

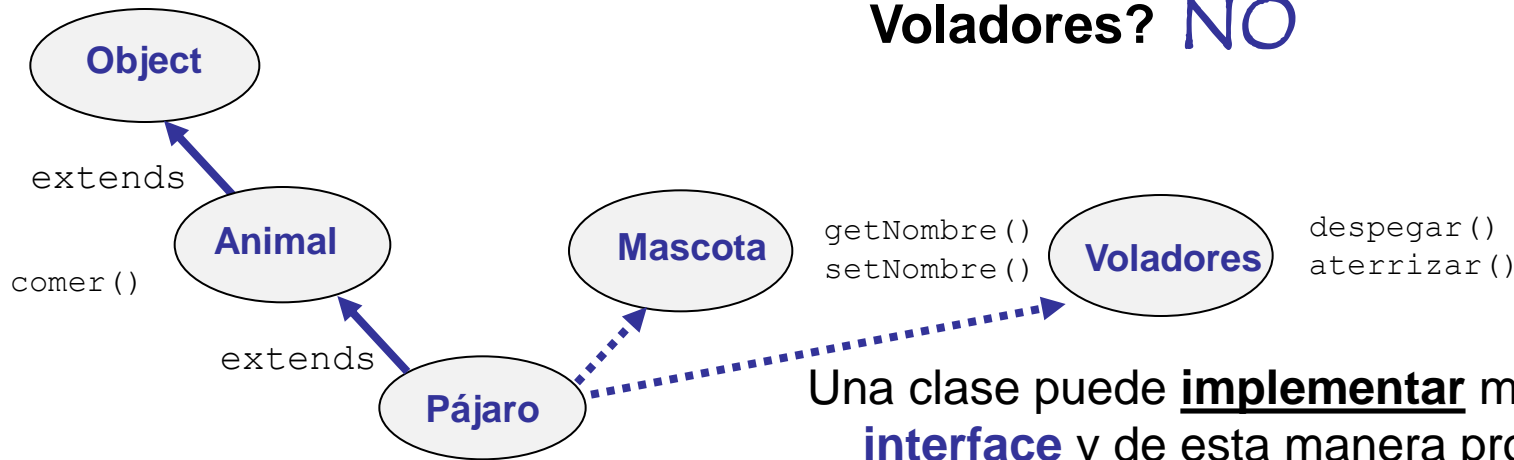
Interfaces

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

Interfaces

¿Qué son?, ¿Para qué sirven?

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




Una clase puede **implementar** más de una **interface** y de esta manera provee un mecanismo “similar” a la **herencia múltiple**.

- Una interface java es una colección de definiciones de métodos **abstractos**, métodos **default** 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** (excepto por los métodos default).

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 métodos default: implícitamente public  
    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, para el caso de los métodos abstractos; para los métodos con comportamiento, se debe explicitar la palabra clave **default**.

```
package clase;

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 default void volar(){
        // algún comportamiento
    }
}
```


```
package clase;

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

- Esta interface **Volador** establece en sus métodos abstractos qué debe hacer la clase que la implementa, sin especificar el cómo. Puede existir algún método **default**, como es el caso del **volar()**.
- Las clases que implementen **Volador** deberán implementar los métodos **despegar()** y **aterrizar()**, 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**. El método **volar** no necesita implementación.
- 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**



```
package clase;
public class Pajaro
    implements Volador {
    . . .
}
```

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

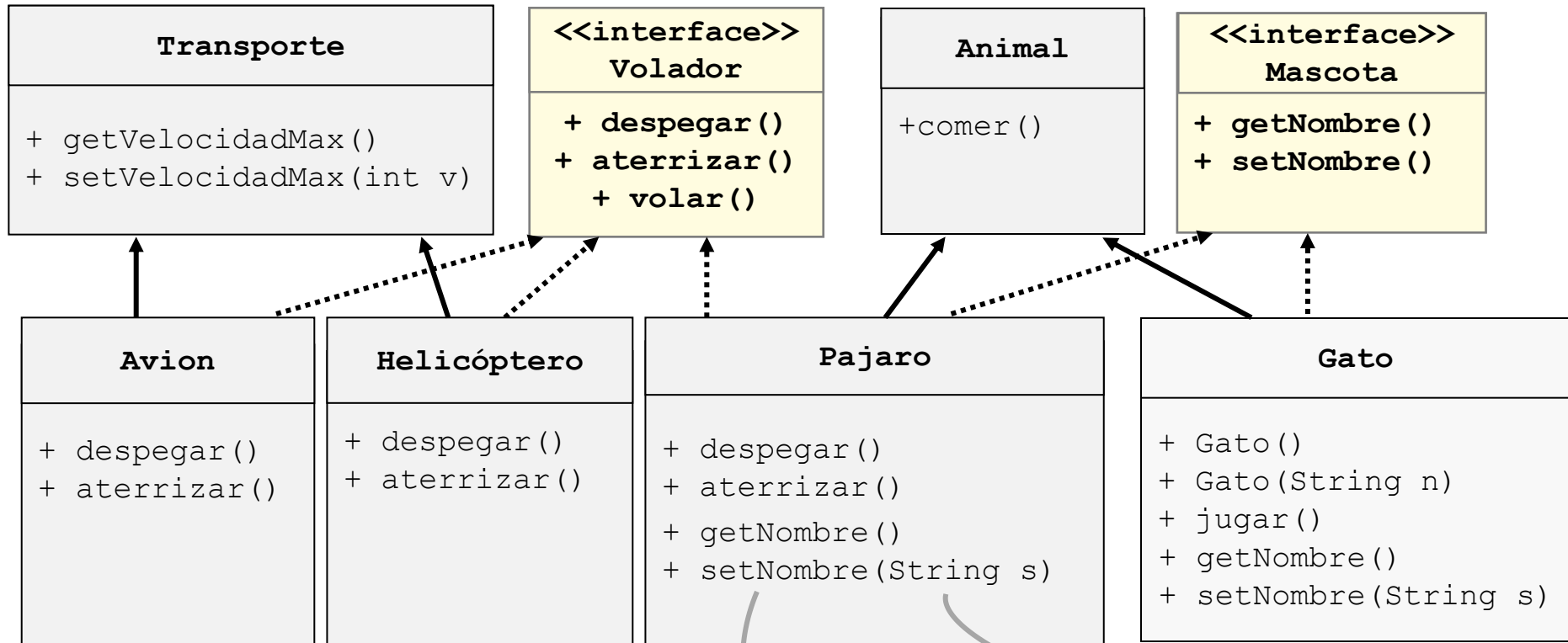
- Una clase que implementa una interface, hereda las constantes y **debe implementar cada uno de los métodos abstractos de la interface !!!**.
- Una clase puede implementar más de una interface y de esta manera provee un mecanismo similar a la herencia múltiple.

```
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);
}
```

Implementación de Interfaces

Considerando este ejemplo de una interface que describe **cosas que vuelan**, podríamos tener múltiples implementaciones de la misma:



Cada tipo de objeto despegue y aterriza de manera diferente, por lo tanto necesita **implementar un comportamiento diferente** para acciones similares.

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

Una clase también puede implementar más de una interface.

Implementación de Interfaces

Cuando una clase implementa una interface se *establece un contrato* entre la interface y la clase que la implementa. El compilador chequea que la clase implemente todos los métodos abstractos de la interface (y si es necesario algún método default para evitar ambigüedad).

```
public class Pajaro extends Animal
    implements Mascota, Volador {
    private String nombre;

    // Métodos de la Interface Mascota
    public void setNombre(String nombre){
        this.nombre = nombre;
    }
    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 interface Volador {
    long UN_SEGUNDO=1000;
    long UN_MINUTO=60000;
    String despegar();
    String aterrizar();
    default void volar(){ . . . }
}
```

```
public class Avion extends Transporte
    implements Volador{
    int velocidadMax;
    . . .
    // Métodos de la Interface Volador
    public String despegar(){
        return("Elevar y guardar ruedas");
    }
    public String aterrizar(){
        return("Desplegar ruedas");
    }
}
```

El método `volar()` de la interface `Volador` tiene un comportamiento por defecto. Las clases podrían sobrescribirlo pero no es necesario

Implementación de Interfaces

¿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.

Implementación de Interfaces

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(){ . . . }  
}
```

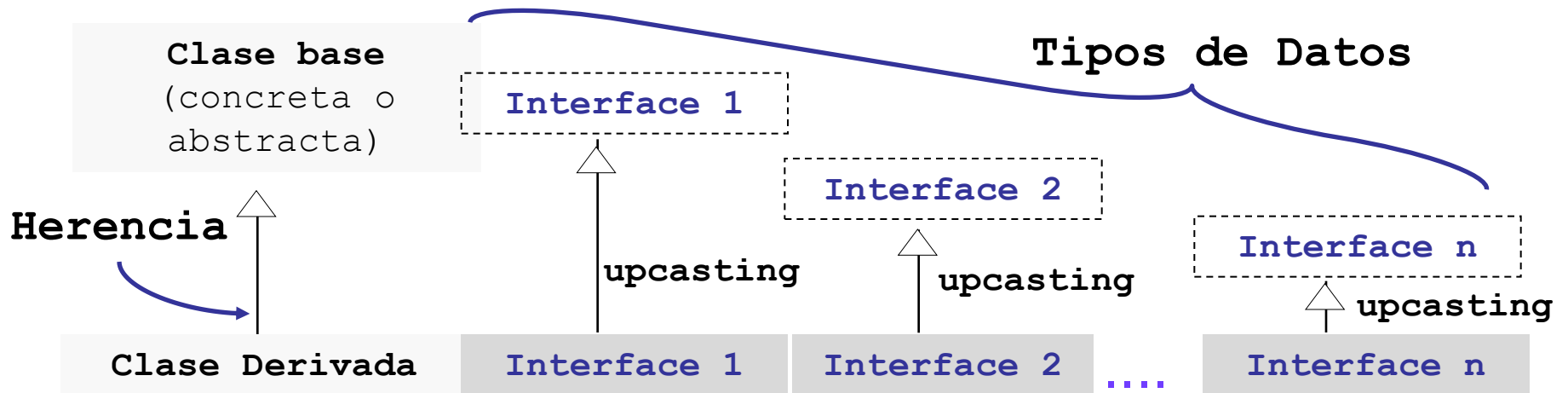
```
public class MiClase implements InterfaceA, interfaceB {  
    public void metodoA() { . . . }  
    public String metodoB(int x) { . . . }  
    public void metodoX() {  
        InterfaceA.super.metodoX();  
    }  
}
```



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

Interfaces y herencia múltiple

- El mecanismo que permite crear una clase derivada a partir de varias clases bases, se llama **herencia múltiple**.
- Java, NO soporta herencia múltiple pero provee interfaces para lograr un comportamiento “similar”. Como las interfaces no tienen estado (no tienen variables de instancia), NO causa problemas combinarlas.



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

Una clase puede heredar de una única clase base e implementar tantas interfaces como quiera.


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++)
            PruebaInterfaces.partida(m[j]);
    }
}
```



Polimorfismo

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

El método despegar() es polimórfico, se comportará de acuerdo al tipo real del objeto receptor.

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

```
Volador[] m = new Volador[3];
```

El mecanismo de upcasting no tiene en cuenta si **Volador** es una clase concreta, abstracta o una interface. Funciona de la misma manera.

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

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.
- 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 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?

```
import java.util.Arrays;

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?

```
import java.util.Arrays;

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); Error en ejecución!!
        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**

1º solución: 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 como 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

```
import java.util.Arrays;

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]);
        }
    }
}
```

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.getEdad;
    }

}
```

¿qué pasa si queremos ahora ordenar por apellido?

La interface `java.util.Comparator`


2º 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** 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:

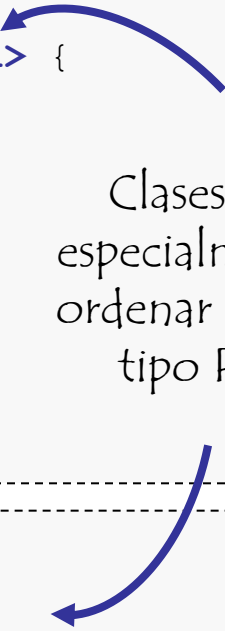
- =0:** si los objetos o1 y o2 son iguales.
- <0:** si o1 es menor que o2.
- >0:** si o1 es mayor que o2.

La interface `java.util.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.util.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`.