Interfaces

- Permite SIMULAR la **herencia múltiple**.
- La definición de un interfaz no tiene constructor, por lo que no es posible invocar el operador **new** sobre un tipo interfaz.
- **Declaración:**
  - `Interface` es el modo de declarar un tipo formado sólo por métodos abstractos (**`abstract`**) y constantes (**`final`**), ambos **`public`**, permitiendo que se escriba cualquier implementación para estos métodos.
  - Aunque un interfaz puede extender **múltiples interfaces**, no puede extender clases.

```
[public] interface MiInterfaz [ extends otraI1,otraI2,... ] {
            double PI = 3.14159;
            void met1(); //public abstract
            ...
}
```

Interfaces

- **Implementación:**
  - Un interfaz se utiliza definiendo una clase que ***implemente*** el interfaz a través de su nombre
  - La clase debe **proporcionar la definición** completa de todos los métodos declarados en el interfaz y, también, la de todos los métodos declarados en todos los superinterfaces de ese interfaz.
  - Una clase puede implementar **más de un interfaz**, incluyendo varios nombre de interfaces separados por comas. En este caso, la clase debe proporcionar la definición completa de todos los métodos declarados en todos los interfaces de la lista y de todos los superinterfaces de esos interfaces.

```
Class MiClase extends OtraClase
                implements UnInterfaz, OtroInterfaz {
        ...
}
```

Colisión de nombres

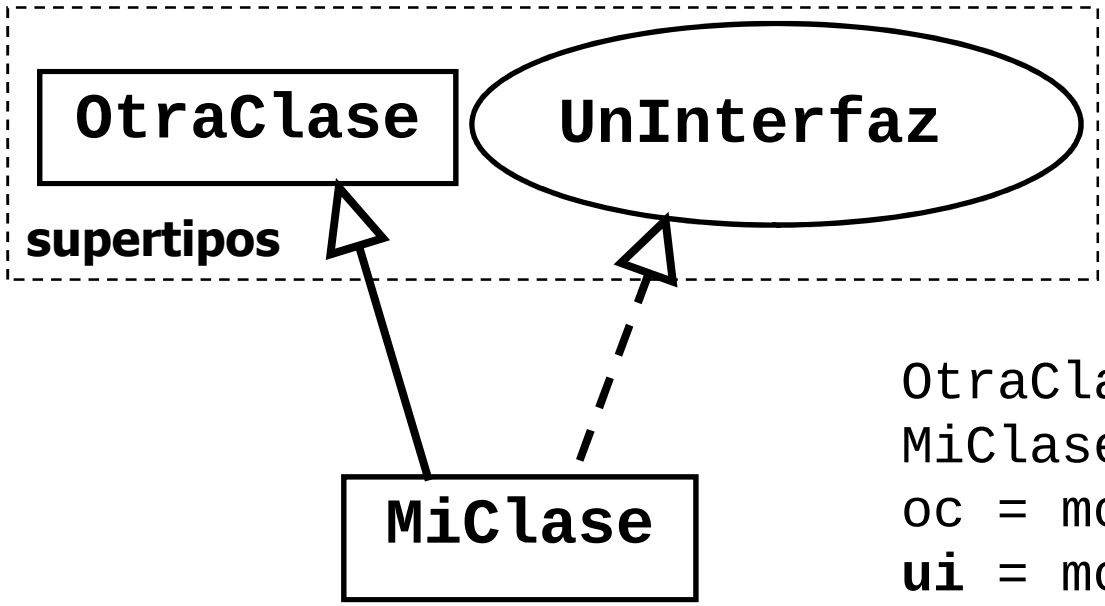
```
public interface Interfaz1 {
  int CTE = 1;
  void met();
}

public interface Interfaz2 {
  int CTE = 789;
  void met();
}


public class Clase implements Interfaz1, Interfaz2{

  public void met(){ //única semántica del método
    System.out.println("Única implementación de met");
    System.out.println("El valor de la cte es" + Interfaz1.CTE);
  }
}
```

Interfaces




```
OtraClase oc; UnInterfaz ui;
MiClase mc = new MiClase();
oc = mc;
ui = mc;
```

- Una interfaz puede utilizarse como nombre de tipo.
- `mc` incluye todos sus supertipos (clases e interfaces).
- A `ui` se le puede asignar cualquier objeto que implemente la interfaz.

Clase abstracta vs interfaces

- Dos **DIFERENCIAS** importantes:
  - Una clase abstracta puede estar ***parcialmente implementada***, partes protected y/o static. Una interfaz está limitada a ***métodos públicos y abstractos***.
  - La s interfaces proporcionan una forma de ***herencia múltiple***. Una clase puede heredar de ***una única clase***, incluso si sólo tiene métodos abstractos.

- Recomendaciones:
  - clase parcialmente diferida $\Rightarrow$ clases **abstractas**
  - clase sin ninguna implementación $\Rightarrow$ **Interfaz**

- Hay cierta **SIMILITUD** entre ambas. El propósito de los interfaces es proporcionar nombres, es decir, solamente declara lo que necesita implementar el interfaz, pero no cómo se ha de realizar esa implementación; es una forma de encapsulación de los protocolos de los métodos sin forzar al usuario a utilizar la herencia.

Clase Arrays

- `static void` **`sort`** `(Object [] a)`

  Aunque el parámetro es un array de `Object` presupone que es un array de objetos comparables (**`Comparable`**`[]`)

- `static void` **`sort`** `(Object [] a,` **`Comparator`** `c)`

- `static boolean` **`equals`** `(Object [] a, Object [] a2)`

- `static int` **`binarySearch`**`(Object [] a, Object key)`

- `static int` **`binarySearch`**`(Object [] a, Object key, Comparator c)`

Interfaces Comparable y Comparator

- La interfaz **java.lang.Comparable** puede ser implementada por cualquier clase cuyos objetos puedan ser ordenados.
- Tiene un único método que devuelve un valor menor, igual o mayor que cero si el objeto actual es menor, igual o mayor que el objeto que se le pasa como parámetro.

```
public interface Comparable{
        int compareTo(Object o);
}
```

- Para las colecciones ordenadas es posible especificar el orden (distinto al *orden natural* definido por el método `compareTo`) que se establece mediante el interfaz **java.util.Comparator**.

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

Ejemplo Comparable

- Compara los empleados de una empresa por antigüedad

```
public class Empleado implements Comparable{
…
  public int compareTo (Object otro){
      int otroAnyo = (Empleado)otro.anyoContrato;
      if (anyoContrato == otroAnyo) return 0;
      else if (anyoContrato < otroAnyo) return -1;
          else return 1;
  }
}
```

- Ordenamos los empleados por antigüedad:

```
Empleado[] plantilla;
 …
Arrays.sort(plantilla);
```

Ejemplo Comparator

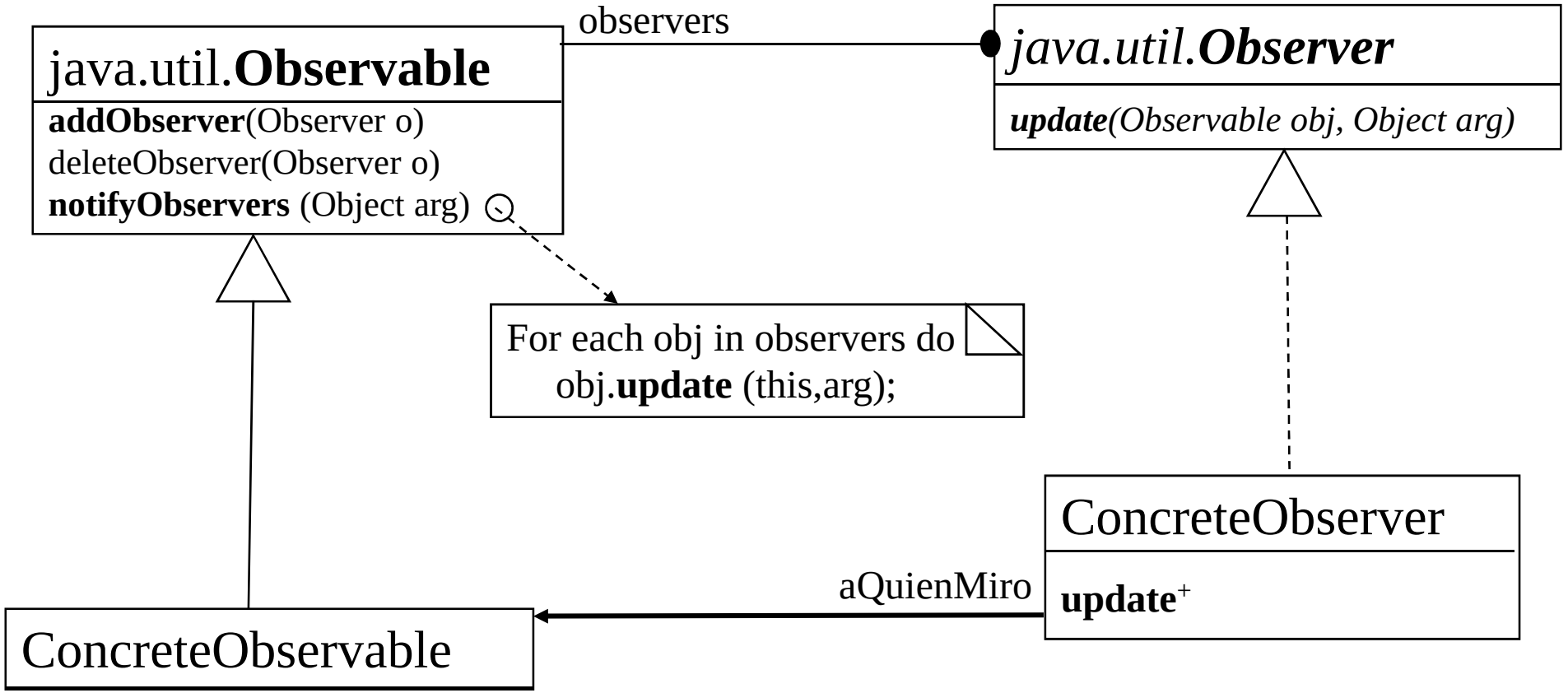
- El criterio para ordenar los empleados atendiendo al orden alfabético de sus nombres:

```
public class ComparadorAlfabetico implements Comparator{
  public int compare(Object o1, Object o2){
   Empleado e1 = (Empleado)o1;
   Empleado e2 = (Empleado)o2;
   return e1.getNombre().compareTo(e2.getNombre());
  }
}
```

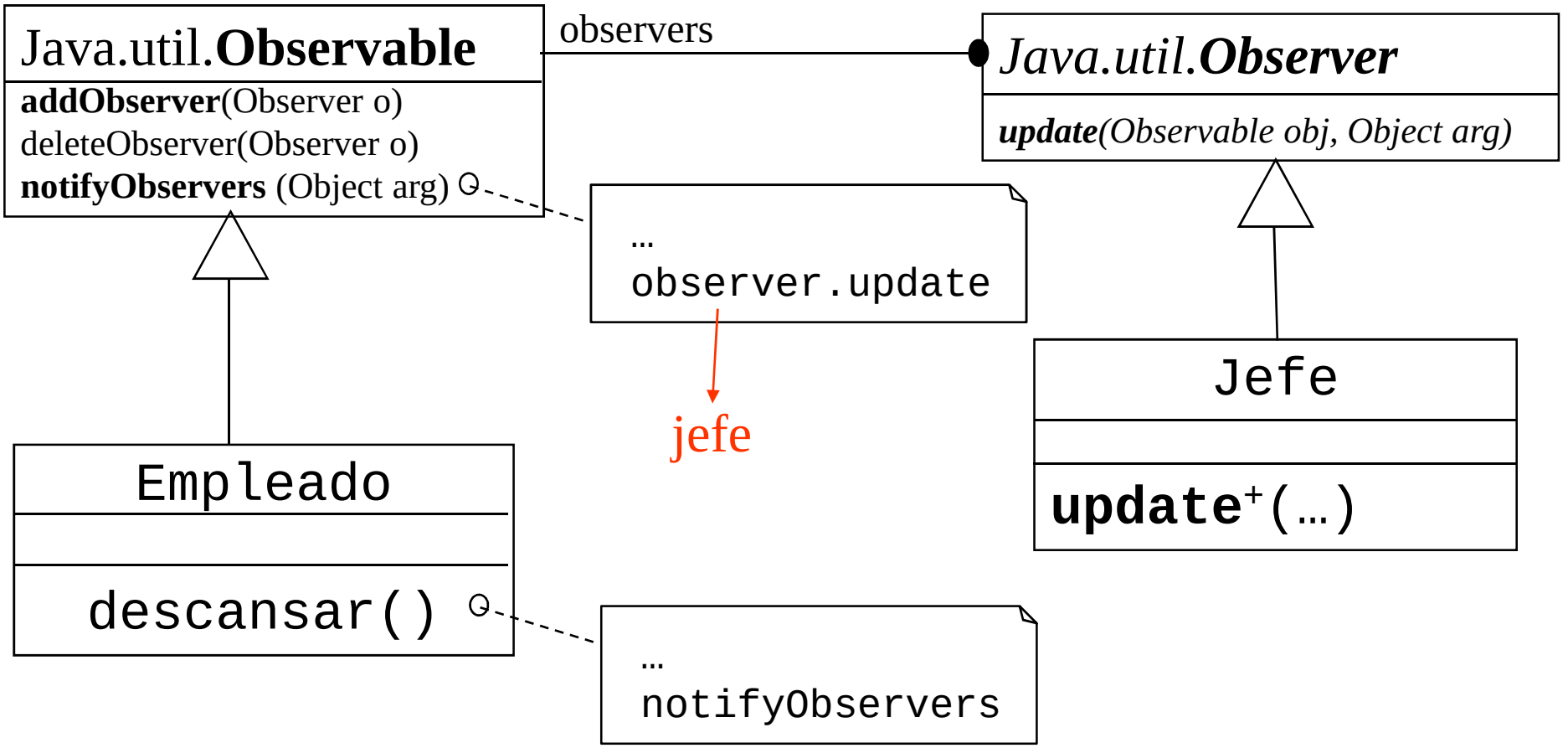
- Ordenamos los empleados por orden alfabético (criterio distinto al “natural”)

```
Empleado[] plantilla;
…
Arrays.sort(plantilla, new ComparadorAlfabetico());
```

Patrón Observer



Patrón Observer



Ejemplo: Observable y Observer

```
public class Empleado extends Observable{
   public void descansar(){
       if (hora!=desayuno) {
              setChanged();
              notifyObservers(“ocioso”);
       }
   }
   ...
}
public class Jefe implements Observer{

   public void supervisar(Empleado e){
       e.addObserver(this);
   }
   public void update (Observable e, Object estado){
       if ((String)estado.equals(“ocioso”))
                     (Empleado)e.darToqueAtencion();
   }
}
```

Clonación de objetos: `Object.clone`

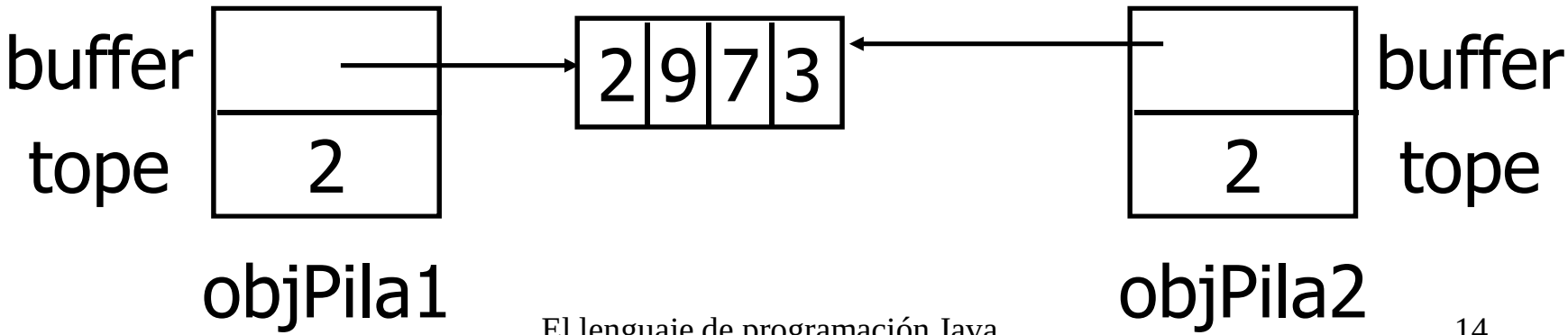
- Devuelve un nuevo objeto cuyo estado inicial es una copia del estado actual del objeto sobre el que se invoca a `clone`
- Factores a tener en cuenta:
  - La clase que proporciona el método **`clone`** debe implementar el interfaz **`Cloneable`**
  - Definir el método `clone` como `public` (en la clase `Object` es `protected`, por lo que no se puede hacer el clone de un `Object`)
  - Puede ser necesario cambiar la implementación por defecto del método para hacer un clone en profundidad
  - Se puede utilizar la excepción **`CloneNotSupportedException`** para indicar que no se debería haber llamado al método clone.

Clonación de objetos

```
public class Pila implements Cloneable {
     public Object clone() throws CloneNotSupportedException{
               return super.clone();
      }
      ...
}
```

- La implementación por defecto hace un clone superficial:

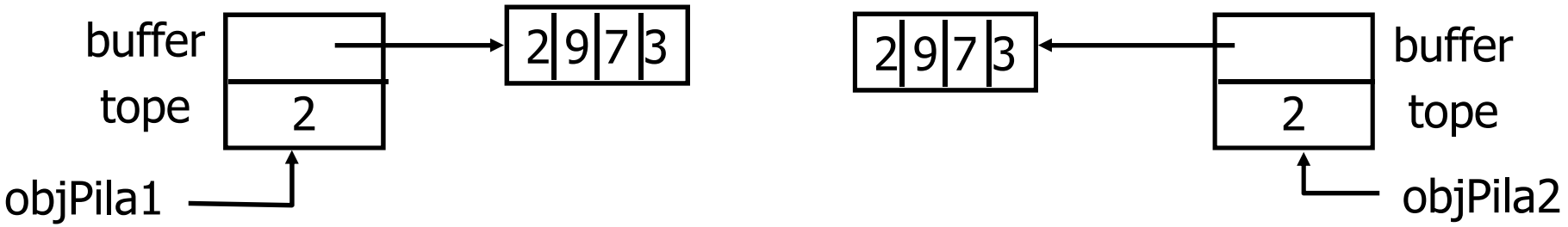
```
objPila2=(Pila)objPila1.clone();
```



Clone en profundidad

- Redefinir clone para que haga una copia en profundidad

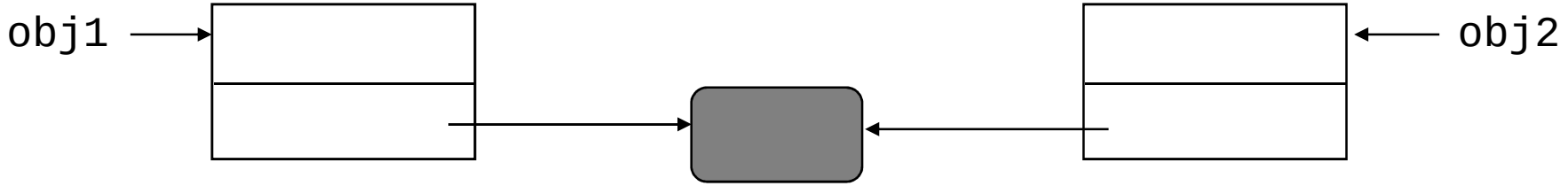
```
public class Pila implements Cloneable{
   ...
   public Object clone() throws CloneNotSupportedException
   {
        Pila nuevaPila = (Pila)super.clone();
        nuevaPila.buffer = (int[])buffer.clone();
        return nuevaPila;
   }
}
```



Interfaz Serializable (java.io)

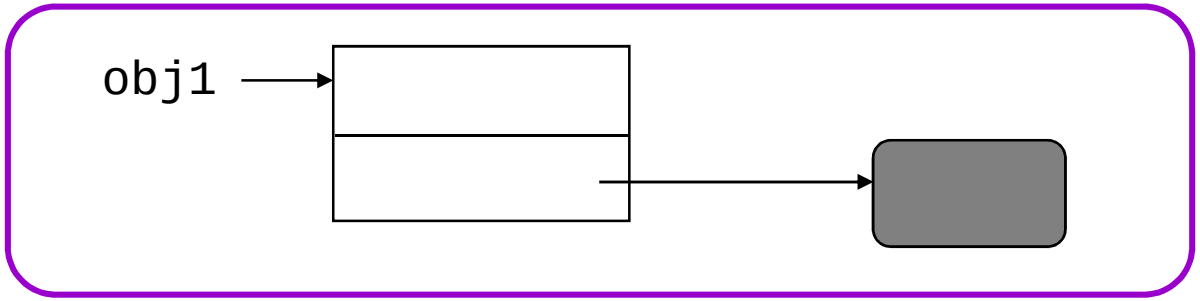
- Convierte un objeto que implemente el interfaz `Serializable` en una secuencia de bytes que puede restablecerse completamente en el objeto original INDEPENDIENTEMENTE de la plataforma donde se haya creado.
- Útil para implementar "**persistencia**" de objetos.
- El interfaz no tiene métodos sirve sólo para identificar la semántica de que es serializable.
- Cualquier subclase de una clase serializable también lo es.
- Este proceso no solo salva una imagen del objeto sino que también, de manera recursiva, guarda todas las referencias que contiene dicho objeto.
- **Si estas serializando en el mismo Stream se recuperará la misma estructura de objetos sin duplicados.**

Efecto de la serialización

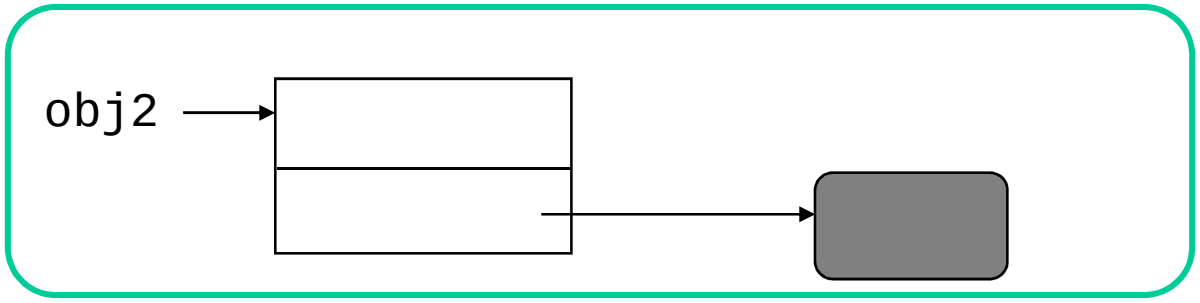


SI guardamos en streams diferentes:

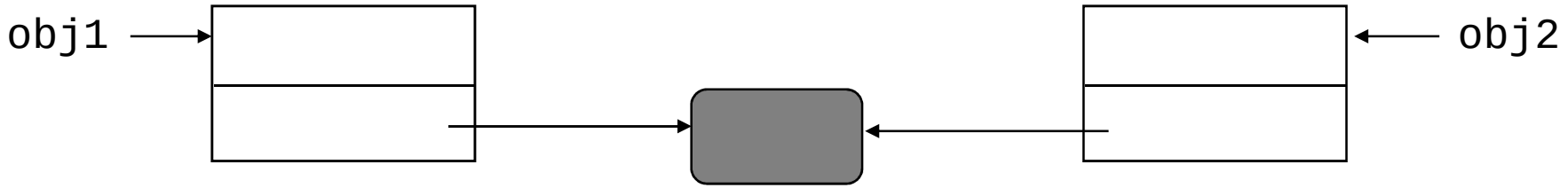
```
stream1.writeObject(obj1);
stream2.writeObject(obj2);
```



SE DUPLICA

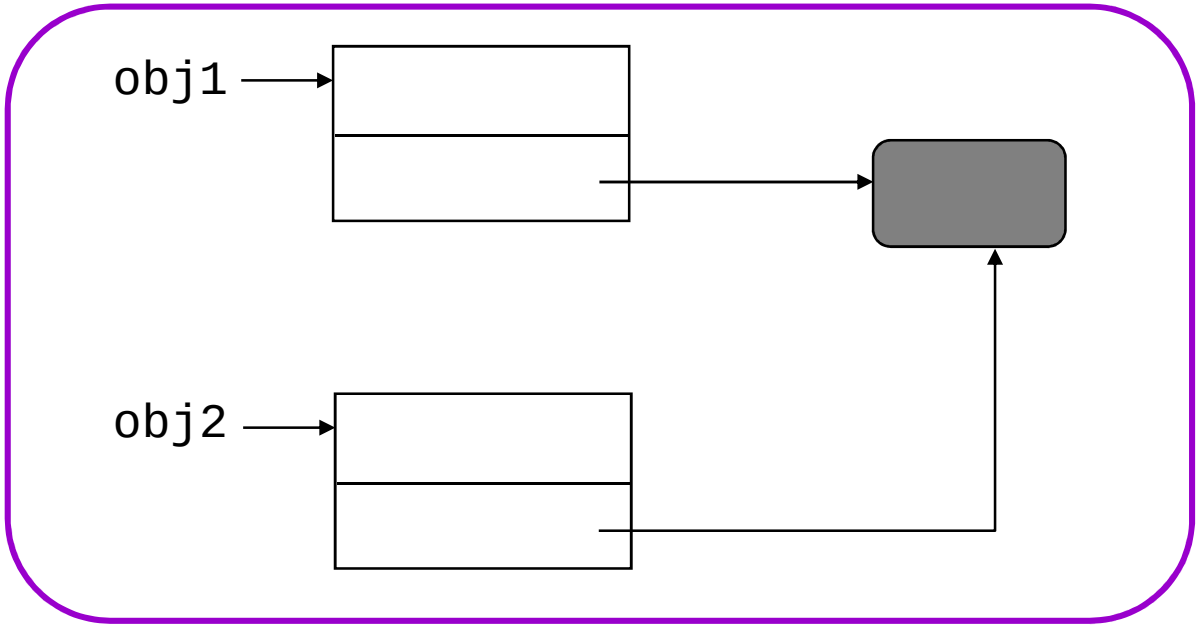


Efecto de la serialización



SI guardamos en el mismo stream:

```
stream1.writeObject(obj1);
stream1.writeObject(obj2);
```



Se mantienen las ref. compartidas

Interfaz Serializable

- Para **serializar** un objeto:
  - Crear algún objeto de clase **`OutputStream`** y encapsularlo en un objeto **`ObjectOutputStream`**
  - invocando a **`writeObject()`**el objeto se serializa y se envía al `OutputStream`
  - Si la clase no implementa la interfaz `Serializable` se lanza la excepción **`NotSerializableException`**.
  - Marcar con **`transient`** los atributos que no se serializan.

- Para **des-serializar** un objeto:
  - Encapsula un objeto **`InputStream`** y encapsularlo en un objeto **`ObjectInputStream`**
  - invocando a **`readObject()`**el objeto se des-serializa y se devuelve una referencia al objeto recuperado
  - downcast para convertir el `Object` a la clase adecuada

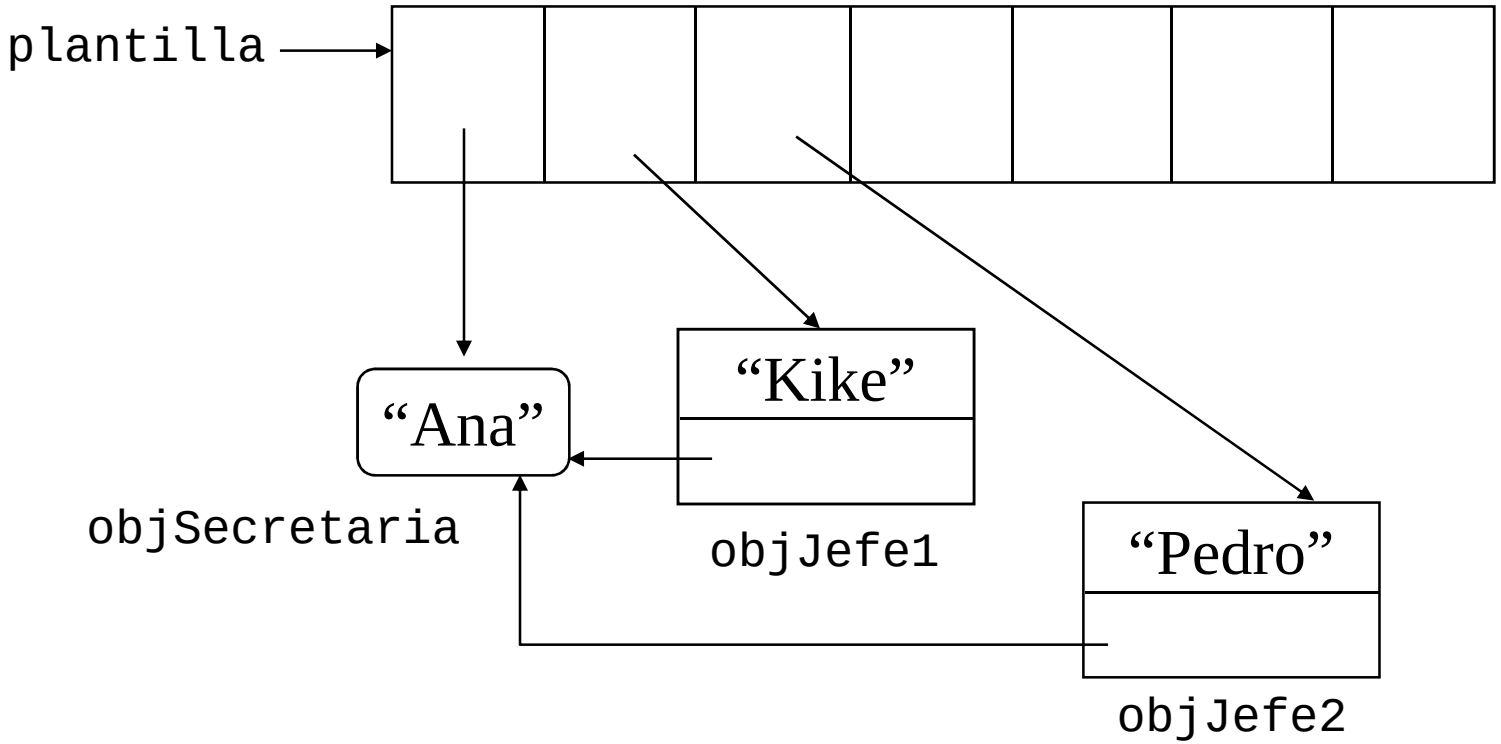
Ejemplo Serializable (guardar)

```
import modelo.*;
import java.io.*;

public class TestSerializable {
  public static void main(String[] args) {
    Empleado[] plantilla = new Empleado[10];
    Secretaria secre = new Secretaria("Ana");
    plantilla[0] = secre;
    plantilla[1] = new Jefe("kike", secre);
    plantilla[2] = new Jefe("Pedro", secre);

    try{
      ObjectOutputStream out = new ObjectOutputStream(new
                               FileOutputStream("empleados.ser"));
      out.writeObject(plantilla);
      out.close();
    }catch (Exception e){
      e.printStackTrace();
    }
```

Referencias compartidas y serialización



`out.writeObject(plantilla)` para conservar las referencias compartidas

Ejemplo Serializable (recuperar)

```
    try{
      ObjectInputStream in = new ObjectInputStream(new
                             FileInputStream("empleados.ser"));
      Empleado [] plantilla2;
      plantilla2= (Empleado[]) in.readObject();
     }catch (Exception e){
      e.printStackTrace();
     }
  }//FIN MAIN
}//FIN TestSerializable
```

- La clase `Empleado` implementa `Serializable`.
- Todos los empleados se guardan en el mismo Stream para no duplicar el objeto `Secretaria`.

Guardar variables de clase

- Las clases que necesiten un tratamiento especial durante la serialización y des-serialización deben implementar métodos especiales con la signatura:

- ```
  private void writeObject(java.io.ObjectOutputStream out)
    throws IOException {}
  ```

- ```
  private void readObject(java.io.ObjectInputStream in)
     throws IOException, ClassNotFoundException {}
  ```

Ejemplo guardar variables static

En la clase **Jugador**:

```
private void writeObject(ObjectOutputStream out)
  throws IOException{
      out.defaultWriteObject();
      out.writeInt(nextNumero);
}

private void readObject(ObjectInputStream in) throws
  IOException, ClassNotFoundException{
      in.defaultReadObject();
      nextNumero = in.readInt();
}
```