrizar() {
        System.out.println("Pajaro.aterrizar()");
    }

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

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

    public void entretener() {
        System.out.println("Pajaro.entrener()");
    }
    public void dormirDePie() {
        System.out.println("Pajaro.dormirDePie()");
    }
}
```

Avion y **Pajaro** pertenecen a jerarquías de clases diferentes, sin embargo **los objetos Avion y Pajaro son de tipo Volador**

Pajaro.java | TestPajaro.java | Transporte.java | *Avion.java

```
package modelo;
import interfaces.Volador;
public class Avion extends Transporte implements Volador {

    public void aterrizar() {
        System.out.println("Avion.aterrizar()");
    }

    public void despegar() {
        System.out.println("Avion.despegar()");

    }

    public void volar() {
        System.out.println("Avion.volar()");

    }
}
```

Interfaces: Asignación a variables y casting

- Las interfaces son tipos de datos: podemos declarar variables cuyo tipo es una interface
- Dos clases de jerarquía diferentes que implementan la misma interface pueden ser “casteadas” al tipo de la interface o asignadas a variables de tipo de la interface
- Veamos ejemplos de operaciones de **asignación/casting** válidas:

```
Volador v = null;
Pajaro p = new Pajaro();
Avion a = new Avion();
v = (Volador)p;    // equivalente a v = p;
v = (Volador)a;    // equivalente a v = a;
```

- Lo anterior funciona sin importar si las clases involucradas (**Pajaro**, **Avion** en este caso) son clases concretas o clases abstractas

Interfaces: Casos donde el casting es necesario

- Supongamos ahora el siguiente caso:

```
Volador v1, v2 = null;
Mascota m1, m2 = null;
```

- Estas variables deberán contener referencias a objetos que implementen las interfaces **Volador** y **Mascota,** respectivamente:

```
v1 = new Avion();
v2 = new Pajaro();
m1 = new Perro();
m2 = new Pajaro();
```

```
if (v1 instanceof Avion) {
    Avion a=(Avion) v1;
    a.getVelocidad();
}
```

```
if (m1 instanceof Perro) {
    Perro p=(Perro) m1;
    p.getNombre();
}
```

¿Qué métodos tengo accesibles en los objetos v1 y v2? despegar(), aterrizar() y volar()

¿Puedo pedirle a v1 que me devuelva su velocidad y a v2 que duerma de pie?

NO directamente!!! podría hacerlo si previamente "casteo" el objeto en uno de tipo Avion o Pajaro respectivamente (downcasting)

y ¿al objeto m1, puedo pedirle su nombre y a m2 que se alimente?

NO!!! También requiere downcasting

Interfaces y polimorfismo

- *Polimorfismo: Capacidad de los objetos de responder de forma diferente al mismo mensaje...*

```
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) {




    }
}
```

El método despegar() es polimórfico:

- El método **despegar()** que se ejecutará está determinado por el tipo real del objeto **v** (**Avion**, **Helicoptero** y **Pajaro**)
- En nuestro caso, más de una clase implementó la interface **Volador** y en consecuencia tenemos múltiples versiones del método **despegar()**. Java “se da cuenta” a que método **despegar()** debe invocar

Interfaces y polimorfismo

- *Polimorfismo: Capacidad de los objetos de responder de forma diferente al mismo mensaje...*

```
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 Ave();
        for (int j=0; j<m.length; j++)
        partida(m[j]);
    }
}
```

El método despegar() es polimórfico:

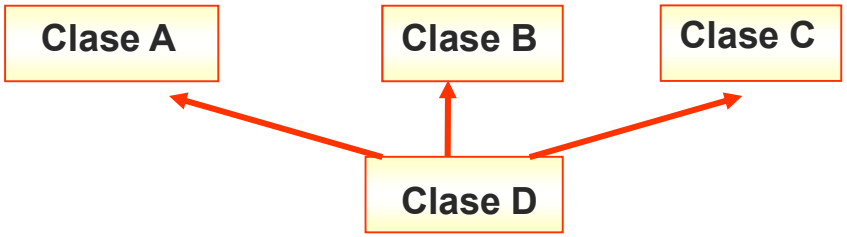
- El método **despegar()** que se ejecutará está determinado por el tipo real del objeto **v** (**Avion**, **Helicoptero** y **Pajaro**) y NO por el tipo de la variable, **Volador.**

- En nuestro caso, más de una clase implementó la interface **Volador** y en consecuencia tenemos múltiples versiones del método **despegar()**. Java “se da cuenta” a que método **despegar()** debe invocar

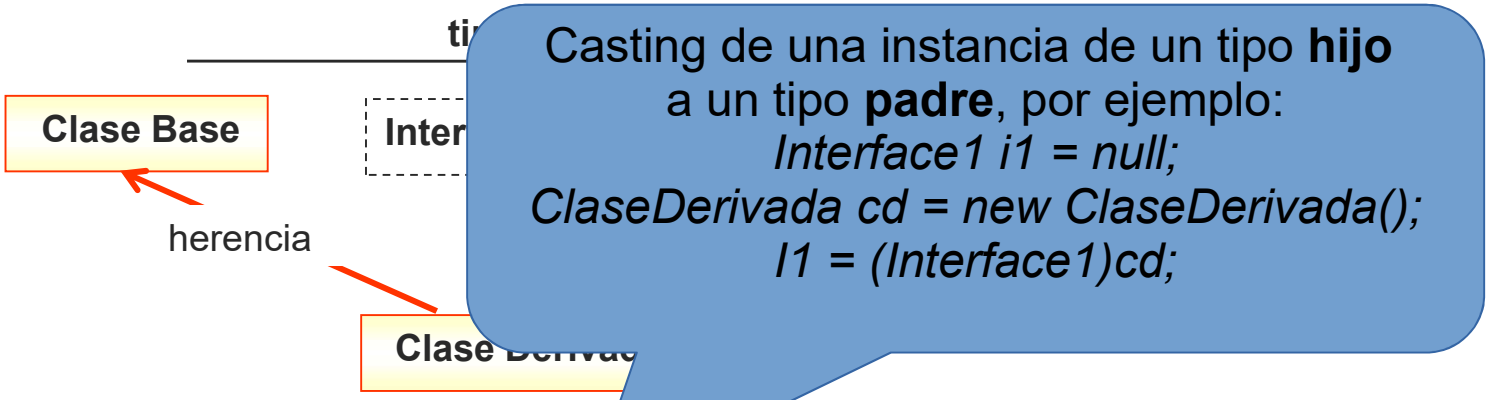
Declaramos un arreglo de tipo **Volador: cada objeto del arreglo debe ser de tipo Volador**

Interfaces y herencia múltiple

- La **Herencia Múltiple** es el mecanismo de los lenguajes de programación orientados a objetos que permite crear una nueva clase derivándola de múltiples superclases
- Cada clase tiene su propia implementación, por lo tanto combinarlas es complejo



Java NO soporta herencia múltiple pero provee interfaces para lograr un comportamiento “similar”. Como las interfaces NO tienen implementación, NO hay problemas al combinarlas, entonces: una clase puede **heredar de una única clase base** e **implementar múltiples interfaces**



- Cada una de las interfaces que la clase implementa provee de un tipo de dato al que puede aplicarse **casting (upcasting)**

Interfaces en la API de Java

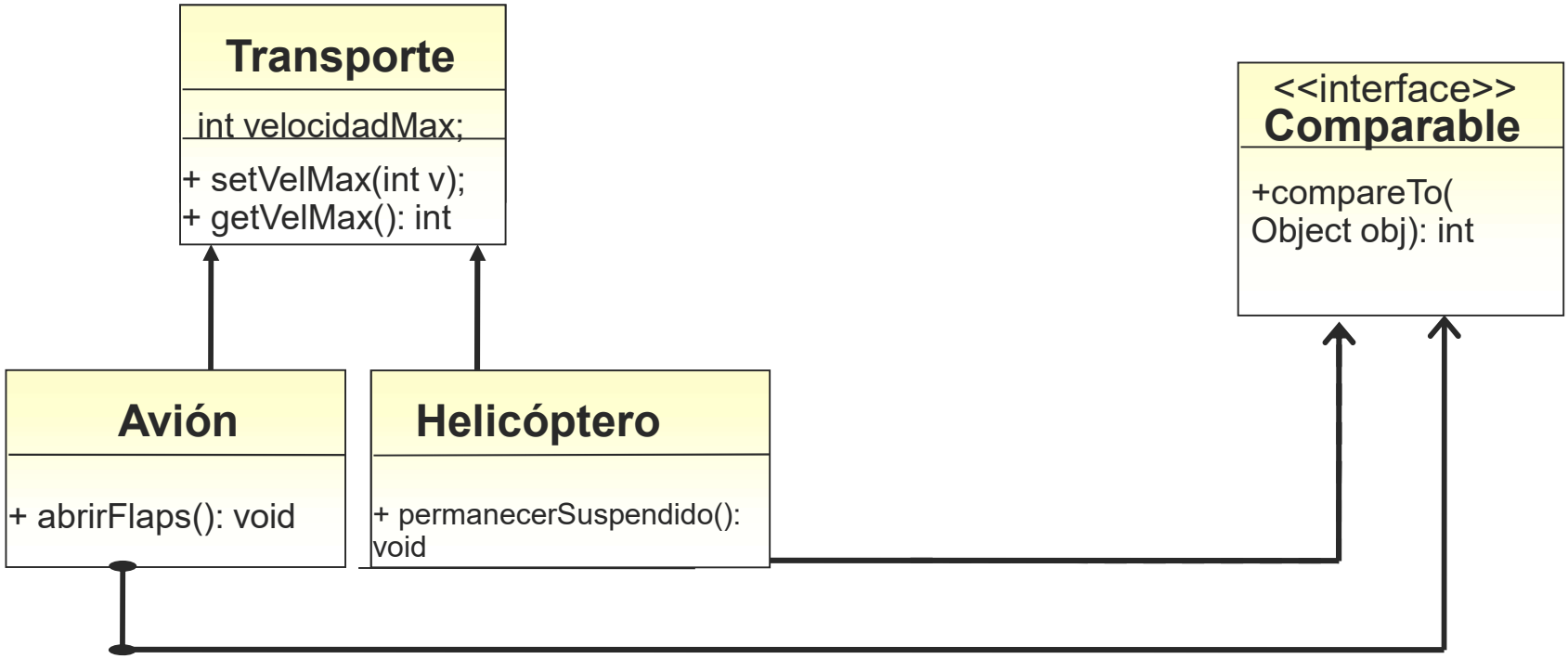
- La API de Java, al igual que con las clases abstractas, hace un uso importante de interfaces internamente, y además provee definiciones de interfaces para usar en aplicaciones de usuario
- Nos focalizaremos en una importance interface de este último grupo:

Comparable: Esta interfaz establece métodos abstractos para permitir la comparación entre instancias “ordenables” (<, =, >)

```
int compareTo(Object otroObjeto);
// < 0 → objeto menor que parámetro
// = 0 → objeto igual que parámetro
// > 0 → objeto mayor que parámetro
```

Ejemplo de Comparable

- Tomando los diferentes tipos de transportes ya definidos (**Avion**, **Helicoptero**, etc.) queremos que sus instancias sean comparables de acuerdo a su velocidad máxima



Ejemplo de Comparable

```
abstract class Transporte {
 int velocidadMax;
 public void setVelMax(int vel) {
  velocidadMax = vel;
 }
 public int getVelMax() {
  return velocidadMax;
 }
}

class Avion extends Transporte
        implements Comparable {

 public void abrirFlaps(){}
 public int compareTo(Object obj) {
  Transporte t = (Transporte)obj;
  if (velocidadMax < t.getVelMax())
   return -1;
  if (velocidadMax == t.getVelMax())
   return 0;
  return -1;
 }
}
```

```
class Helicoptero extends Transporte
        implements Comparable {

 public void permanecerSuspendido(){}
 public int compareTo(Object obj) {
  Transporte t = (Transporte)obj;
  if (velocidadMax < t.getVelMax())
   return -1;
  if (velocidadMax == t.getVelMax())
   return 0;
  return -1;
 }
}
```

Existe alguna mejora en el código que podría aplicarse?

Ejemplo de Comparable

```
abstract class Transporte implements Comparable {
 int velocidadMax;
 public void setVelMax(int vel) {
  velocidadMax = vel;
 }
 public int getVelMax() {
  return velocidadMax;
 }
 public int compareTo(Object obj) {
  Transporte t = (Transporte)obj;
  if (velocidadMax < t.getVelMax())
   return -1;
  if (velocidadMax == t.getVelMax())
   return 0;
  return -1;
 }
}
```

```
class Helicoptero extends Transporte {
 public void permanecerSuspendido(){...}
}

class Avion extends Transporte {
 public void abrirFlaps(){...}
}
```

De esta manera, se elimina la duplicación de código ya que el criterio de comparación se hereda

=

Pensar bien a qué nivel de la jerarquía se implementan las interfaces!

Ejemplo de Comparable

```
class TestComparable {
 public static void main(String[] args) {
  Avion avion = new Avion();
  Helicoptero helicoptero = new Helicoptero();
  avion.setVelMax(1200);
  helicoptero.setVelMax(500);
  System.out.println(avion.compareTo(helicoptero));
  System.out.println(helicoptero.compareTo(avion));
 }
}
```

¿Cuál es la salida?

1

-1

```
public int compareTo(Object obj) {
  Transporte t = (Transporte)obj;
  if (velocidadMax < t.getVelMax())
   return -1;
  if (velocidadMax == t.getVelMax())
   return 0;
  return -1;
}
```

Comparable y arreglos

- Definir la interface comparable para instancias de ciertas clases implica que es posible ordenar automáticamente un arreglo que contenga instancias de dichas clases
- Ejemplo: Supongamos que tenemos un arreglo de instancias de la clase **Transporte**:

```
class TestComparableOrdenamiento {
 public static void main(String[] args) {
  Avion avion = new Avion();
  Helicoptero helicoptero = new Helicoptero();
  Auto auto = new Auto();
  avion.setVelMax(1200);
  helicoptero.setVelMax(500);
  auto.setVelMax(200);
  Transporte[] arreglo = new Transporte[]{avion, helicoptero, auto};
  ...
 }
}
```

Comparable y arreglos

- Utilizando el método Arrays.sort de la API de Java, podemos ordenar automáticamente el arreglo ya creado!

```
class TestComparableOrdenamiento {
 public static void main(String[] args) {
  Avion avion = new Avion();
  Helicoptero helicoptero = new Helicoptero();
  Auto auto = new Auto();
  avion.setVelMax(1200);
  helicoptero.setVelMax(500);
  auto.setVelMax(200);
  Transporte[] arreglo = new Transporte[]{avion, helicoptero, auto};
  Arrays.sort(arreglo);
  // Puedo usar reglo ordenado a partir de acá
 }
}
```

- Para pensar: ¿En qué orden quedarían los elementos del arreglo luego del ordenamiento? ¿Qué debería hacer si quiero invertir el criterio de ordenamiento?

Interfaces: Resumen de características #1

- Una interface es una **colección de definiciones de métodos abstractos** sin implementación y opcionalmente con **declaraciones de constantes**, agrupados bajo un nombre
- Al igual que las clases abstractas, **no es posible crear instancias de una interface**
- Una **clase debe implementar la/las interface/s** que declara y así proveer el comportamiento necesario de los métodos declarados en la interface (ejemplo: Avion → Volador)

Interfaces: Resumen de características #2

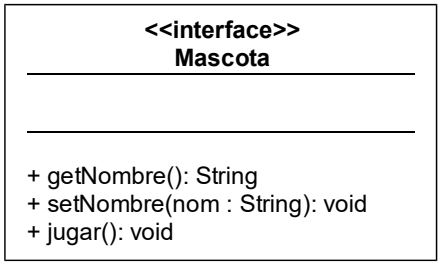
- Una clase que implementa una interface hereda las constantes de la interface
- Una clase que no implementa todos los métodos de sus superinterfaces **debe ser declarada como abstracta**
- Puede existir herencia entre interfaces
- Una interface establece **qué** debe hacer la clase que la implementa, sin especificar el **cómo**. Una instancia de dicha clase, es del **tipo** de la clase y de la interface

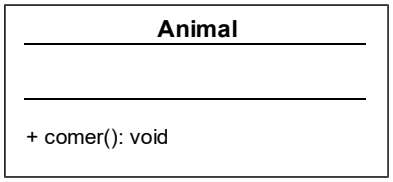
Mini trivia antes de codificar...

- ¿Puede tener una **interface** un método con código? ¿Por qué?
- ¿Puede tener una **interface** atributos? ¿Y constantes?
- ¿Cuántas interfaces puede implementar una clase?
- ¿Podemos definir una interface como subinterface de múltiples interfaces? ¿Pueden mostrar algún ejemplo?

Ejercicios en máquina – Parte 1

- Declare la interface pública **Mascota** en un paquete llamado **interfaces**, y la clase abstracta **Animal**, dadas por las siguientes definiciones:





- Codifique 2 clases públicas que implementen la interface **Mascota**: **Gato** y **Pez**, subclases de **Animal**. Ambas clases deben declarar una variable de instancia, **nombre**, de tipo String, que contendrá el nombre de la mascota y, un constructor con un argumento String que permita inicializar la variable

- El método **jugar()** imprimirá en pantalla el tipo de animal está invocando al método (por ejemplo: Gato.jugar() o Pez.jugar()).

- Ambas clases sobre-escriben el método **comer()** heredado de la clase Animal.

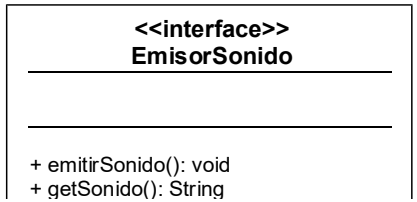
¿De cuántas versiones del método **jugar()** dispone el código desarrollado?

Ejercicios en máquina – Parte 2

- Codifique la clase **TestAnimales** para probar las implementaciones de **Mascota**, de acuerdo a las siguientes indicaciones:
  - Codifique un método de clase llamado **miMascota(Mascota)** que recibe como parámetro un objeto de tipo **Mascota** y devuelve **void**. La implementación de este método consiste en invocar al método **getNombre()** y **jugar()** del objeto que recibe como parámetro. Imprima por pantalla el nombre de la mascota
  - El método **main()** debe crear un arreglo de tipo **Mascota** que contiene 2 instancias de **Pez**, y 2 de **Gato**. Luego, el método debe recorrer el arreglo e invocar al método **miMascota()** pasándole como parámetro cada uno de los elementos del arreglo
  - ¿El método **jugar()** en cada caso funciona correctamente? ¿Por qué?

Ejercicios en máquina – Parte 3

- Codifique la interface pública **EmisorSonido** en el paquete **interfaces** dada la siguiente definición:



- Codifique la clase pública **Perro** en el paquete **Modelo** como subclase de **Animal** y que implemente las interfaces **Mascota** y **EmisorSonido**. Considere que:
  → para la implementación de la interface **Mascota** tendrá que definir una variable de instancia para guardar el nombre del perro y que se inicializará en el constructor
  → para la implementación de **EmisorSonido** también deberá definir una variable de instancia para guardar el sonido del **Perro** y que se inicializará en un constructor
  → el método **emitirSonido()** deberá imprimir en pantalla el sonido que emite el animal. Por ejemplo, cuando se lo invoca sobre un objeto **Perro** deberá imprimir "Guau!! Guau!! Guau!!!"
  → la clase **Perro** sobre-escribe el método **comer()** heredado de **Animal**

- Ahora también la clase **Gato** deberá implementar la interface **EmisorSonido**. Haga los cambios necesarios para esto, similar a como lo hizo para la clase **Perro**
- ¿De cuántas versiones del método **emitirSonido()**, **jugar()** y **comer** dispone? Discuta qué versión es una sobre-escritura por herencia y cuál por implementación de interfaces

Ejercicios en máquina – Parte 4

- Realice los siguientes cambios en la clase **TestAnimales**:
  - Agregue un método de clase llamado **queSonidoEmite(EmisorSonido)** que recibe como parámetro un objeto de tipo **EmisorSonido** y devuelve **void**. La implementación de este método consiste en invocar al método **emitirSonido()** del objeto que recibe como parámetro
  - En el método **main()** cree un arreglo de tipo **EmisorSonido**, agréguele 2 instancias de **Perro**, y 3 de **Gato**. Luego recorra iterativamente el arreglo e invoque al método **queSonidoEmite()** con cada uno de los elementos del arreglo
- ¿El método **emitirSonido()** funciona correctamente en cada caso? ¿Por qué?

Ejercicios en máquina – Parte 5

- Realice los siguientes cambios en la clase **Animal**:
  - Agregue un atributo de tipo int llamado **peso** y métodos para establecer y devolver dicho atributo
  - Haga que la clase **Animal** implemente la interface **Comparable**, y su implementación retorne un valor entero dependiendo del peso de los animales a comparar
  - Cree una clase **ComparadorPeso** con un método **verificarPeso** que recibe dos instancias de **Animal** como parámetro, e imprimir por pantalla el primer parámetro es “Más liviano”, “Pesa lo mismo” o es “Más pesado” que el segundo parámetro. ¿Cómo implementó **verificarPeso**?
- Cree una clase **TestComparadorPeso** con un método **main()** que cree dos instancias **Gato**, establezca pesos diferentes, cree una instancia de **ComparadorPeso** e invoque a **verificarPeso** con las instancias de **Gato** creadas