Tecnología de la Programación

Tema 4. Herencia

Alberto Díaz
Purificación Arenas, Yolanda García,
Marco Antonio Gómez, Simon Pickin,

Herencia

- Mecanismo exclusivo y fundamental de la POO.
- Es el principal mecanismo de la OO para fomentar y facilitar la reutilización del software.
  - Si se necesita una nueva clase de objetos y se detectan suficientes similitudes con otra clase ya desarrollada, se toma esa clase existente como punto de partida para desarrollar la nueva de forma que:
    - Se adoptan automáticamente características ya implementadas con el ahorro de tiempo y esfuerzo que eso supone.
    - Se adoptan automáticamente características ya probadas, ahorrando tiempo de pruebas y depuración.

Herencia

- Una clase B es una subclase de A si posee o puede acceder a
  - todos los métodos y atributos de una clase A, y
  - métodos y/o atributos adicionales
- Esta definición es compatible con la definición abstracta
  - una subclase posee un comportamiento y unos datos que son una extensión (un conjunto estrictamente mayor) de los de otra clase
- Los objetos de la clase B también pertenecen a la clase A
- Relación es-un/a
  - clase Animal, subclase Perro => perro es un animal
  - clase Persona, subclase Alumno => alumno es una persona

Primera aproximación

```
public class Persona {
    private long dni;
    private String nombre;

    public Persona() { dni = -1; nombre = ""; }
    public Persona(long nuevoDni, String nuevoNombre) {
        this.dni = nuevoDni;
        this.nombre = nuevoNombre;
    }
    .... // omitidos métodos get  y set
    public void mostrar() {
        System.out.println("Nombre: " + this.nombre);
        System.out.println("DNI: " + this.dni);
    }
```

Primera aproximación

- La clase la hemos utilizado durante un tiempo en el desarrollo de una aplicación donde de repente nos surge la necesidad de manejar un tipo concreto de Personas
  - los Alumnos.
- La diferencia es que para los alumnos hay que añadir a lo que ya guarda la clase Persona otro tipo de información
  - el número de matrícula y el número de créditos aprobados.
- En vez de crear una clase Alumno desde cero posiblemente copiando partes del código de Persona, y dado que un alumno es una persona, podemos utilizar el mecanismo de herencia
  - creando una clase Alumno como subclase de Persona

Primera aproximación

```
public class Alumno extends Persona {
        private long numMatricula;
        private int creditosAprobados;

        public Alumno() { numMatricula = -1; creditosAprobados = 0; }

        // omitido constructor con parámetros

        public long getNumMatricula() { return numMatricula; }
        public long getCreditosAprobados() { return creditosAprobados; }
        public void setNumMatricula(int m) { numMatricula = m; }
        public void setCreditosAprobados(int c) { creditosAprobados = c; }
}
```

Primera aproximación

- La clase Alumno hereda de la clase Persona (debido al **extends** Persona utilizado en su declaración).
- Eso significa que las instancias de la clase Alumno son también instancias de la clase Persona, y por lo tanto un objeto de Alumno también tiene disponibles los métodos públicos de la clase Persona:

```
Alumno alum = new Alumno();
alum.setNombre("Walterio Malatesta");
alum.setDNI(12312312);
alum.setCreditosAprobados(9);
alum.mostrar();
```

- El objeto alum anterior incorpora tanto los atributos privados declarados en la clase Alumno como los heredados de la clase Persona, por lo que en memoria almacena dos long, un int y un String.

Primera aproximación

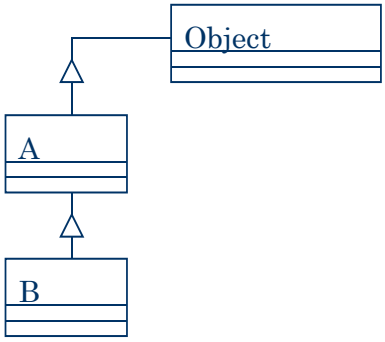
- Se dice que Persona es la clase padre, superclase o clase base de la clase Alumno.
- De forma simétrica, Alumno es la clase hija, subclase o clase derivada de la clase Persona.
- También se dice que Alumno especializa Persona y que Persona generaliza Alumno.

Herencia

- La clase derivada o subclase:
  - Hereda todas las características (atributos y métodos) de la clase base o superclase.
  - Puede definir características adicionales (atributos y métodos).
  - Puede redefinir características heredadas (métodos).
- El proceso de herencia no afecta de ninguna forma a la superclase.

Herencia y jerarquía de clases

- Java posee herencia simple
  - Una clase sólo puede ser subclase de otra
- La relación de herencia induce una organización jerárquica cuya cima es la clase Object
- Si una clase no deriva explícitamente de alguna otra, implícitamente deriva de la clase Object.
- La clase Object, definida en el paquete java.lang, define e implementa el comportamiento común a todas las clases, incluidas las que nosotros definimos.



Constructores

- Los constructores no se heredan

```
public class A
{
  ...
  public A(int ix, int iy){ ... };
}
public class B extends A
{
  ...
}
...
B b = new B(1,2); // error, ningún constructor encaja
```


Constructores

- El constructor de la superclase se puede invocar con super (debe ser la primera instrucción)

```
public class A
{
        private int x, y;
        public A(int ix, int iy) { x = ix; y = iy; }
}
public class B extends A
{
        private int z;
         B(int ix, int iy) { super(ix, iy); z = 0; }
         B(int ix, int iy, int iz) {super(ix, iy); z=iz;}
         B(B b) {z = b.z; super(b.x, b.y); // error
}
```

Constructores

- Los constructores de una subclase siempre deben contener una llamada a un constructor de la superclase
  - super(parámetros);
    - siempre tiene que ser la primera instrucción del código de un constructor
- Si no se pone nada, el compilador asume que hay una llamada sin parámetros:
  - super();
    - Esto implica que la superclase tendría que tener definido un constructor sin parámetros
    - Si sólo tuviera constructores con parámetros, entonces el compilador señalaría el error

Constructores

- Asumiendo que estamos utilizando los constructores sin parámetros, el orden de creación de objetos es el siguiente
  - Localizar la clase padre.
    - Si la clase padre tiene a su vez otra clase padre, continuar hasta encontrar una clase que no tenga padre.
  - Inicializar los atributos de la clase padre localizada (llamando a sus constructores si hiciera falta).
    - Esto implica ejecutar la asignación que esté incluida en la declaración del atributo.
  - Ejecutar el constructor sin parámetros de la clase padre localizada.
  - Inicializar los atributos de la clase hija (llamando a sus constructores si hiciera falta) y continuar con todas las hijas localizadas.
    - Esto implica ejecutar la asignación que esté incluida en la declaración del atributo.
  - Ejecutar el constructor sin parámetros de la clase hija y continuar con los de todas las hijas localizadas.

Sobreescritura de métodos

```
public class A
        private int x, y;

        public void print()
        { System.out.println(x + " " + y); }

public class B extends A
        private int z;
```

- Un problema que tiene la clase B que hereda de A es que el método print sólo muestra las variables x e y

```
B b = new B(1, 2, 3);
b.print();  // 1 2
// No hace lo debido
```


Sobreescritura de métodos

- Algunos de los métodos heredados pueden no resultar adecuados, siendo necesario volver a implementarlos en la subclase.
- La sobreescritura nos permite redefinir la implementación de un método dando una nueva implementación.
- Se crea un método en la subclase con la misma signatura (con el mismo nombre, numero/tipo de parámetros y tipo devuelto que el de la superclase).
- Desde la subclase, se pasará por alto el de la superclase y se ejecutará el de la subclase.
- Si en un método redefinido se quiere ejecutar un método de la superclase que se ha redefinido, se puede utilizar la palabra clave super para pasarle el correspondiente mensaje.

Sobreescritura de métodos

- Para B se puede redefinir el método print ⇒ anulación / redefinición

```
class B extends A
{
  private int z;
  ….
  void print()
 {  System.out.println(x + " ” + y + " ” + z); }
}

B b = new B(1, 2, 3);
b.print(); // 1 2 3    Hace lo debido
```


Sobreescritura de métodos

- Posibilidad de redefinición parcial de métodos

```
class B extends A
{
        void print() // Redefinición parcial
        {
                super.print(); // el print de A
                System.out.println(z);
        }
}
```

Sobreescritura de métodos

- La misma idea se puede aplicar al ejemplo inicial de Persona y Alumno.
- Esto es, deberíamos redefinir el método mostrar en alumno para que mostrara toda la información
  - Usando super.mostrar()

- ¿Qué mostraría este código?

```
public static void main(String args[]) {
    Alumno alumno=new Alumno();
    Persona persona=new Persona();

    persona.mostrar();
    alumno.mostrar();
    }
```


Sobreescritura de métodos

- Cuando se ejecuta en el main el método mostrar del objeto almacenado en la variable persona, es intuitivo que lo que se imprimirá es el nombre y el dni.
- Ahora bien, cuando se invoca mostrar sobre un objeto de la clase Alumno, hay que pensar que hay dos métodos posibles que se pueden ejecutar al tener la misma signatura
  - el mostrar de Alumno y el mostrar de Persona.
- Cuando se dan estos casos, **se elige el método que está en la clase más especializada**, dentro de la parte de la jerarquía que se esté considerando.
  - Así pues, se ejecutaría sólo el método mostrar de Alumno.
- En Java, es posible que el entorno agregue automáticamente la anotación de sobreescritura @Override.
  - Esta anotación sólo indica explícitamente que el método en cuestión sobreescribe un método de alguna clase padre en la jerarquía de herencia a la que pertenece la clase.

Ocultación de métodos estáticos

- Hemos visto la posibilidad de redefinir métodos de la superclase en la clase derivada. Este mecanismo solo hace referencia a métodos de instancias.
- Los métodos static (o métodos de clase) no pueden ser redefinidos.
- Si una subclase define un método static con la misma signatura que un método static de la superclase, el método de la clase derivada oculta al de la superclase.

Redefinición de atributos (no debiera ocurrir nunca)

- Podemos utilizar un nombre de un atributo de la superclase para un atributo de una subclase.
- Esto produce una ocultación del atributo de la superclase.
- Dentro de la subclase, si queremos referenciar el atributo de la superclase debemos hacerlo mediante super (sólo si son public o protected).

Redefinición y sobrecarga

- No confundir redefinición con sobrecarga.
  - La sobrecarga de métodos se refiere a la definición de varios métodos con el mismo nombre y diferente lista de parámetros.
  - La redefinición vuelve a definir en una clase derivada un método de la superclase.

Subtipos y subclases

- Desde un punto de vista teórico podemos ver las clases como tipos o conceptos de forma que cuando establecemos una relación de herencia lo que estamos indicando es que la nueva clase es un subtipo o subconcepto.
  - todos los mamíferos son vertebrados (y por tanto tienen una serie de características y comportamientos comunes con el resto de vertebrados no mamíferos)
  - todos los alumnos son personas, y se comportarán como tal.
- No obstante, subtipo y subclase son dos cosas independientes, el lenguaje/compilador no puede garantizar que se mantengan
  - el programador de la clase hija podría sobreescribir un método de la clase padre de tal forma que su semántica no tenga nada que ver con el subconcepto al que pertenece.

Subtipos y subclases

- Podemos decir que:
  - Una clase es subclase de otra si ésta ha sido construida usando el mecanismo de herencia.
  - Una clase es subtipo de otra si preserva el propósito original, o si cumple el principio de sustitución enunciado más abajo.
  - Todos los subtipos son subclases pero no todas las clases son subtipos.
- El principio de sustitución dice que si B es subclase de A entonces en cualquier situación se puede sustituir una instancia de A por una de B obteniéndose el mismo comportamiento observable.
  - Es una cuestión de disciplina del programador

Visibilidad

- Ya vimos como los distintos niveles de protección limitan el acceso a los miembros de la clase desde el exterior.
- ¿Pero como afectan estos niveles de protección a los miembros heredados?
  - Miembros públicos -> Son accesibles desde los descendientes, y se heredan como públicos
  - Miembros privados -> No son accesibles desde los descendientes.
- Un nuevo nivel de protección
  - miembros protected.
- Un miembro protegido es accesible únicamente desde los descendientes.

Visibilidad

- Las reglas de visibilidad completas para Java aparecen en la tabla siguiente.

	Miembros de la superclase			
	Públicos	Protegidos	Por defecto	Privados
Clase en el mismo paquete	Sí	Sí	Sí	No
Subclase en el mismo paquete	Sí	Sí	Sí	No
Clase en otro paquete	Sí	No	No	No
Subclase en otro paquete	Sí	Sí	No	No

- En ella aparece a qué elementos de una clase se puede acceder desde fuera en base a si se hace en el mismo paquete y como clase derivada o no.

Clases abstractas

- Una clase abstracta es una clase que se introduce sólo para que se deriven nuevas clases de ella, no para que se creen objetos con su nombre.
- Cuando las clases se organizan en jerarquía, lo normal es que las clases que representan los conceptos más abstractos ocupen un lugar más alto en la jerarquía.
- Una clase abstracta no se puede instanciar.
- La utilidad de este tipo de clases está en la aplicación de herencia para obtener clases que representan conceptos concretos para los que sí que tiene sentido su instanciación.
- Clase abstracta:
  - Modela el comportamiento común de sus clases derivadas.
  - Implementa métodos que son comunes a todas sus subclases.
  - Establece métodos que necesariamente han de ser implementados por sus subclases (métodos abstractos).

Clases abstractas

- En las clases abstractas pueden aparecer métodos sin cuerpo (virtuales o abstractos).
- Se obliga a que todas las clase derivadas ofrezcan una parte común en su interfaz.
- Creamos un método abstracto para ver si una persona trabaja, y este método se implementará en las clases derivadas.

```
abstract class Trabajador {
        abstract public boolean trabaja();
}
```

- Esto obliga a redefinir el método trabaja() en las clases derivadas.

- Cuando una clase contiene un método abstracto tiene que declararse abstracta.
- No todos los métodos de una clase abstracta tienen que ser abstractos.
- Las clases abstractas no pueden tener métodos privados (no se podrían implementar) ni tampoco estáticos.

Clases abstractas

- Una clase abstracta no se puede usar para crear un objeto

```
C c1 = new C(); // error
```

- Sólo se pueden crear objetos de clases derivadas de la clase abstracta

```
public class L extends C
{
        private int z;
        public L(int nz)        { z = nz; }
        public void imprimir() {System.out.println(z);}
}

// Ahora sí!
L l1 = new L(3);
```

Clases finales

- Una clase final es una clase que no se puede derivar ⇒ no posee subclases

```
public final class B extends A
{
  ...
}
public class C extends B
{ // error B es final
  ...
}
```


Métodos finales

- Un método final es un método que no se puede redefinir en una subclase de la clase en donde se definió

```
final void mover(int dx, int dy)
{
        x += dx; y += dy;
} // No redefinible
```

Claúsulas de visibilidad

- Resumen de cláusulas de visibilidad
  - Atributo: {private | public | protected} {final} {static}
  - Método: {private | public | protected} {final | abstract} {static}
  - Clase: {public} {final | abstract}
- Sólo se puede especificar una de las cláusulas puestas en la misma llave

Clases Abstractas: Un ejemplo

```
abstract public class Figura {
   // centro de la figura
   protected int x;
   protected int y;

   public Figura(){
       this.x=0; this.y=0;
   }
   public Figura(int xx, int yy){
       this.x = xx; this.y = yy;
   }
   abstract public int area();
   abstract public void drawFigura();
}
```

Clases Abstractas: Un ejemplo

```
public class Rectangulo extends Figura {

private int base;
private int altura;

public Rectangulo(int xx, int yy,int b, int a){
  super(xx,yy);
  this.base=b; this.altura=a;
}
public int area(){
  return this.base * this.altura;
}
public void drawFigura(){…}
}
```


Clases Abstractas: Un ejemplo

```
public class Cuadrado extends Figura {
   private int lado;

public Cuadrado(int xx, int yy, int l){
   super(xx,yy);
   this.lado=l;
}
public int area(){
   return lado*lado;
};
public void drawFigura(){…}
}
```

Interfaces

- Podemos llevar al extremo las clases abstractas y hacer una clase que tenga todos los métodos abstractos. Esa es la idea de una interfaz.

- En Java las interfaces se pueden ver como clases:
  - que no tienen atributos
  - que sólo declaran signaturas de métodos y que nunca proporcionan un cuerpo de método
  - que todos los métodos declarados son públicos
  - que pueden heredar de otros interfaces y pueden hacerlo de más de una

Interfaces

- Un interface es una colección de definiciones de métodos (sin implementaciones) y de valores constantes.
- Los interfaces se utilizan para definir un protocolo de comportamiento que puede ser implementado por cualquier clase.
- Los interfaces son útiles para:
  - Capturar similitudes entre clases no relacionadas sin forzar una relación entre ellas.
  - Declarar métodos que una o varias clases necesitan implementar.
- En Java, un interfaz es un tipo de dato de referencia, y por tanto, puede utilizarse en muchos de los sitios donde se pueda utilizar cualquier tipo (como en un argumento de métodos y una declaración de variables).

Interfaces

- Para declararlos se utiliza la palabra clave **`interface`** en vez de **`class`**
- Normalmente irá en un fichero **`.java`** independiente igual que una clase.

```
public interface EjemploInterfaz1 {
        public void metodo1();
        public Integer metodo2();
        public Integer metodo3();
}

public interface EjemploInterfaz2 {
        public void metodo4();
        public Integer metodo5();
        public Integer metodo6();
}
```


Interfaces

- Las interfaces se **implementan** en clases
- Implementar requiere usar la palabra reservada **`implements`** asociada a tantos interfaces como se quiera.
- Se fuerza a la clase a proporcionar un cuerpo para los métodos declarados (también puede dejar alguno sin implementar convirtiendo la clase en clase abstracta).

```
class EjemploImplementación implements EjemploInterfaz1, EjemploInterfaz2 {
        public void metodo1() { ....}
        public Integer metodo2(){ ....}
        public Integer metodo2(){ ....}
        public void metodo4(){ ....}
        public Integer metodo5(){ ....}
        public Integer metodo6(){ ....}
}
```

Interfaces

- El uso de los interfaces se hace declarando variables del mismo tipo que la interfaz
- Posteriormente se asignan instancias de clases que implementen la interfaz
- Es muy útil cuando se trata de ocultar partes de una aplicación.

```
class EjemploImplementación implements EjemploInterfaz1, EjemploInterfaz2 {
{ ....}
public static void main(String args[]){
EjemploImplementación ejemplo=new EjemploImplementación();
EjemploInterfaz1 i1=ejemplo;          // i1 sólo puede ver los 3 métodos de EjemploInterfaz1
EjemploInterfaz2 i2=ejemplo;          // i2 sólo puede ver los 3 métodos de EjemploInterfaz2
i1.metodo1();
}
}
```


Interfaces: ejemplo

```
interface Animal
{
   public void habla();
}

interface Bestia
{
   public void grita();
}
```

Interfaces: ejemplo

```
public class Vaca implements Animal,Bestia
{
    public void habla()
    {
        System.out.println("Soy una vaca");
    }
    public void grita()
    {
        System.out.println("Que Soy una vaca!!!!!!");
    }
}
public class Perro implements Animal,Bestia
{
    public void habla() {...}  public void grita() {...}
}
public class Gato implements Animal,Bestia{. . . }
```


Interfaces: ejemplo

```
public class Test
{
    public static void main (String args[])
    {
       Perro p = new Perro();
       Vaca v = new Vaca();

        p.habla();
        p.grita();

        v.habla();
        v.grita();
   }
```

Interfaces

- Hay muchos interfaces que define Java que saldrán en la asignatura:
  - Serializable (clases que se pueden serializar),
  - Runnable (para indicar código a ejecutar concurrentemente), o
  - ActionListener (para indicar una acción a realizar cuando se produce un evento concreto).
- Cómo se implementen estos interfaces queda a nuestra discreción.
- Sólo se pide que se implementen los métodos que se piden, dejando libertad para determinar estructuras de datos y métodos adicionales que necesitemos.

Resumen de Interfaces

- Una clase puede implementar varios interfaces
  - tiene que implementar todos los métodos
  - si queda alguno sin implementación, la clase tiene que ser abstracta
- Jerarquía de clases interface
  - Heredan de (extienden a) otros interfaces
  - Su jerarquía es diferente de la de las clases
  - Permiten herencia múltiple (heredar de varios interfaces)
- Los interfaces son tipos
  - podemos declarar referencias de ellos
  - se les puede asignar objetos de clases que implementen los interfaces

Interfaces versus clases abstractas

- Interfaces
  - si se necesita herencia múltiple
  - Comportamiento común en clases en jerarquías de herencia distintas
- Clases abstractas
  - si hay implementaciones parciales
  - si se necesita control de acceso
  - si se necesitan métodos estáticos

Clase Object

- Object define un conjunto de métodos útiles, que pueden ser redefinidos en cada clase. En particular:
  - public boolean equals (Object o)
    - Permite definir el criterio de igualdad utilizado para los objetos de una determinada clase (el operador == sólo chequea igualdad de referencias)
  - public String toString()
    - Permite decidir la representación externa de un objeto
    - NombreClase@direcciónMemoriaHexadecimal
  - public Object clone()
    - Crea y devuelve una copia
  - public Class getClass()
    - devuelve la clase a la que pertenece el objeto con el que se le invoca

Clase Object: método equals()

- Redefinición de equals // por defecto sólo referencias

```
public class Otra
{
    private int x;
    public boolean equals (Object obj)
    {
        if (this == obj) return true;
        if (obj == null) return false;
        if (this.getClass() != obj.getClass()) return false;

        Otra otra = (Otra) obj;
        return otra.x == this.x;
    }
}
```


El método equals() y el método hashCode()

- Propiedades que debería preservar el método sobrescrito equals()
  - a.equals(b) ⇔ b.equals(a) (conmutatividad)
  - a.equals(a) (reflexividad)
  - a.equals(b) and b.equals(c) ⇒ a.equals(c) (transitividad)
  - Y el sentido común dice que:
  - a.equals(b) ⇒ a == b (coherencia1)
- Cada clase hereda el método hashCode() de la clase Object
  - Es utilizada por colecciones tales como Hashtable y HashMap
- Siempre debería ser cierto que:
  - a.equals(b) ⇒ a.hashCode() == b.hashCode() (coherencia2)
- Por tanto, una clase que sobresescribe equals()
  - también debe sobreescribir hashCode()

El método hashCode()

- Algunas colecciones usan el valor hashcode para ordenar y localizar a los objetos que están contenidos dentro de ellas.
- El hashcode es un numero entero, sin signo, que sirve en colecciones de tipo hash para un mejor funcionamiento en cuanto a rendimiento.
- Este método debe ser sobrescrito en todas las clases que sobrescriban el método equals, si no se quiere tener un comportamiento extraño al utilizar las colecciones de tipo hash y otras clases.
- Se recomienda utilizar números primos multiplicados por los valores de los atributos, para una mejor distribución del hashcode generado.

El método hashCode()

```
public class Empleado {
   private int idEmpleado;
   private String nombre;
 // Métodos de la Clase
@Override
public int hashCode() {
   int hash = 1;
   hash = hash * 17 + idEmpleado;
   hash = hash * 31 + nombre.hashCode();
   return hash;
}
}
```

Copiar objetos con el método Java clone()

- La asignación entre objetos es una asignación de referencias.
- Para crear una copia de un objeto, existe el método Java clone()
  - Un método de la clase Object (que se puede sobrescribir)

```
Fecha f1, f2;
f2 = f1.clone();   // Copia del objeto

if (f1 == f2)      // Esto NO será cierto
 System.out.println("El mismo");

if (f1.equals(f2)) // Esto sí (si lo hemos sobrescrito)
 System.out.println("Iguales");
```


Clase Object: método clone()

- clone()
  - Crea un objeto que es una copia “superficial” del objeto actual independientemente de su tipo
  - El tipo objeto a copiar debe implementar el interfaz vacío Clonable
  - Es un método protegido de Object, para usarlo desde otro paquete hay que redeclararlo como público

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

Clase Object: método clone()

- clone()
  - Si los atributos de una clase son objetos es necesario una copia profunda
    - por defecto, clone copia las referencias de los atributos
    - Esa copia es segura si:
      - El atributo es de un objeto inmutable, es decir, no se puede modificar (P. ej.: String)
      - El atributo permanece constante, no se modifica ni existen métodos que lo referencien.
  - hay que llamar a los métodos clone() de cada uno de los objetos de los que queramos crear una copia
  - En la ejecución de clone() no se llama a ningún constructor
  - Los arrays implementan la interfaz Cloneable

Clase Object: método clone()

```
public class B implements Cloneable
        private int x;
public Object clone() throws CloneNotSupportedException
        B copia = (B) super.clone();
        return copia;

public class A implements Cloneable
        private int y;
        private B b;
public Object clone() throws CloneNotSupportedException
        A copia = (A) super.clone();
        copia.b = (B) this.b.clone();
        return copia;
```

Clase Object: método clone()

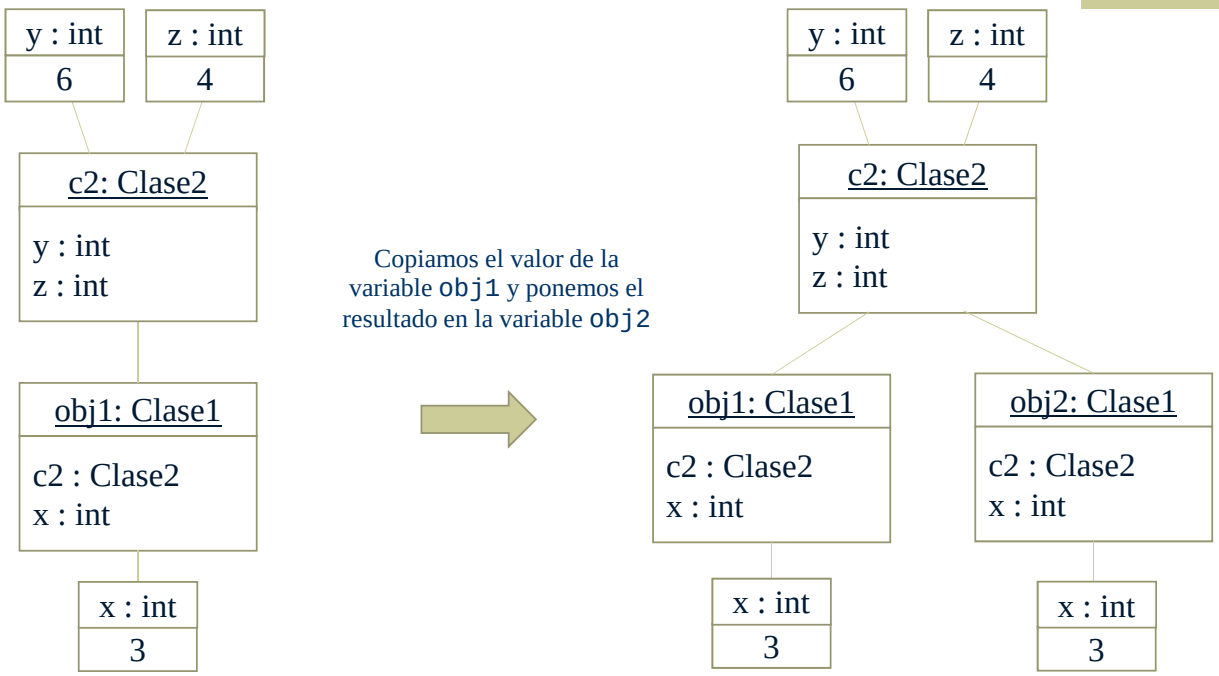
```
public class B implements Cloneable
        private int[] x;
public Object clone() throws CloneNotSupportedException

         B copia = (B) super.clone();
          copia.x = (int[]) this.x.clone();
          return copia;
```


Object

```
public class A implements Cloneable
        private int y;
        B[] b;
public Object clone() throws CloneNotSupportedException
        A copia = (A) super.clone(); copia.b = (B[]) this.b.clone();
        for (int i=0; i < copia.b.length; i++)
                copia.b[i] = (B) this.b[i].clone();
        return copia;
```

Ejemplo: copia superficial



Ejemplo: copia profunda