

# Interfaces

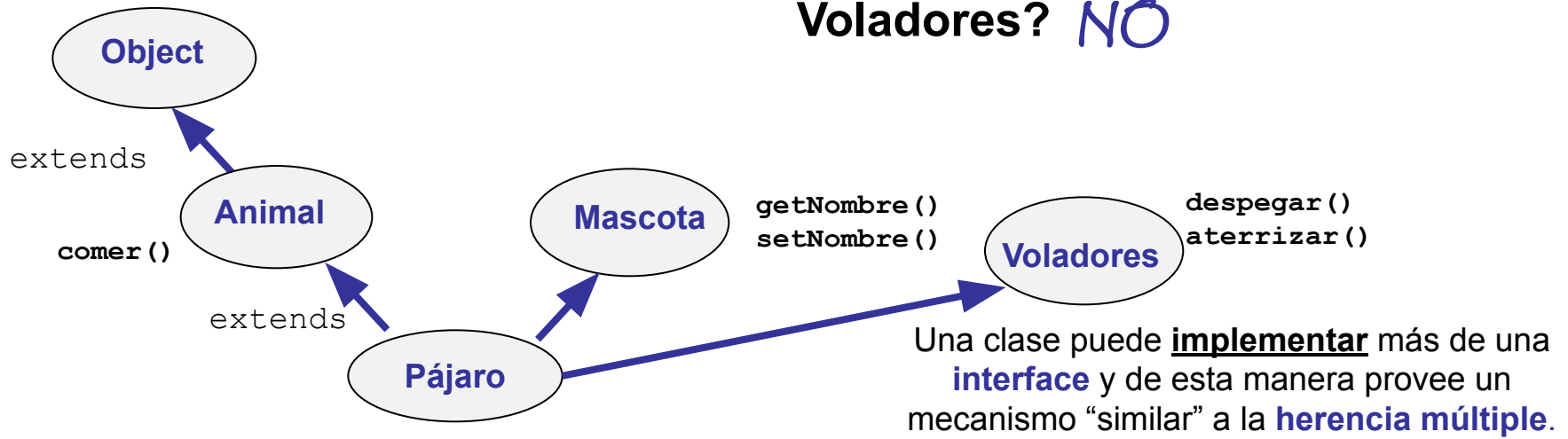
## (en JDK 7 y versiones anteriores)

- Interfaces
  - ¿Qué son las interfaces?
  - ¿Para qué sirven?
- Declaración de interfaces en java
- Un ejemplo:
  - Declarando interfaces
  - Implementando múltiples interfaces
  - Upcasting
- Colisiones en interfaces
- Las interfaces **Comparable**
- Interfaces vs. clases abstractas

# Interfaces

¿Qué son?, ¿Para qué sirven?

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



- Una interface java es una **colección de definiciones de métodos** sin implementación/cuerpo y **de declaraciones de variables de clase constantes**, agrupadas bajo un nombre.
- Las interfaces proporcionan un mecanismo para que una clase defina comportamiento (métodos) de un tipo de datos diferente al de sus superclases.
- Una interface establece **qué** debe hacer la clase que la implementa, sin especificar el **cómo**.

# Declaración de Interfaces

## ¿Cómo se define una interface?

```
package nomPaquete;  
public interface UnaInter extends SuperInter1, SuperInter2, ... {  
    Declaración de métodos: implícitamente public y abstract  
    Declaración de constantes: implícitamente public, static y final  
}
```

- El especificador de acceso **public**, establece que la interface puede ser usada por cualquier clase o interface de cualquier paquete. Si se omite el especificador de acceso, la interface solamente podría ser usada por las clases e interfaces contenidas en el mismo paquete que la interface declarada.
- Una interface puede extender múltiples interfaces. Hay herencia múltiple de interfaces.
- Una interface hereda todas las constantes y métodos de sus **SuperInterfaces**.

# Declaración de Interfaces

- **Ambas declaraciones son equivalentes.** Las variables son implícitamente **public**, **static** y **final** (constantes). Los métodos de una interface son implícitamente **public** y **abstract**.

```
public interface Volador {  
    public static final long UN_SEGUNDO=1000;  
    public static final long UN_MINUTO=60000;  
    public abstract String despegar();  
    public abstract String aterrizar();  
    public abstract String volar();  
}
```

```
public interface Volador {  
    long UN_SEGUNDO=1000;  
    long UN_MINUTO=60000;  
    String despegar();  
    String aterrizar();  
    String volar();  
}
```

- Esta interface **Volador** establece **qué** debe hacer la **clase que la implementa**, sin especificar el **cómo**.
- Las clases que implementen **Volador** deberán implementar los métodos **despegar()**, **aterrizar()** y **volar()**, todos públicos y podrán usar las constantes **UN\_SEGUNDO** y **UN\_MINUTO**. Si una clase no implementa algunos de estos métodos, entonces la clase debe declararse **abstract**.
- Las interfaces se guardan en archivos con el mismo nombre de la interface y con extensión **.java**.

# Implementación de Interfaces

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

```
public class Pajaro
    implements Volador {
    . . .
}
```

```
public interface Volador {
    long UN_SEGUNDO=1000;
    long UN_MINUTO=60000;
    public String despegar();
    public String aterrizar();
    public String volar();
}
```

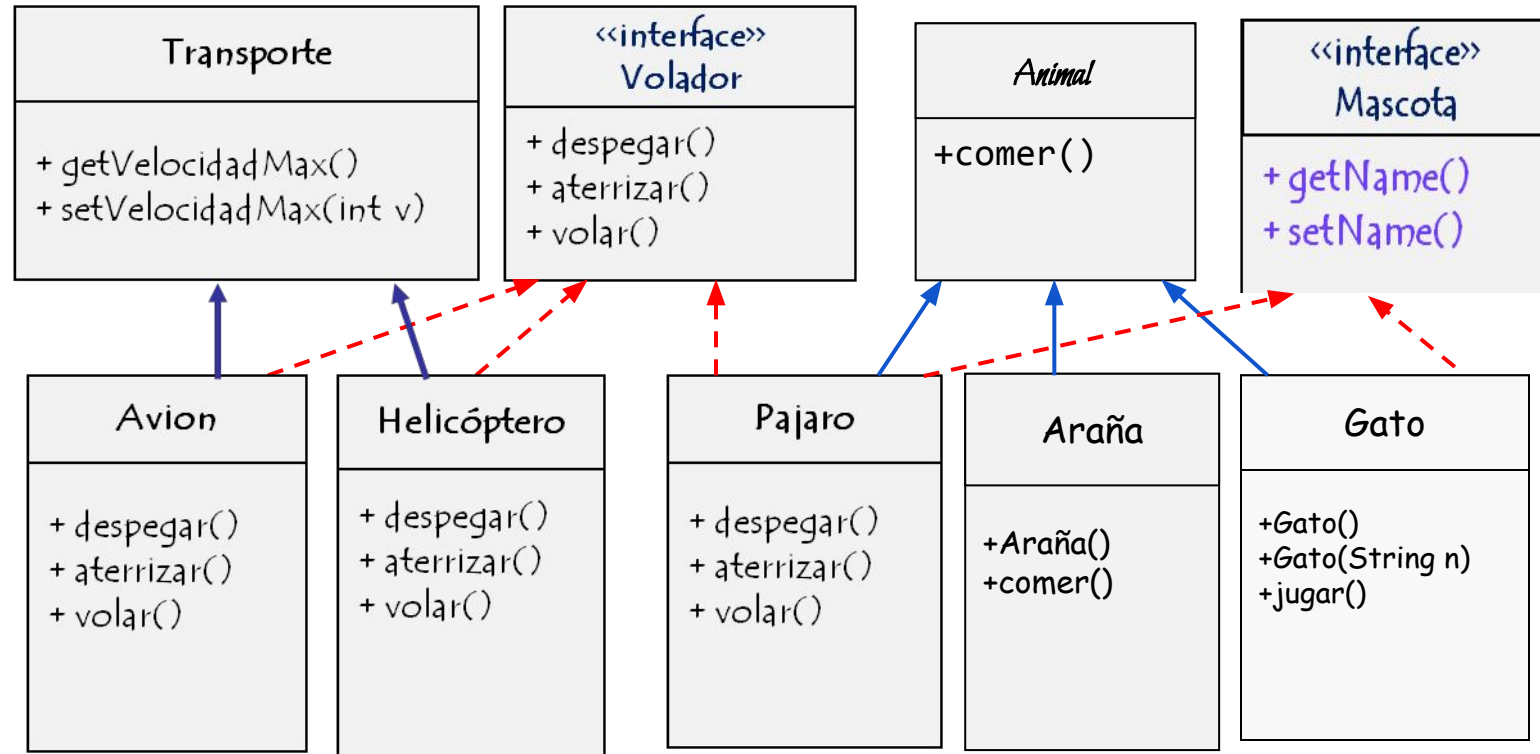
- Una clase que implementa una interface, hereda las constantes y **debe implementar cada uno de los métodos declarados en la interface!!!**.
- Una clase puede implementar más de una interface y de esta manera provee un mecanismo similar a la herencia múltiple.

```
public class Pajaro extends Animal implements Volador, Mascota {
    public String despegar() {...}
    public String aterrizar(){...}
    public String volar(){...}
    public getName(){...}
    public setName(){...}
}
```

```
public interface Macota{
    public getName();
    public setName();
}
```

# Implementación de Interfaces

Considere el ejemplo de una interface que describe cosas que vuelan. El vuelo incluye acciones tales como despegar, aterrizar y volar.



Cada objeto despegue, aterriza y vuela de manera diferente,  
por lo tanto necesita **implementar un procedimiento diferente** para acciones similares

Además de implementar la interface Volador, Pajaro es parte de una jerarquía de clases.

# Implementación de Interfaces

- Cuando una clase implementa una interface se establece como un contrato entre la interface y la clase que la implementa.
- El compilador hace cumplir este contrato asegurándose de que todos los métodos declarados en la interface se implementen en la clase.

```
public class Pajaro extends Animal
    implements Mascota, Volador {
    String nombre;
    public Pajaro(String s) {
        nombre = s;
    }
    // Métodos de la Interface Mascota
    public void setNombre(String nom){
        nombre = nom;}
    public String getNombre(){
        return "El Pájaro se llama "+nombre;}
```

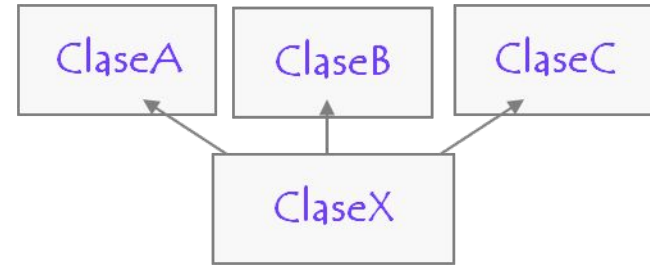
```
// Métodos de la Interface Volador
public String despegar(){
    return("Agitar alas");
}
public String aterrizar(){
    return("Bajar alas");
}
public String volar(){
    return("Mover alas");
}
}
```

```
public class Avion extends Transporte
    implements Volador{
    int velocidadMax;
    . . .
    // Métodos de la Interface Volador
    public String despegar(){
        return("encender motores");
    }
    public String aterrizar(){
        return("Exponer ruedas");
    }
    public String volar(){
        return("Guardar ruedas");
    }
}
```

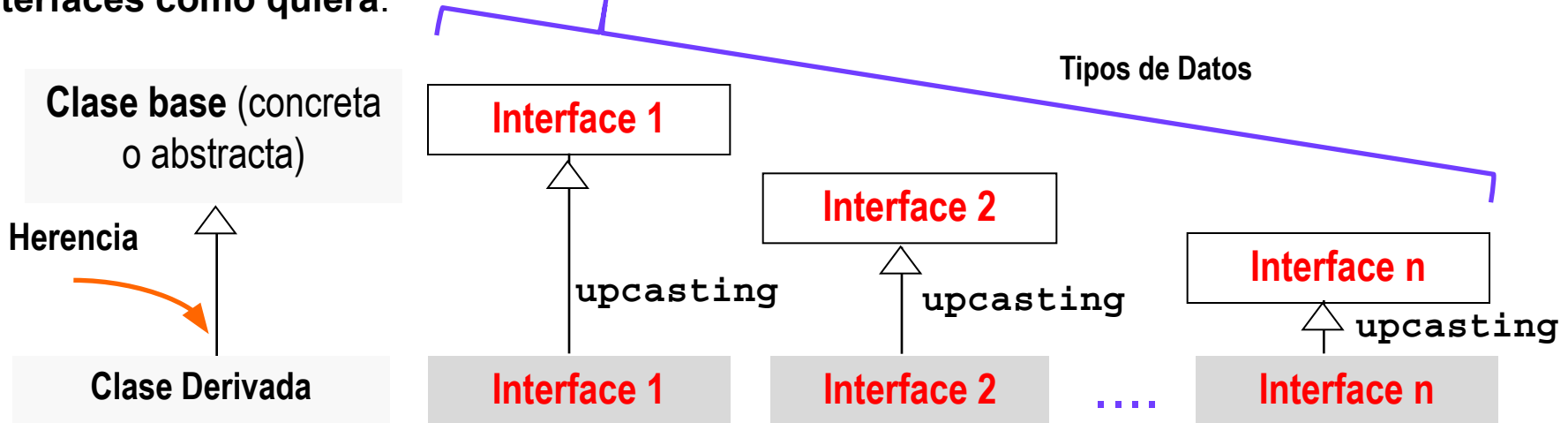
```
public interface Volador {
    long UN_SEGUNDO=1000;
    long UN_MINUTO=60000;
    public String despegar();
    public String aterrizar();
    public String volar();
}
```

# Interfaces y herencia múltiple

- El mecanismo que permite crear una clase derivada de varias clases bases, se llama **herencia múltiple**. Como cada clase tiene una implementación propia, la combinación puede generar ambigüedades.



- Java, NO soporta herencia múltiple pero provee interfaces para lograr un comportamiento “similar”. Como las interfaces no tienen implementación, **NO causa problemas combinarlas**, entonces: una clase puede heredar de **una única clase base** e **implementar tantas interfaces como quiera**.



Cada una de las interfaces que la clase implementa, provee de un **tipo de dato** al que puede hacerse **upcasting**.



# Interfaces y upcasting

```
package taller;

public class PruebaInterfaces {

    public static void partida(Volador v) {
        v.despegar(); El binding dinámico resuelve a qué
    }                               método invocar.

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

**Upcasting** castea al tipo de la interface

- Las interfaces definen un nuevo **tipo de dato** entonces, podemos definir:

**Volador[] m = new Volador[]**

- El mecanismo de **upcasting** no tiene en cuenta si **Volador** es una clase concreta, abstracta o una interface. Funciona de la misma manera.
- **Polimorfismo**: el método **despegar()** es polimórfico, se comportará de acuerdo al tipo del objeto receptor, esto es, el **despegar()** de Avion es diferente al **despegar()** de Pajaro.

El principal objetivo de las interfaces es permitir el “**upcasting**” a otros tipos, además del upcasting al tipo base. Un mecanismo similar al que provee la herencia múltiple.

# Interfaces vs. Clases Abstractas

- Las interfaces y las clases abstractas proveen una interface común. La interface Volador, podría definirse como una clase abstracta, y las clases concretas que la extiendan proveerán el comportamiento correspondiente.
- Las interfaces son completamente abstractas, no tienen ninguna implementación.
- Con interfaces no hay herencia de métodos, con clases abstractas si.
- No es posible crear instancias de clases abstractas ni de interfaces.
- Una clase puede extender sólo una clase abstracta, pero puede implementar múltiples interfaces.

## ¿Uso interfaces o clases abstractas?

- Si es posible crear una clase base con métodos sin implementación y sin variables de instancia, es preferible usar **interfaces**.
- Si estamos forzados a tener implementación o definir atributos, entonces usamos **clases abstractas**.
- **Java no soporta herencia múltiple de clases**, por lo tanto si se quiere que una clase sea además del tipo de su superclase de otro tipo diferente, entonces es necesario usar **interfaces**.

# Ordenando objetos

¿Qué pasa si definimos un arreglo con elementos de tipo `String` y los ordenamos?

```
public class Test {  
    public static void main(String[] args) {  
        String animales[] = new String[4];  
        animales[0] = "camello";  
        animales[1] = "tigre";  
        animales[2] = "mono";  
        animales[3] = "pájaro";  
        for (int i = 0; i < 4; i++) {  
            System.out.println("animal "+i+": "+animales[i]);  
        }  
        Arrays.sort(animales);  
        for (int i = 0; i < 4; i++) {  
            System.out.println("animal "+i+": "+animales[i]);  
        }  
    }  
}
```

**Arrays** es una clase del paquete **java.util**, la cual sirve para manipular arreglos, provee mecanismos de búsqueda y ordenación.

animal 0: camello  
animal 1: tigre  
animal 2: mono  
animal 3: pájaro

animal 0: camello  
animal 1: mono  
animal 2: pájaro  
animal 3: tigre

Después de invocar al método `sort()`, el arreglo quedó ordenado alfabéticamente. Esto es porque los objetos de tipo `String` son comparables.

# Ordenando objetos

## ¿Qué pasa si ordenamos objetos de tipo Persona?

```
public class Test {  
    public static void main(String[] args) {  
        Persona personas[] = new Persona[4];  
        personas[0]= new Persona("Paula","Gomez",16);  
        personas[1]= new Persona("Ana","Rios",6);  
        personas[2]= new  
Persona("Maria","Ferrer",55);  
        personas[3]= new Persona("Juana","Araoz",54);  
  
        for (int i=0; i<4;i++){  
            System.out.println(i+":personas[i]);  
        }  
        Arrays.sort(personas) ;  
        for (int i = 0; i<4; i++) {  
            System.out.println(i+": "+personas[i]);  
        }  
    }  
}
```

```
public class Persona {  
    private String nombre;  
    private String apellido;  
    private int edad;  
    public Persona  
        (String n,String a,int e){  
        nombre=n;  
        apellido=a;  
        edad=e;  
    }  
    public String toString(){  
        return apellido+", "+nombre;  
    }  
    . . .  
}
```

¿cómo ordenamos?, ¿por nombre, por apellido, por edad??. Al invocar al método **sort()**, y pasar el arreglo **personas**, da un error porque los objetos **Persona** no son comparables.

# La interface `java.lang.Comparable`

Hemos visto que cuando creamos una clase, comúnmente se sobrescribe el método **`equals(Object o)`**, para determinar si dos instancias son iguales o no. También es común, necesitar saber si una instancia es mayor o menor que otra (con respecto a alguno de sus datos) ▯ **así, poder compararlos**

## La interface `Comparable<T>`

Si una clase implementa la interface **`java.lang.Comparable`**, hace a sus instancias comparables. Esta interface tiene sólo un método, **`compareTo()`**, el cual determina cómo comparar dos instancias de una misma clase. El método es el siguiente:

```
public interface Comparable<T> {  
    public int compareTo(T o);  
}
```

La intención es que cada clase que la implemente reciba un T del tipo de la clase

Este método retorna:

- `=0`: si el objeto receptor es igual al pasado en el argumento.
- `>0`: si el objeto receptor es mayor que el pasado como parámetro.
- `<0`: si el objeto receptor es menor que el pasado como parámetro.

# La interface `java.lang.Comparable`

La clase `Persona` implementa la interface **`Comparable<T>`**

```
public class Test {  
    public static void main(String[] args){  
        Persona personas[] = new Persona[3];  
        personas[0]= new Persona("Paula","Gomez",16);  
        personas[1]= new Persona("Ana","Rios",6);  
        personas[2]= new Persona("Maria","Ferrer",55);  
        personas[3]= new Persona("Juana","Araoz",54);  
        for (int i=0; i<4;i++){  
            System.out.println(i+": "+personas[i]);  
        }  
        Arrays.sort(personas);  
        for (int i = 0; i<4; i++) {  
            System.out.println(i+": "+personas[i]);  
        }  
    }  
}
```

Al invocar al método **`sort()`**, ahora si los puede ordenar,  
con el criterio establecido en el **`compareTo()`**

```
0:Gomez, Paula:16  
1:Rios, Ana:6  
2:Ferrer, Maria:55  
3:Araoz, Juana:54
```

```
0:Rios, Ana:6  
1:Gomez, Paula:16  
2:Araoz, Juana:54  
3:Ferrer, Maria:55
```

```
import java.util.*;  
public class Persona  
    implements Comparable<Persona> {  
    private String nombre;  
    private String apellido;  
    private int edad;  
    public Persona(String n,String a,  
                                                             int e){  
        nombre=n;  
        apellido=a;  
        edad=e;  
    }  
    public String toString(){  
        return apellido+", "+nombre;  
    }  
    public int compareTo(Persona o){  
        return this.edad - o.edad;  
    }  
}
```

# La interface `java.lang.Comparable`

La clase `Persona` implementa la interface **`Comparable<Persona>`**

```
import java.util.*;

public class Persona implements Comparable<Persona> {
    private String nombre;
    private String apellido;
    private int edad;
    public Persona(String n, String a, int e) {
        nombre=n;
        apellido=a;
        edad=e;
    }

    . . .

    public String toString(){
        return apellido+" "+nombre;
    }

    public int compareTo(Persona o){
        return this.edad - o.edad;
    }
}
```

```
public class Test {
    public static void main(String[] args) {
        public static void main(String[] args) {
            Persona per1 = new Persona("Paula","Gomez",16);
            Persona per2 = new Persona("María","Shan",10);
            if (per1.compareTo(per2)>0)
                System.out.println(per1.getNombre()+
                    " es mayor que: "+ per2.getNombre());
        }
    }
}
```

# Árboles Binarios de Búsqueda

## Inserción de un dato (Weiss)

```
package ayed.arboles;

public class ArbolBinarioDeBusqueda<T extends Comparable<T>> {
    private T dato;
    private ArbolBinarioDeBusqueda<T> hijoIzquierdo;
    private ArbolBinarioDeBusqueda<T> hijoDerecho;
    . . .
    public void agregar(T x) {
        if (dato==null)
            dato = x;
        else
            this.agregar(x, this);
    }
    private void agregar(T x, ArbolBinarioDeBusqueda<T> t) {
        if (x.compareTo(t.getDato())<0)
            if (!t.tieneHijoIzquierdo())
                t.setHijoIzquierdo(new ArbolBinarioDeBusqueda<T>(x));
            else
                this.agregar(x, t.getHijoIzquierdo());
        else if (x.compareTo(t.getDato())>0)
            if (!t.tieneHijoDerecho())
                t.setHijoDerecho(new ArbolBinarioDeBusqueda<T>(x));
            else
                this.agregar(x, t.getHijoDerecho());
    }
}
```

```
ArbolBinarioDeBusqueda<Integer> abb =
    new ArbolBinarioDeBusqueda<Integer>();
abb.insertar(2);
abb.insertar(6);
abb.insertar(1);
abb.insertar(13);
abb.insertar(5);
abb.insertar(7);
abb.recorridoporNiveles();
```

