

Copia de Objetos

Dpto. Lenguajes y Sistemas Informáticos
Universidad de Granada

Programación y Diseño Orientado a Objetos

(Curso 2023-2024)

Créditos

- Las siguientes imágenes e ilustraciones son libres y se han obtenido de:
 - ▶ Emojis, <https://pixabay.com/images/id-2074153/>
- El resto de imágenes e ilustraciones son de creación propia, al igual que los ejemplos de código

Objetivos

- Entender la diferencia entre realizar
 - ▶ Copia de identidad
 - ▶ Copia de estado superficial
 - ▶ Copia de estado profunda
- Entender qué son los objetos inmutables y su relación con la copia de objetos
- Saber qué es la copia defensiva y los problemas que puede haber si no se usa
- Saber realizar copias profundas con clone y con constructores de copia

Contenidos

1 Introducción

2 Copia defensiva

- clone y la interfaz Cloneable
- Constructor copia
- Ruby y clone
- Copia por serialización

Introducción a la copia de objetos

- Una operación muy habitual en programación es esta:

Pseudocódigo: Asignación

```
1 copia = original;  
2 // ¿Qué se realiza con dicha operación?  
3 // Si modifico original ¿se ve afectada copia? ¿Y si modifico copia?
```

- No es una operación trivial tratándose de objetos
- De hecho, existen distintos casos:
 - ▶ Copia de identidad
 - ▶ Copia de estado
 - ★ ¿Y si un atributo del objeto a copiar referencia a otro objeto?
 - ★ ¿Cómo lo copiamos? ¿Por identidad o por estado?
 - ★ Si es por estado, ¿cuándo parar?
 - ▶ Todo esto puede quedar “oculto” bajo el operador de asignación

Introducción a la copia de objetos

- Aparecen dos conceptos

- ▶ Profundidad de la copia

- ★ Hasta que nivel se van a realizar copias de estado en vez de identidad

- ▶ Inmutabilidad de los objetos

- ★ Un objeto es inmutable si no dispone de métodos que modifiquen su estado

- Hay que tener cuidado con los objetos que referencian a otros si estos no son inmutables

★ ¿Por qué?

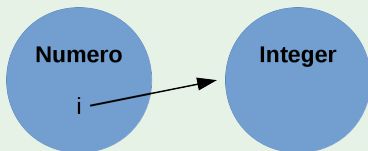


Ejemplo

- ¿Por qué puede ser necesario realizar copias de estado?

Java: Una clase mutable que almacena un entero

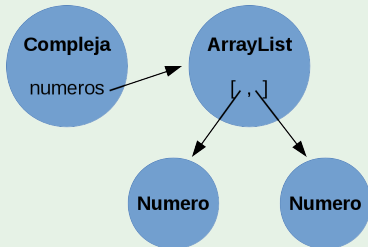
```
1 class Numero{
2     private Integer i;
3
4     public Numero(Integer a) {
5         i = a;
6     }
7
8     public void inc() {
9         // Este método modifica el estado del objeto
10        // La clase es mutable
11
12        i++;
13    }
14
15    @Override
16    public String toString() {
17        return i.toString();
18    }
19 }
```



Ejemplo

Java: Una clase que es una colección de los números anteriores

```
1 class Compleja {  
2     // Clase mutable que referencia a objetos mutables  
3  
4     private ArrayList<Numero> numeros;  
5  
6     public Compleja() {  
7         numeros = new ArrayList<>();  
8     }  
9  
10    public void add(Numero i) {  
11        numeros.add(i);  
12    }  
13  
14    ArrayList getNumeros() {  
15        return numeros;  
16    }  
17 }
```



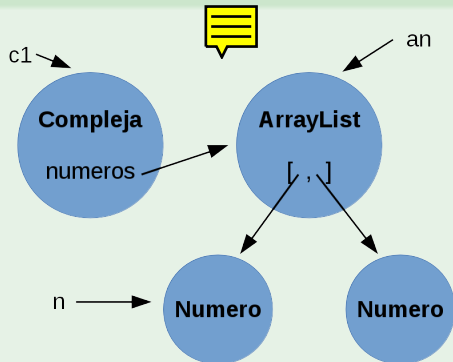
Ejemplo

Java: Probamos las clases anteriores

```

1 Compleja c1 = new Compleja();
2
3 c1.add (new Numero(3));
4 c1.add (new Numero(2));
5
6 // Acceso sin restricción a la lista
7 ArrayList<Numero> an = c1.getNumeros();
8 an.clear();
9 c1.add (new Numero(33));
10
11 for (Numero i : c1.getNumeros()) {
12     System.out.println(i); // R--> 33
13 }
14 //Se devuelven referencias al estado interno
15 Numero n = c1.getNumeros().get(0);
16 n.inc();
17
18 for (Numero i : c1.getNumeros()) {
19     System.out.println(i); // R--> 34
20 }

```



- Se expone:

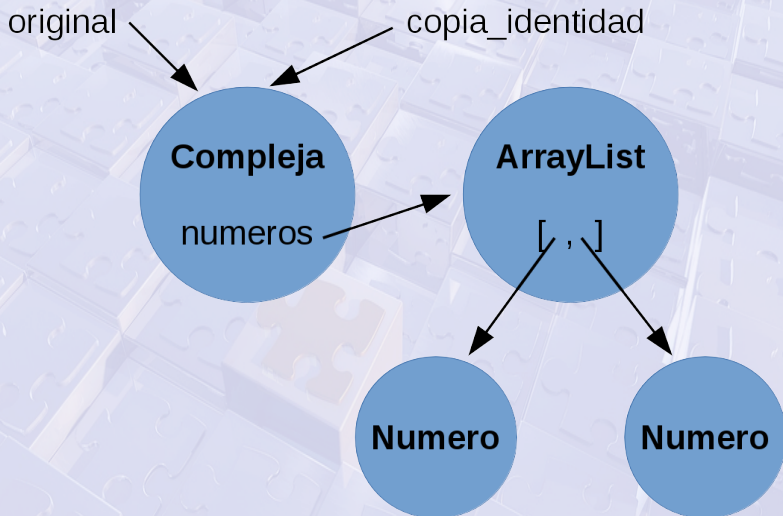
- ▶ El estado de los objetos Compleja
- ▶ El de los objetos Numero referenciados por los objetos Compleja

Copia defensiva

- Alude a devolver una copia de estado en vez de devolver una copia de identidad
 - ▶ **Objetivo:** Evitar que el estado de un objeto se modifique sin usar los métodos que la clase designa para ello
 - ▶ **Requisito:** Realizar copias profundas y no solo copias superficiales
 - ▶ **Cuándo:** Los consultores deben usar este recurso con los objetos mutables que devuelvan
- En el ejemplo anterior:
 - ▶ La lista de números no es inmutable porque existen métodos para poder alterarla
 - ▶ Se debería duplicar la lista y devolver esa copia en el consultor `getNumeros()`

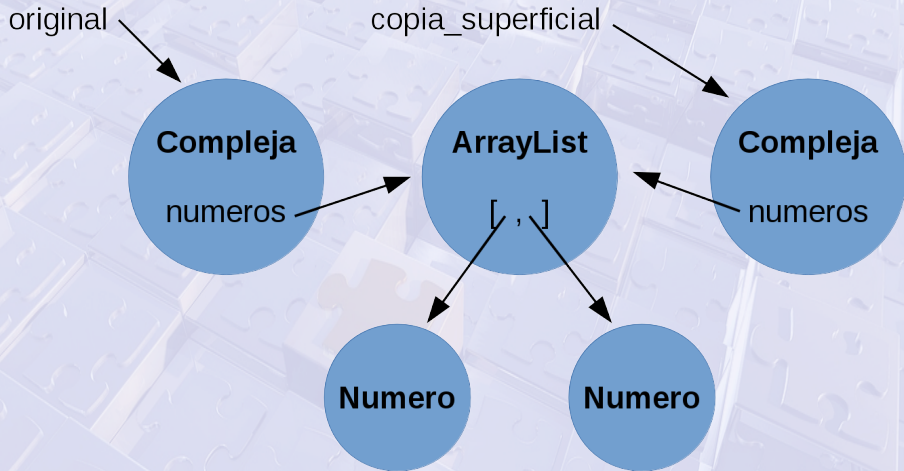
Tipos de copia

Copia de identidad



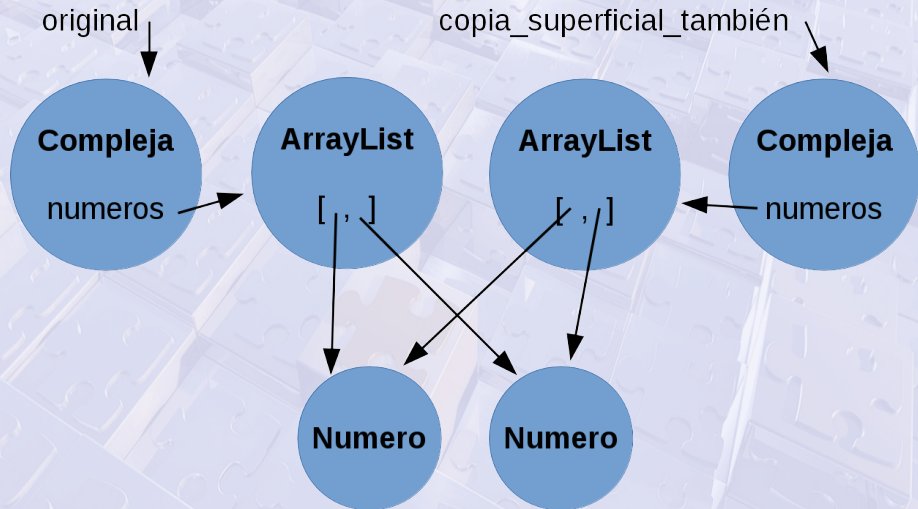
Tipos de copia

Copia superficial



Tipos de copia

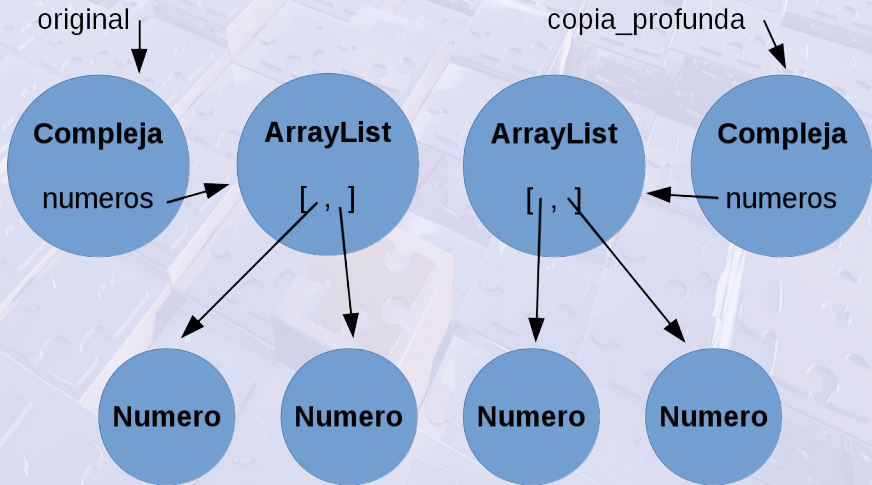
Copia superficial



Tipos de copia

Copia profunda

- **Todos** los objetos mutables se han duplicado



Ejemplo

Java: Version un poco más segura de la clase Compleja

```
1 class ComplejaSegura {
2     private ArrayList numeros;
3
4     public ComplejaSegura() {
5         numeros = new ArrayList();
6     }
7
8     public void add(Número i) {
9         numeros.add(i);
10    }
11
12    ArrayList getNumeros() {
13        return (ArrayList) (numeros.clone());
14        // Usamos clone para devolver una copia del estado de numeros
15    }
16 }
```

Ejemplo

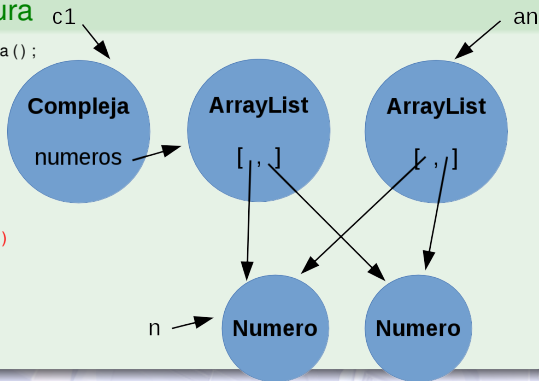


Java: Usando ComplejaSegura

```

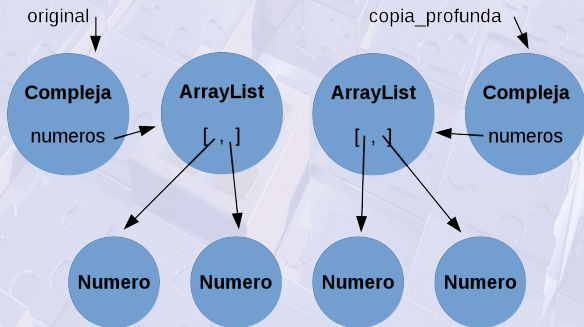
1 ComplejaSegura c1 = new ComplejaSegura();
2
3 c1.add (new Numero(3));
4 c1.add (new Numero(2));
5
6 c1.getNumeros().clear();
7 for(Numero i : c1.getNumeros()) {
8     System.out.println(i);
9 } // R --> 3 2    OK!    :-)
10
11 ArrayList<Numero> an = c1.getNumeros();
12 Numero n = an.get(0)
13 n.inc(); // Sigue siendo un problema
14
15 for(Numero i:c1.getNumeros()) {
16     System.out.println(i);
17 } // R--> 4 2    :-)

```



Copia profunda

- Hay que llegar a nivel requerido en cada caso
- En el ejemplo anterior no solo hay que duplicar la lista sino también los elementos de la misma
 - ▶ En caso contrario ambas listas compartirán las referencias a los mismos objetos
 - ▶ Y eso es un problema al tratarse de objetos mutables



Ejemplo

Java: Mejorando la clase Numero

```
1 class Numero implements Cloneable{
2     private Integer i;
3
4     public Numero(Integer a) {
5         i = a;
6     }
7
8     public void inc() {
9         i++;
10    }
11
12    @Override
13    public Numero clone() throws CloneNotSupportedException {
14
15        return (Numero) super.clone();
16
17        // Los objetos de la clase Integer son inmutables
18        // por ello, la implementación realizada es válida
19        // no siendo necesaria la implementación que está comentada
20
21        /*
22        Numero nuevo = (Numero) super.clone()
23        nuevo.i = i.clone();
24        return nuevo;
25        */
26    }
27 }
```



Ejemplo

Java: Mejorando más la clase Compleja

```

1 class ComplejaMasSegura implements Cloneable {
2     // ...
3     ArrayList getNumeros() {
4         ArrayList nuevo = new ArrayList();
5         Numero n = null;
6         for (Numero i : this.numeros) {
7             try {
8                 n = i.clone();
9             } catch (CloneNotSupportedException e) {
10                 System.err.println ("CloneNotSupportedException" );
11             }
12             nuevo.add (n);
13         }
14         return (nuevo);
15     }
16
17     @Override
18     public ComplejaMasSegura clone() throws CloneNotSupportedException {
19         ComplejaMasSegura nuevo= (ComplejaMasSegura) super.clone();
20         nuevo.numeros = this.getNumeros(); // Ya se hace copia profunda
21         return nuevo;
22     }
23
24     // También habría que modificar el método public void add (Numero i)
25     // ¿Cómo se implementaría?
26 }

```

clone y la interfaz Cloneable

- Al utilizar este mecanismo se redefine

```
1    protected Object clone () throws CloneNotSupportedException
```

- Se suele redefinir de la siguiente forma

```
1    public MiClase clone () throws CloneNotSupportedException
```

- El método creado debe crear una copia base (`super.clone()`) y crear copias de los atributos no inmutables
 - ▶ Esto último puede conseguirse usando también el método `clone` sobre esos atributos

Reflexiones sobre clone y la interfaz Cloneable

- Según la documentación oficial

- ▶ *Creates and returns a copy of this object. The precise meaning of “copy” may depend on the class of the object. The general intent is that, for any object x, the expressions are true:*

```
1  x.clone() != x
2  x.clone().getClass() == x.getClass()
3  x.clone().equals(x)
4  //These are not absolute requirements!!!!
```



- ▶ *By convention, the object returned by this method should be independent of this object (which is being cloned)*

Reflexiones sobre clone y la interfaz Cloneable

- La interfaz Cloneable no define ningún método.
- Esto rompe con el significado habitual de una interfaz en Java

```
1  class Raro implements Cloneable {  
2      // Este código no produce errores  
3  }
```



- En la clase Object se realiza la comprobación de si la clase que originó la llamada a clone implementa la interfaz
- Si no es así se produce una excepción

Reflexiones sobre clone y la interfaz Cloneable

- El hecho de que la interfaz Cloneable no se comporte como el resto de interfaces y que el funcionamiento del sistema de copia de objetos basado en el método clone esté basado en “recomendaciones” ha sido ampliamente criticado por diversos autores
- Otro hecho también muy criticado es el relativo a que al utilizar el método clone no intervienen los constructores en todo el proceso
- Los atributos “final” no son compatibles con el uso de clone

Constructor copia

- También es posible copiar objetos creando un constructor que acepte como parámetro objetos de la misma clase
- Este constructor se encargaría de hacer la copia profunda
- Este esquema presenta algunos problemas cuando se utilizan jerarquías de herencia

Ejemplo

Java: Constructor de copia

```
1 class A {  
2     private int x;  
3     private int y;  
4  
5     public A (int a, int b) {  
6         x = a;  
7         y = b;  
8     }  
9  
10    public A (A a) {  
11        x = a.x;  
12        y = a.y;  
13    }  
14  
15    @Override  
16    public String toString() {  
17        return "(" + Integer.toString(x) + "," + Integer.toString(y) + ")";  
18    }  
19 }
```

Ejemplo

Java: Constructor de copia en una clase que hereda de A

```
1 class B extends A {  
2     private int z;  
3  
4     public B (int a, int b, int c) {  
5         super (a, b);  
6         z = c;  
7     }  
8  
9     public B (B b) {  
10        super (b);  
11        z = b.z;  
12    }  
13  
14    @Override  
15    public String toString() {  
16        return super.toString() + "(" + Integer.toString(z) + ")";  
17    }  
18 }
```

Ejemplo

Java: Usamos las clases anteriores

```
1 A a1 = new A(11, 22);
2 A a2 = new B(111, 222, 333);
3 A a3;
4
5 // El que se ejecute esta línea o la siguiente determina
6 // el constructor copia al que llamar
7
8 // a3 = a1;
9 a3 = a2;
10
11 A a4 = null;
12
13 // ¿ a3 es un A o un B ?
14
15 // a4 = new A(a3);
16 a4 = new B((B) a3);
17
18 System.out.print(a4);
```

Ejemplo

Java: Usamos reflexión para evitar el problema anterior

```
1 import java.lang.reflect.*;
2
3 // Obtenemos el constructor copia
4 Class cls = a3.getClass();
5 Constructor constructor = null ;
6 try {
7     constructor = cls.getDeclaredConstructor(cls);
8 } catch (NoSuchMethodException e) {}
9
10 //Usamos el constructor copia
11 try {
12     a4 = (A) ((constructor == null) ? null : constructor.newInstance (a3));
13 } catch (InstantiationException ex) {
14     Logger.getLogger(Reflexion.class.getName()).log(Level.SEVERE, null, ex);
15 } catch (IllegalAccessException ex) {
16     Logger.getLogger(Reflexion.class.getName()).log(Level.SEVERE, null, ex);
17 } catch (IllegalArgumentException ex) {
18     Logger.getLogger(Reflexion.class.getName()).log(Level.SEVERE, null, ex);
19 } catch (InvocationTargetException ex) {
20     Logger.getLogger(Reflexion.class.getName()).log(Level.SEVERE, null, ex);
21 }
```

Ruby y clone

- En Ruby el método clone de la clase Object también realiza la copia superficial
- Si se desea realizar la copia profunda debe realizarla el programador

Copia por serialización

- En Ruby se puede recurrir a la serialización, deserialización para crear una copia profunda

```
1 b = Marshal.load ( Marshal.dump(a) )
```

- En este proceso el objeto se convierte a una secuencia de bits y después se construye otro a partir de esta secuencia
- Esta última técnica también es aplicable a Java y en ambos casos es poco eficiente, no aplicable en todos los escenarios
- La documentación de Ruby advierte que el uso del método load puede llevar a la ejecución de código remoto

Copia de objetos

→ *Diseño* ←

- Entonces, ¿en todas las asignaciones de parámetros mutables y en todas las devoluciones de atributos mutables debemos realizar copias defensivas?
 - ▶ No tiene porqué. Depende:
 - ★ De dónde vengan los parámetros a asignar
 - ★ A quién se le dé el atributo
 - ★ Del software que se esté desarrollando, etc.
 - ▶ Hay que estudiar cada caso y decidir
 - ▶ Por ejemplo:
 - ★ No es igual diseñar un paquete que solo es usado por nuestro equipo de desarrollo (como el caso del paquete del supuesto de las prácticas)
 - ★ Que diseñar una biblioteca genérica que van a usar terceros (como una biblioteca con clases matemáticas genéricas)
 - ★ En el segundo caso podemos ser “más defensivos” que en el primero

Copia de Objetos

Dpto. Lenguajes y Sistemas Informáticos
Universidad de Granada

Programación y Diseño Orientado a Objetos

(Curso 2023-2024)

Temporary page!

\LaTeX was unable to guess the total number of pages correctly. There was some unprocessed data that should have been added to the next page. This extra page has been added to receive it.

If you rerun the document (without altering it) this surplus page will be removed, because \LaTeX now knows how many pages to expect for this document.