

# Taller de Lenguajes II

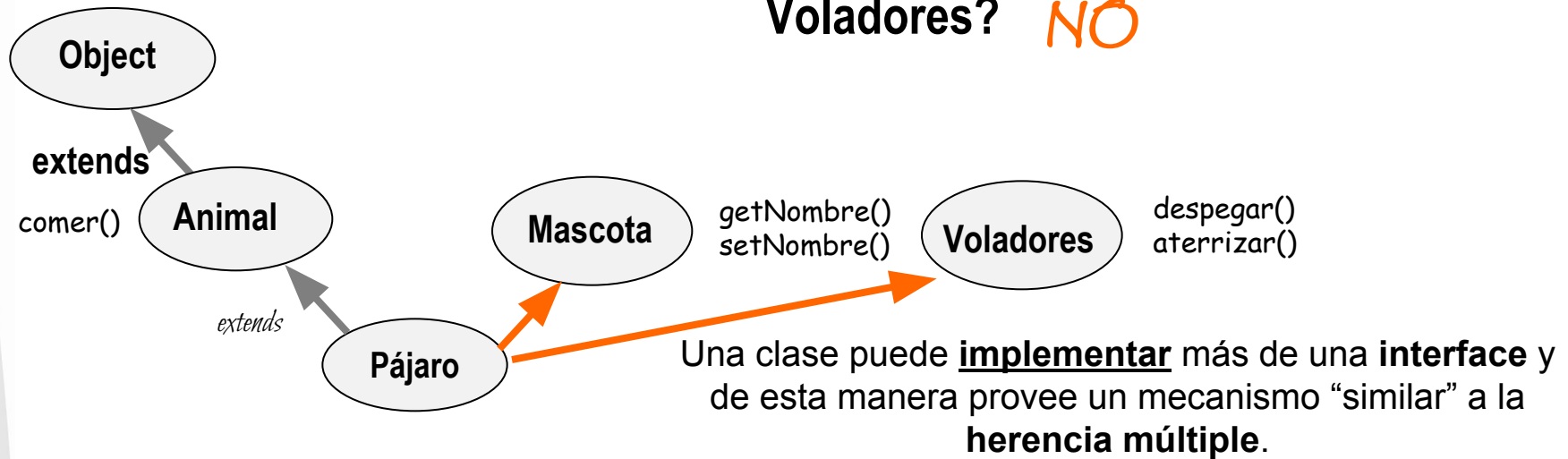
## Temas de hoy: Interfaces

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

# 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  
}
```



lista de nombres de  
interfaces

- 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();
}
```

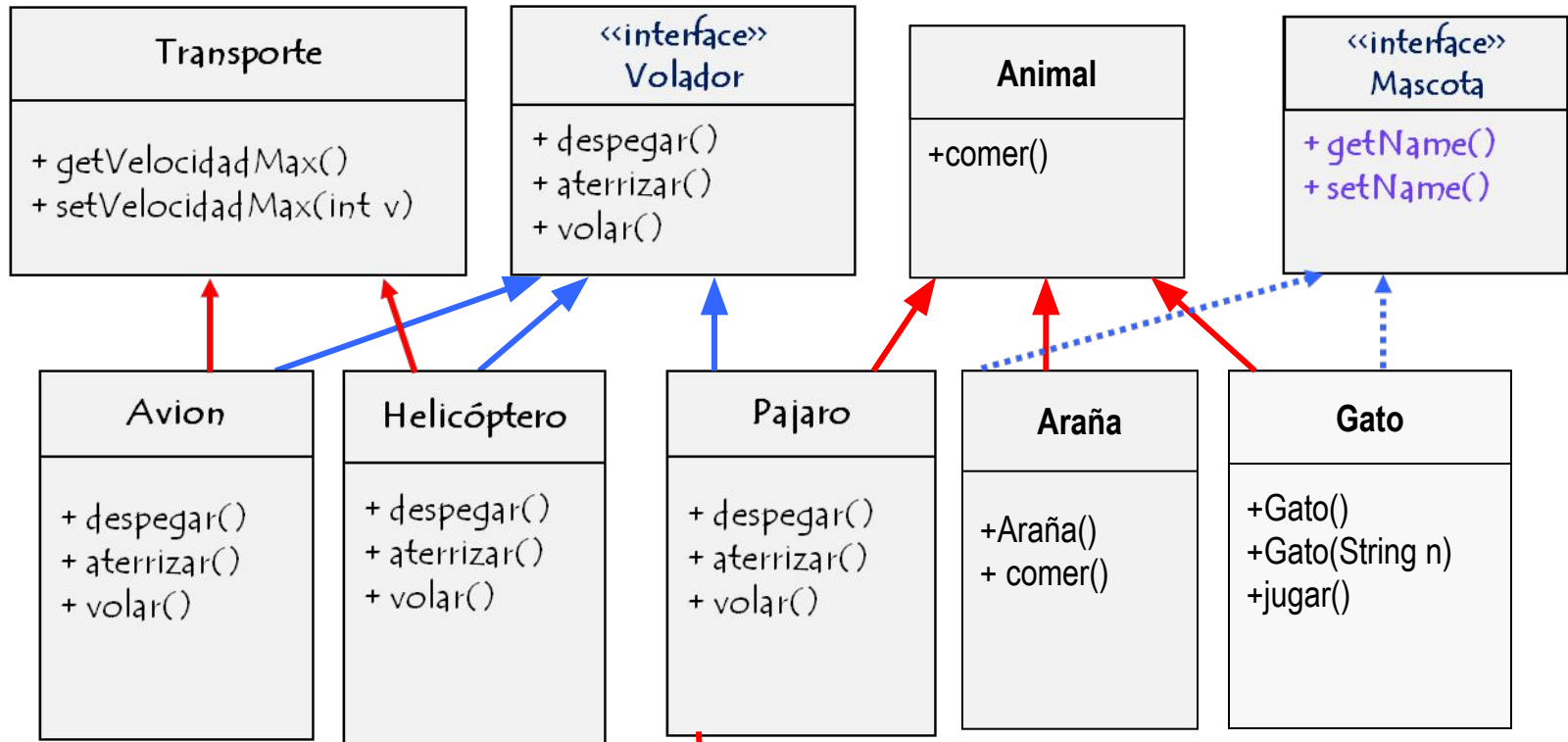
- Una clase que implementa una interface, hereda las constantes y **debe implementar cada uno de los métodos declarados en la interface.**
- Una clase puede implementar más de una interface y de esta manera provee un mecanismo similar a la herencia múltiple.

```
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 despegar, 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{

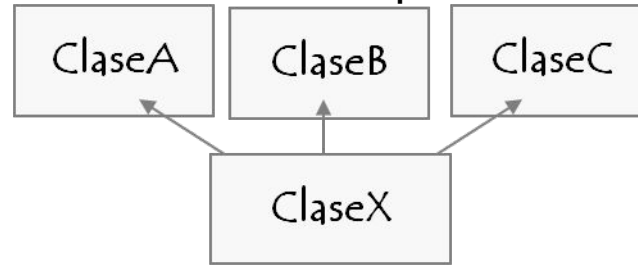
    int velocidadMax;

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

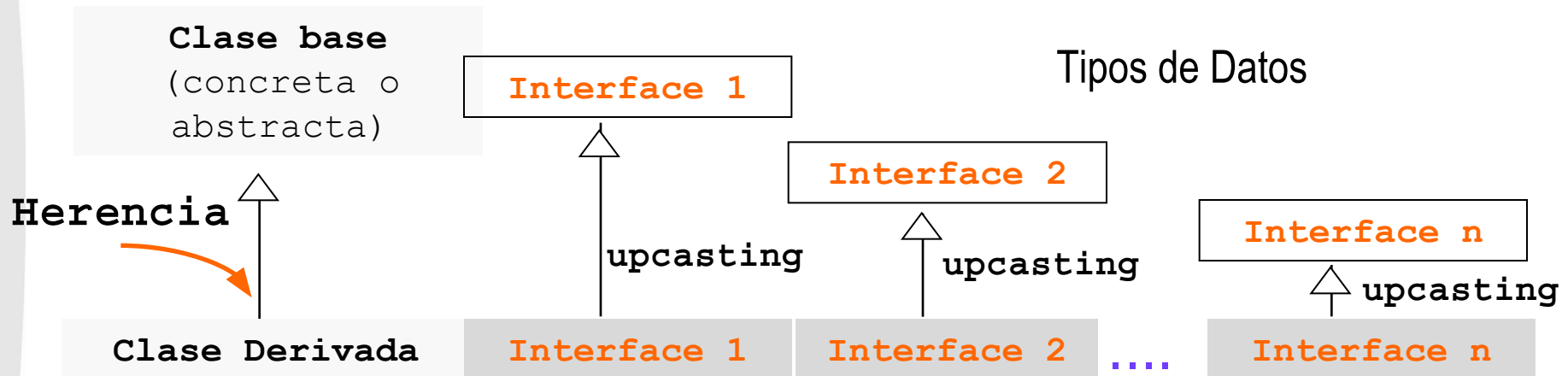
public interface Volador {
    long UN_SEGUNDO=1000;
    long UN_MINUTO=60000;
    public String despegar();
    public String aterrizar();
    public String volar();
}
```

# 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 traer problemas!!.



- 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();
    }

    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]);
    }
}
```

El **binding dinámico** resuelve a que método invocar. En este caso, más de una clase implementó la misma interface y en consecuencia, el método **despegar()** correspondiente será invocado.

**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.

**Nota 1:** 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.

# Colisión de nombres en interfaces

Cuando se implementan múltiples interfaces, pueden aparecer conflictos: ¿qué pasa si más de una interface define el mismo nombre de método?

```
public interface Alfa {  
    public int f();  
}
```

```
public interface Beta {  
    public void f();  
}
```

```
public class Alfabeta implements Alfa, Beta{  
    public void f() {.. }  
    public int f() { .. }  
}
```

**Colisión!!!**. Las interfaces Alfa y Beta son incompatible, ambas definen el método `f()` pero con distinto tipo de retorno.

**Nota:** Especificador de acceso

```
public interface Alfa {  
    int f(); //es public  
}
```

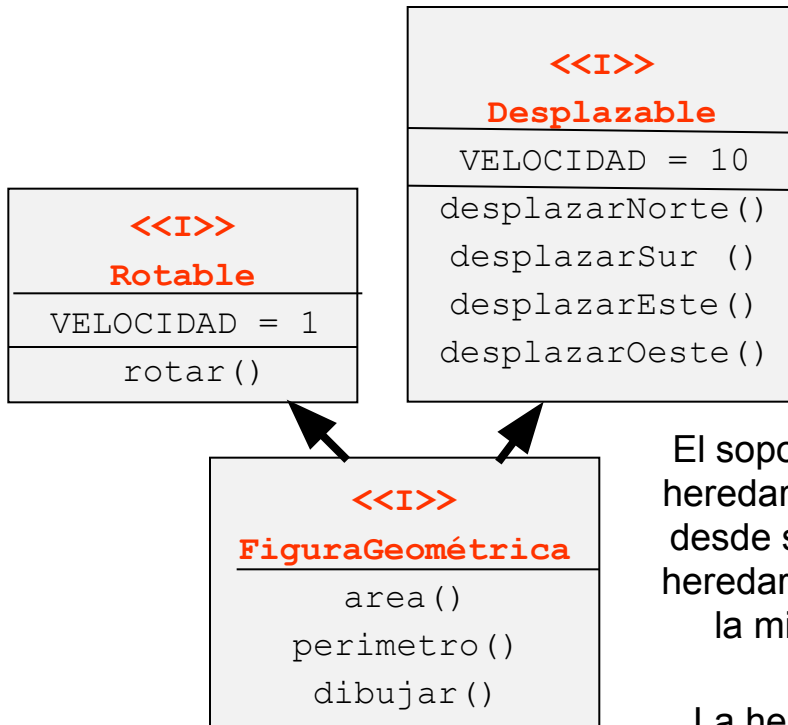
```
public class AlfaBeta implements Alfa {  
    int f() { .. }  
}
```

**Error!!**. Las clases que implementan una interface no pueden poner a los métodos implementados, un especificador de acceso más restrictivo que el que tiene el método en la interface: que siempre es

**public**

# Herencia múltiple de interfaces

JAVA provee herencia múltiple de interfaces. A diferencia de las clases, una interface sí puede extender varias interfaces. En el diagrama se observa que la interface **FiguraGeometrica** extiende a las interfaces **Rotable** y **Desplazable**.



La interface **FiguraGeometrica** hereda de sus superinterfaces los métodos `rotar()`, `desplazarNorte()`, `desplazarSur()`, `desplazarEste()`, `desplazarOeste()` y la constante `VELOCIDAD`. Asimismo la interface **FiguraGeometrica** define sus propios métodos.

El soporte de herencia múltiple de interfaces en JAVA consiste en heredar definiciones de métodos abstractos y constantes de clase, desde sus *superinterfaces*. Este mecanismo no es ambiguo, al no heredar comportamiento –código- es posible que dos métodos con la misma firma se hereden de diferentes *superinterfaces*, sin provocar conflicto.

La herencia múltiple en clases puede provocar que se hereden métodos con la misma firma y diferente comportamiento, siendo necesario definir reglas que establezcan de qué superclase heredar (JAVA no lo soporta). En el caso de las interfaces como no hay implementación no es necesario establecer de dónde se hereda.

# Heredando mismos nombres de constantes

¿Qué sucede en la Interface **FiguraGeometrica** con la constante **VELOCIDAD** heredada desde sus superinterfaces? Si en **FiguraGeometrica** no se hace referencia a **VELOCIDAD** no hay conflicto.

```
package graficos.interfaces;

public interface Desplazable {
    public static final double VELOCIDAD=10;
    void desplazarNorte();
    void desplazarSur();
    void desplazarEste();
    void desplazarOeste();
}
```

```
package graficos.interfaces;

public interface Rotable {
    public static final double VELOCIDAD = 1;
    public void rotar(int grados);
}
```

```
package capitulo5.interfaces;

public interface FiguraGeometrica extends Desplazable, Rotable {
    // public static final double PROPORCION = VELOCIDAD*0.6;
    public static final double PROPORCION = Rotable.VELOCIDAD*0.6;
    void dibujar();
    double area();
    double perimetro();
}
```

Si en **FiguraGeometrica** se hace referencia a **VELOCIDAD**, se debe quitar la ambigüedad de nombres anteponiendo a la constante el nombre de la interface.

No se puede determinar qué velocidad queremos usar

Lo mismo sucedería al referenciar **VELOCIDAD** desde una clase que implemente las interfaces **Desplazable** y **Rotable**. Se necesita identificar qué interface se quiere usar.

# Interfaces vs. Clases Abstractas

JAVA provee dos mecanismos para definir tipos de datos que admiten múltiples implementaciones: clases abstractas e interfaces.

- Las interfaces y las clases abstractas proveen una interface común. La interface Volador, también podría definirse como una clase abstracta, con tres métodos abstractos: despegar(), aterrizar() y volar(). Las clases concretas que la implementen o extiendan (interface/clase abstracta) 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 o concreta), 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 el 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]);  
        }  
    }  
}
```

**Error en ejecución!!**

```
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??. El método **sort()** recibe un arreglo de Personas que 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

## Una solución es implementar 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

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

¿qué pasa si queremos ahora ordenar por apellido?

# La interface `java.lang.Comparator`

## Otra solución: implementar la interface `java.util.Comparator`

Implementando la interface `java.util.Comparator`, también define una manera de comparar instancias de una clase. Sin embargo, este mecanismo, permite comparar instancias por distintos criterios.

Por ejemplo: podríamos comparar a dos objetos personas por edad, por apellido o por nombre. En estos casos, se debe crear un *comparator* que defina como comparar dos objetos de tipo `Persona`.

Para crear un *comparator*, se debe escribir una clase (con cualquier nombre) que implemente la interface `java.util.Comparator` e implementar la lógica de comparación en el método `compare(..)`. Este método tiene el siguiente encabezado:

```
public interface Comparator{  
    public int compare(T o1, T o2)  
}
```

La intención de esta interface es que la clase que la implementa reciba T del tipo de esa clase.

### El método retorna:

`=0`: si los objetos `o1` y `o2` son iguales.  
`<0`: si `o1` es menor que `o2`.  
`>0`: si `o1` es mayor que `o2`.

# La interface `java.lang.Comparator`

Implementemos 2 clases comparadoras para la clase **Persona**, una que las compara por edad y la otra por nombre.

```
package ayed2010;
import java.util.Comparator;
public class ComparadorNombre implements Comparator<Persona> {
    public int compare(Persona p1, Persona p2) {
        if (!(p1.getApellido().equals(p2.getApellido())))
            return p1.getApellido().compareTo(p2.getApellido());
        else
            return p1.getNombre().compareTo(p2.getNombre());
    }
}
```

Clases creadas  
especialmente para  
ordenar objetos de tipo  
Persona

```
package ayed2010;
import java.util.Comparator;
public class ComparadorEdad implements Comparator<Persona> {
    public int compare(Persona p1, Persona p2) {
        return p1.getEdad()-p2.getEdad();
    }
}
```

# La interface `java.lang.Comparator`

Ahora podemos ordenar a los objetos de tipo `Persona`, por distintos criterios. Al invocar al método `sort()`, *debemos indicar con que criterio ordenar*, es decir, que clase *comparator* usar.

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

## Ordenador por edad

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

## Ordenador por nombre

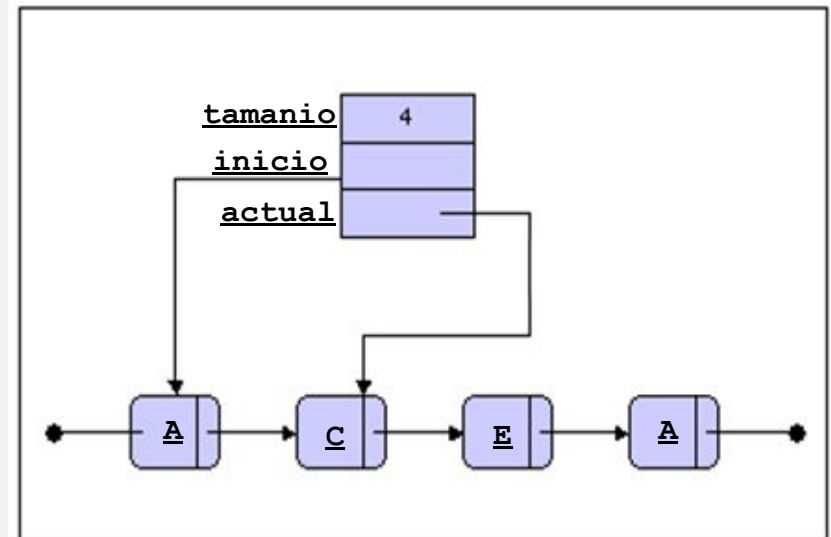
```
persona 0:Ana, Rios:6  
persona 1:Maria, Araoz:54  
persona 2:Maria, Ferrer:55  
persona 3:Paula, Gomez:16
```

El método `sort(Object[] datos, Comparator c)` de `Arrays`, ordenará al arreglo **datos** con el criterio implementado en el método `compare(Object o1, Object o2)`, en la clase `Comparator`.

# Estructuras e iteradores

Pensemos ¿cómo verificar si una lista contiene elementos repetidos?

```
public class ListaEnlazadaGenerica<T> extends ListaGenerica<T> {  
    // int tamaño = 0; heredado de ListaGenerica<T>  
    private NodoGenerico<T> inicio;  
    private NodoGenerico<T> actual;  
  
    @Override  
    public void comenzar() {  
        actual = inicio;  
    }  
  
    @Override  
    public void proximo() {  
        actual = actual.getSiguiente();  
    }  
  
    @Override  
    public T elemento(int i) {  
        . . .  
    }  
    . . .  
}
```



Java provee un mecanismo para recorrer las estructuras de manera homogénea a través del uso de iteradores. Asimismo esta solución permite tener más de un iterador simultáneamente.

# La interface `Iterator<E>`

Existe una interface llamada `Iterator` que define un protocolo genérico para iterar

```
package java.util;
```

```
public interface Iterator<E> {
```

```
    boolean hasNext();
```

```
    E next();
```

```
    void remove();
```

```
}
```

Retorna `true` si hay más elementos y `false` en caso contrario

Retorna el próximo elemento. Dispara una excepción si no hay próximo elemento

Elimina el elemento actual

Los iteradores permiten recorrer una colección de datos usando una interface estándar sin tener que conocer la representación interna de los datos que se recorren.

También se puede definir un iterador especial que ejecute algún procesamiento y retorne solamente determinados elementos de la colección.

# Las interfaces `Iterable<T>` e `Iterator<E>`

¿Cómo hacemos para hacer que la clase `ListaGenericaEnlazada` devuelva iteradores?

```
public class ListaEnlazadaGenericaIterable<T> extends ListaGenerica<T> {  
    private NodoGenerico<T> inicio;  
    ...  
    public java.util.Iterator<T> iterator() {  
        return new MiIterador();  
    }
```

`MiIterador` por ser una **clase interna** a `ListaEnlazadaGenericaIterable` puede acceder a las variables e invocar los métodos de la clase que la contiene

```
private class MiIterador implements java.util.Iterator<T> {  
    private NodoGenerico<T> actual = null;  
    MiIterador() {  
        actual = getInicio();  
    }
```

```
    public boolean hasNext() {  
        return actual != null;  
    }
```

El método **hasNext()**, retorna si hay o no más elementos.

```
    public T next() {  
        T dato = actual.getDato();  
        actual = actual.getSiguiete();  
        return dato;  
    }
```

El método **next()**, retorna el próximo ítem en la colección. La primer invocación al mismo dará el 1º elemento, la segunda invocación el 2º elemento y así sucesivamente.

```
    public void remove() {}
```

Con este método, se puede eliminar de la colección, el último elemento retornado por **next()**

5 8 15 10 7

next retorna 5

hasNext retorna true

```
}
```

# Las interfaces `Iterable` e `Iterator`

Recordemos que un arreglo puede recorrerse usando la sentencia “*foreach*” de la siguiente forma:

```
Integer[] pares = {2, 4, 6, 8, 10};  
for (Integer i: pares) {  
    System.out.println(i);  
}
```

¿Sería posible recorrer nuestra `ListaEnlazadaGenerica` de la misma manera?

El “*foreach*” requiere que las estructuras a iterar sean “*iterables*”. Para hacer que una estructura de datos sea iterable, ésta debe implementar la interface `java.lang.Iterable<T>`

```
package java.lang;  
import java.util.Iterator;  
  
public interface Iterable<T> {  
    Iterator<T> iterator();  
}
```



# Las interfaces `Iterable` e `Iterator`

La interface `Iterable` tiene un único método llamado: `iterator()` el cual retorna un iterador (objeto de tipo `Iterator`).

Las clases que implementan la interface `Iterable`, permiten que un objeto de su tipo, sea iterado usando la sentencia “foreach” o con un estructura de control tradicional.

```
package java.lang;
import java.util.Iterator;
public interface Iterable<T> {
    Iterator<T> iterator();
}
```

```
package java.util;
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();
}
```

```
public class ListaEnlazadaGenericaIterable<T> implements Iterable<T>{
    . . .
    Iterator<T> iterator() {
        return new MiIterator();
    }
}
```

Es una clase que implementa `java.util.Iterator`. Retorna un iterador para un conjunto de elementos de tipo `T`.

La idea de los iteradores es que vía el método `iterator()`, cada colección pueda crear y retornar un objeto que implemente la interface `Iterator` que permita recorrerla.

# Las interfaces Iterable e Iterator

¿Cómo hacemos para hacer a la clase ListaGenericaEnlazada sea iterable?

```
public class ListaEnlazadaGenericaIterable<T> implements Iterable<T> {
    private NodoGenerico<T> inicio;
    . . .
    public java.util.Iterator<T> iterator() {
        return new MiIterador();
    }

    private class MiIterador implements java.util.Iterator<T> {
        private NodoGenerico<T> actual = null;

        MiIterador() {
            actual = ListaEnlazadaGenericaIterable.this.getInicio();
        }

        public boolean hasNext() {
            return actual != null;
        }

        public T next() {
            T dato = actual.getDato();
            actual = actual.getSiguiente();
            return dato;
        }

        public void remove() {}
    }
}
```

La implementación no cambió, al implementar la interface Iterable se debe dar comportamiento al método iterator()

# Las interfaces Iterable e Iterator

¿Cómo iteramos un objeto de tipo `ListaEnlazadaGenericaIterable`?

```
import java.util.*;

public class TestListaIterable {

    public static void main(String[] args) {

        ListaEnlazadaGenericaIterable<Integer> lista=new
                                                    ListaEnlazadaGenericaIterable<Integer>();

        lista.agregar(10);
        lista.agregar(1);
        lista.agregar(12);

        Iterator<Integer> it1 = lista.iterator();
        while (it1.hasNext()){
            System.out.print(it1.next().intValue());
        }

        for (Integer i:lst)
            System.out.print(i);

        Iterator it2 = lst.iterator();
        while (it2.hasNext()){
            System.out.print(it2.next().intValue());
        }
    }
}
```

es un Integer

es un Object

Cuando el compilador ve un for each, usado con un objeto Iterable, automáticamente llama al `iterator()` y arma un loop usando los métodos `hasNext()` y `next()`.

**ERROR al llamar al `intValue()` sobre un Object!!!**

# Interfaces en Java 8

# Interfaces en JAVA 8

## ¿Cómo se define una interface en Java 8?

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

La nueva especificación de Interfaces permite la implementación de métodos usando la cláusula “*default*”. Estos métodos serían heredados por las clases que implementan la interface. Los métodos default son necesarios para la evolución de las interfaces y en consecuencia, para mantener compatibilidad con el código escrito.

Hacer reingeniería de un framework (como por ejemplo el de colecciones), involucraría seguramente la modificación de interfaces, y esto rompería el código de todas las clases que las implementan. Por lo tanto, los implementadores de JDK tuvieron que imaginar formas de extender interfaces de una manera compatible con versiones anteriores e inventaron métodos predeterminados.

# Interfaces en JAVA 8

Si modificamos la interface Volador, poniendo un comportamiento por defecto en el método `volar()` nos quedaría así:

Los métodos default pueden ser agregados a una interface sin romper el código de las clases que la implementan porque los métodos default tienen implementación

```
package clase;

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

- Una clase que implementa una interface, hereda las constantes y los métodos default y deberá **implementar cada uno de los restantes métodos abstractos de la interface.**

```
package clase;

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

```
package clase;

public interface Mascota{
    String getName();
    String setName(String s);
}
```

# Interfaces en JAVA 8

## Ambigüedad

*¿Qué pasa si una clase implementa dos interfaces que tienen el mismo método abstracto, con la misma firma?*

```
public interface InterfaceA {  
    abstract void metodoA();  
    abstract void metodoX();  
}
```

```
public interface InterfaceB {  
    abstract String metodoB(int x);  
    abstract void metodoX();  
}
```

```
public class MiClase implements InterfaceA, InterfaceB {  
    public void metodoA() { . . . }  
    public String metodoB(int x) { . . . }  
    public void metodoX() {  
        // comportamiento  
    }  
}
```

La clase está obligada a implementar cada uno de los métodos abstractos de sus interfaces -> lo implementa una vez y cumple con ambos contratos.

# Interfaces en JAVA 8

## Ambigüedad


*¿Qué pasa si una clase implementa dos interfaces que tienen exactamente el mismo método declarado como default?*

Si ambas interfaces definen un método *default* con la misma firma, se produce una ambigüedad. La clase debe implementar (sobrescribir) el método con algún comportamiento para quitar esa ambigüedad.

```
public interface InterfaceA {  
    abstract void metodoA();  
    default void metodoX(){ . . . }  
}
```

```
public interface InterfaceB {  
    abstract String metodoB(int x);  
    default void metodoX(){ . . . }  
}
```

```
public class MiClase implements InterfaceA, interfaceB {  
    public void metodoA() { . . . }  
    public String metodoB(int x) { . . . }  
    public void metodoX() {  
        InterfaceA.super.metodoX();  
    }  
}
```



La implementación del método en la clase puede hacer referencia a algunas de las dos implementaciones de las interfaces o definir un comportamiento nuevo.