Interfaces

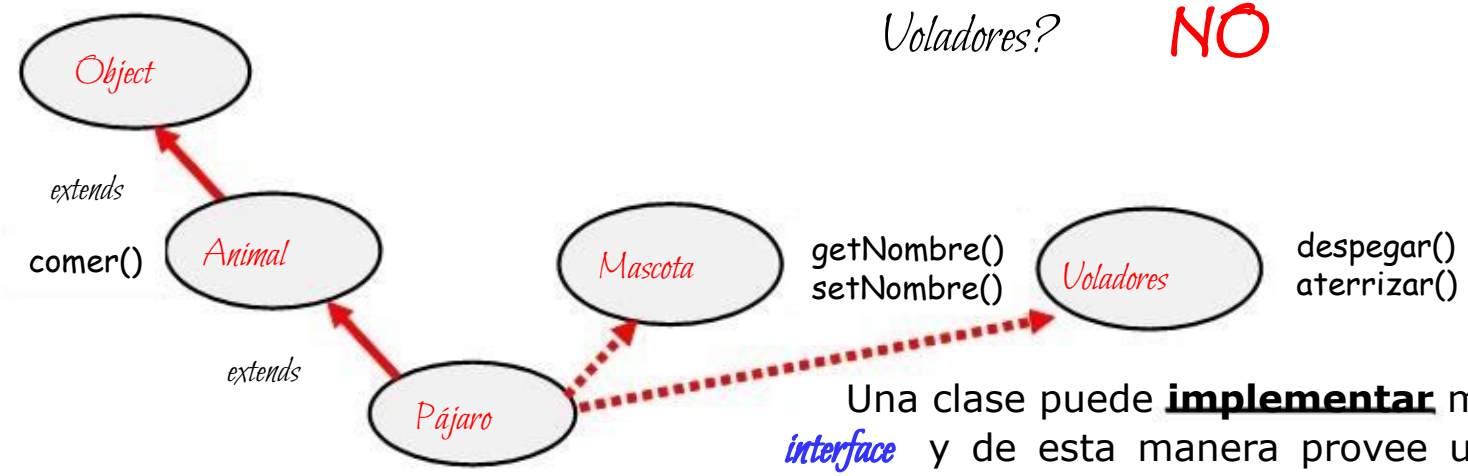
- Interfaces
  - ¿Qué son las interfaces?
  - ¿Para que sirven?
- Declaración de interfaces en java
- Un ejemplo:
  - Declarando interfaces
  - Implementando múltiples interfaces - Upcasting
- Las interfaces **`Comparable`** y **`Comparator`**

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 sin implementación/ cuerpo y de 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?

lista de nombres de interfaces

```
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();
}
```

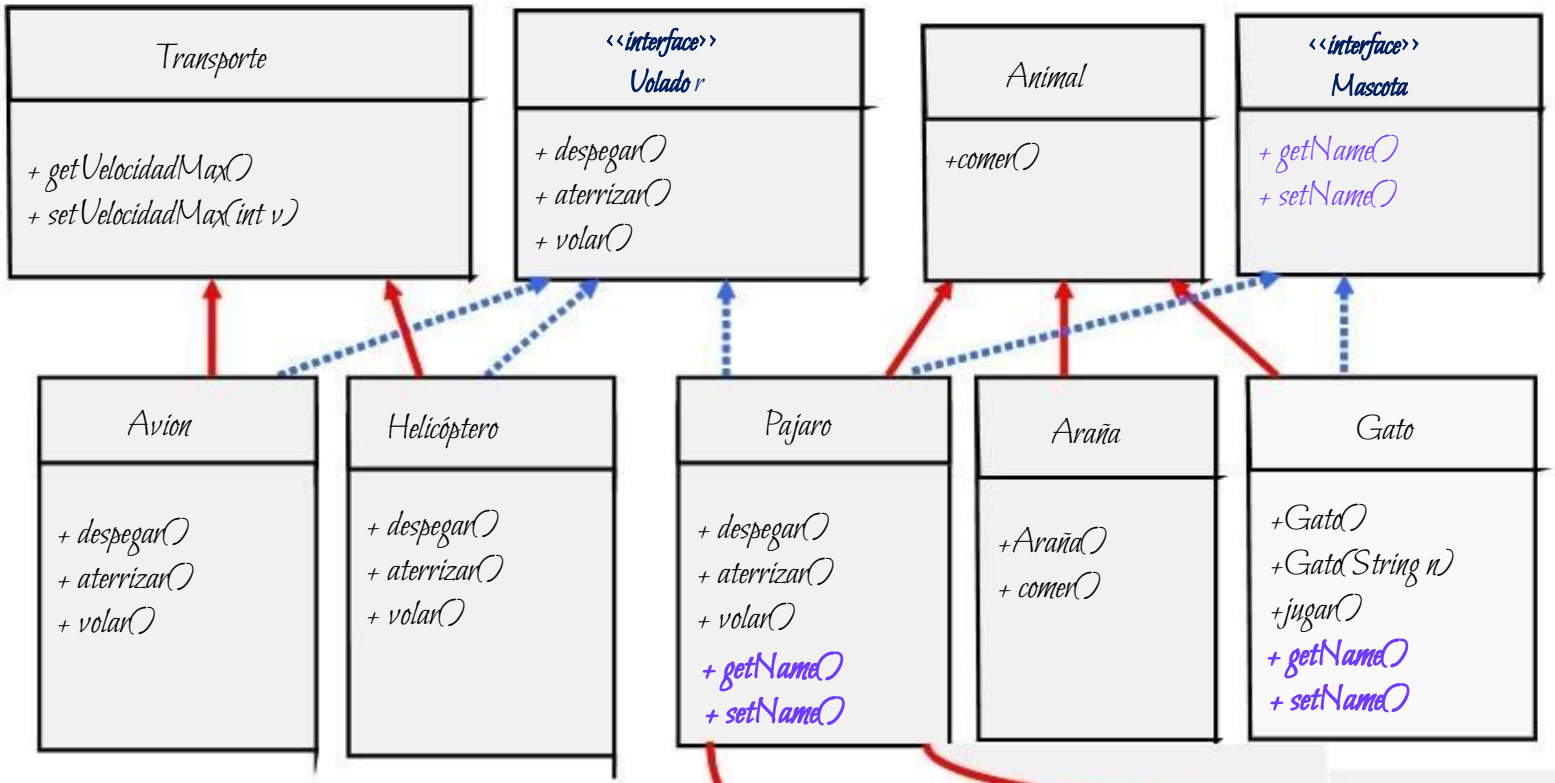
- Una clase que implementa una interface, hereda las constantes y **debe implementar cada uno de los métodos declarados en la interface !!!**.
- Una clase puede implementar más de una interface y de esta manera provee un mecanismo similar a la herencia múltiple.

```
Public  class  Pajaro  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 despega, 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.

Una clase también puede implementar más de una interface.

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.

Implementación de Interfaces

```
public class Pajaro extends Animal
                implements Mascota, Volador   {
 String nombre;int velocidadMax;
 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";
 }
}
```

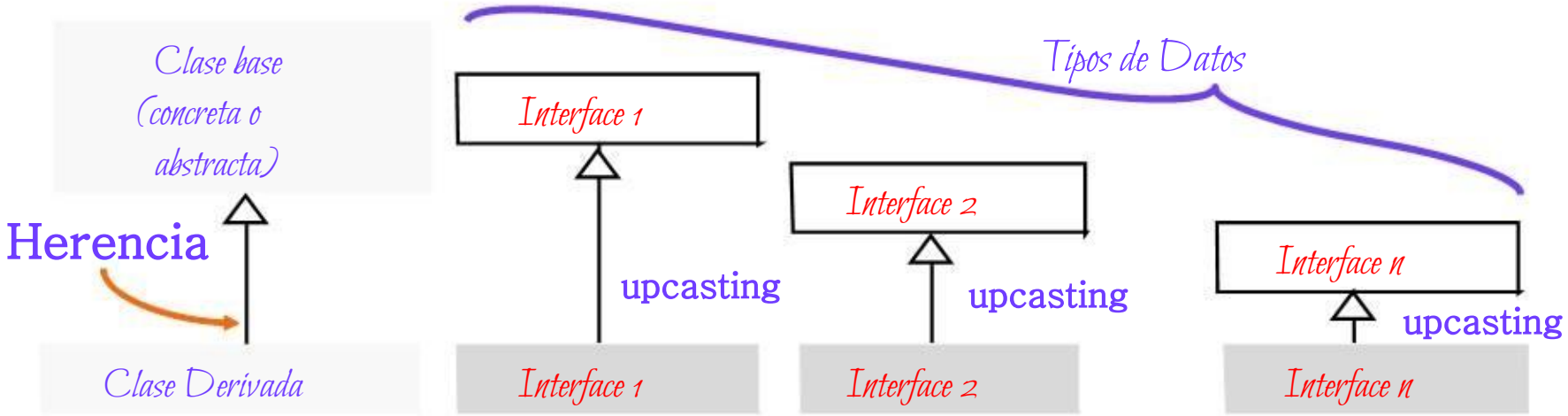
```
public class Avion extends Transporte
            implements Volador{

// Métodos de la Interface  Volador
public String despegar(){
 return "prender turbinas";
}
public String aterrizar(){
    return "Bajar ruedas";
}
public String volar(){
  return "mantener velocidad";
}
}
```

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

Interfaces y herencia múltiple

- Java, NO soporta herencia múltiple pero provee interfaces para lograr un comportamiento "similar".
  Una clase puede heredar de ***una única clase base*** e ***implemenatar tantas interfaces como quiera*** .



Cada una de las interfaces que la clase implementa, provee de un ***tipo de dato*** al que puede hacerse un **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 al método f() pero con distinto tipo de retorno.

Nota: Especificador de acceso

```
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**!!

Interfaces - Clases Abstractas

- Las **interfaces** y las **clases abstractas** proveen una interface común. La **interface** **Volador**, podría definirse como una **clase abstracta**, con tres métodos abstractos: **despegar()**, **aterrizar()** y **volar()**. 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 objectos

¿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);   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 sobre-escribe 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**

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:

La interface 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
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 {
 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(Object o)
   int edad=((Persona) o).getEdad();
   return this.edad - edad;
 }
}
```

La interface Comparable

¿qué pasa si queremos ahora ordenar por apellido?. Queremos que el Arrays.sort(personas) algunas veces ordene por edad y otras por apellido => ¿alcanza con este compareTo(Object o)?

NO!!

```
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()

```
import java.util.*;
public class Persona
          implements Comparable   {
 private String nombre;
 private String apellido;
 private int  edad;

public int compareTo(Object o)
         throws ClassCastException  {
  if  (!(o instanceof Persona))
    throw new  ClassCastException();
  int edad=((Persona) o).getEdad();
  return this.edad  -  edad;
}
}
```

Tendríamos que cambiar el **compareTo()** *cada vez que necesitemos otro orden !!!*

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 int compare(Object o1, Object o2)
```

Se puede pasar instancias de cualquier tipo, pero la intención es que sean del mismo tipo

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 Comparator

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

Faltaría chequear si persona1 y persona2 son instancias de Persona

```
public class ComparadorNombre implements Comparator {
public int compare(Object persona1, Object persona2){
String nom1 = ((Persona) persona1).getNombre().toUpperCase();
String ape1 = ((Persona) persona1).getApellido().toUpperCase();
String nom2 = ((Persona) persona2).getNombre().toUpperCase();
String ape2 = ((Persona) persona2).getApellido().toUpperCase(); if
(!(ape1.equals(ape2)))
    return ape1.compareTo(ape2);
    else
      return nom1.compareTo(nom2);
  }
}
```

Clases creadas especialmente para ordenar objetos de tipo Persona

```
public class ComparadorEdad implements Comparator {
public int compare(Object persona1, Object persona2){
int edad1 = ((Persona) persona1).getEdad();
int edad2 = ((Persona) persona2).getEdad();
   return edad1-edad2;
  }
}
```

La interface 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.