

Construcción de objetos

Prof. Francisco Velasco Anguita

Dpto. Lenguajes y Sistemas Informáticos
Universidad de Granada

Programación y Diseño Orientado a Objetos

Doble Grado en Ingeniería Informática
y Administración y Dirección de Empresas
(Curso 2024-2025)

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

- Saber diseñar e implementar constructores
- Saber cómo crear varios constructores para una misma clase, tanto en Java como en Ruby
- Saber cómo reutilizar código que sea común a varios constructores
- Conocer cómo se libera la memoria ocupada por los objetos cuando dejan de ser útiles

Contenidos

1 Constructores

- Java
- Ruby

2 Constructores de copia

- Java
- Ruby

3 Memoria dinámica y pila

Cuestiones generales

- Antes de usar **los objetos es necesario crearlos**
- La creación implica la **reserva de memoria** y la **inicialización**
- Normalmente el programador no tiene que ocuparse de la reserva de memoria en sí misma, pero sí de la inicialización
- En algunos lenguajes el programador decide el lugar donde se alojará cada objeto (pila o *heap*)

Constructores

- Los lenguajes orientados a objetos suelen disponer de unos **métodos especiales** denominados **constructores**
- A pesar de su nombre, estos métodos **solo se encargan de la inicialización de las instancias**

Se deben inicializar TODOS los atributos de instancia

- No son métodos de instancia y no especifican ningún tipo de retorno
- Existen diferencias importantes a este respecto en los distintos lenguajes de programación orientados a objetos

Clases-plantilla / Clases-objeto

- Con relación a los constructores:
- **Clases-plantilla**
 - ▶ En muchos casos tienen el mismo nombre de la clase
 - ▶ Son invocados automáticamente utilizando la palabra reservada `new`
- **Clases-objeto**
 - ▶ Pueden tener un nombre arbitrario
 - ▶ Suelen ser métodos de clase

Java

- Tienen el mismo nombre que la clase y no devuelven nada (tampoco `void`)
- Los constructores se utilizan únicamente para asegurar la **inicialización de los atributos**
- Al permitir la sobrecarga de métodos, **puede haber varios**, con distintos parámetros
- **Se puede reutilizar** un constructor desde otro constructor
- Si no se crea ningún constructor existe uno por defecto sin parámetros
- Para construir un objeto se antepone la palabra reservada **new** al nombre de la clase

Ejemplos

Java: Constructor básico

```
1 class Point3D {  
2  
3     // Atributos de instancia  
4     private int x;  
5     private int y;  
6     private int z;  
7  
8     Point3D (int a,int b,int c) { // Constructor  
9         // Se inicializan los atributos de instancia , TODOS  
10        x = a;  
11        y = b;  
12        z = c;  
13    }  
14 }
```

Ejemplos

Java: Clase con varios constructores y código común

```
1 class RestrictedPoint3D {
2     private static int LIMITMAX = 100; // Atributos de clase
3     private static int LIMITMIN = 0;
4     private int x; // Atributos de instancia
5     private int y;
6     private int z;
7
8     private int restricToRange (int a) { // Método de instancia
9         int result = Math.max (LIMITMIN, a);
10        result = Math.min (result, LIMITMAX);
11        return result;
12    }
13
14    RestrictedPoint3D (int x, int y, int z) { // Constructor
15        this.x = restricToRange (x);
16        this.y = restricToRange (y);
17        this.z = restricToRange (z);
18        // Debido a la igualdad de nombres,
19        // es necesario usar "this" para referirse a los atributos
20    }
21
22    RestrictedPoint3D (int x, int y) { // Constructor
23        this (x, y, 0); // Se llama al otro constructor
24    }
25 }
```

★ ¿Qué me decís sobre el método max?

Ejemplos

Java: Uso de la clase anterior

```
1 public static void main (String[] args) {  
2     RestrictedPoint3D p1 = new RestrictedPoint3D (-1, 101, -2000);  
3     RestrictedPoint3D p2 = new RestrictedPoint3D (1, 99);  
4     RestrictedPoint3D p3 = new RestrictedPoint3D (50, 51, 52);  
5     RestrictedPoint3D p4 = new RestrictedPoint3D (-2000, 50, 2000);  
6 }
```

- ★ ¿Cuál es el estado de cada punto creado?
- ★ ¿Qué métodos son llamados en cada construcción?

Ruby

- El equivalente al constructor es un método especial llamado `initialize`
- Es un método de instancia privado que es llamado automáticamente por el método de clase `new`
- Se ocupa de la **creación e inicialización de atributos de instancia**
 - ▶ Cualquier método de instancia puede crear atributos de instancia
 - ▶ Lo recomendable es limitar esta labor al método `initialize`
- **No se puede sobrecargar** `initialize` (ni ningún otro método)
 - ▶ Entonces, **¿se pueden tener varios constructores?** Opciones:
 - ★ Creando métodos de clase que cumplan el cometido de los constructores (igual que `new`)
 - ★ Haciendo que `initialize` admita un número variable de parámetros

Ejemplos

Ruby: Ejemplo con un constructor

```
1 class RestrictedPoint3D
2
3   # Atributos de clase
4   @@LIMIT_MAX = 100
5   @@LIMIT_MIN = 0
6
7   private
8   def restric_to_range (a) # método de instancia
9     result = [@@LIMIT_MIN, a].max
10    result = [@@LIMIT_MAX, result].min
11    result
12  end
13
14  def initialize (x, y, z) # creación e inicialización de atributos de instancia
15    @x = restric_to_range (x)
16    @y = restric_to_range (y)
17    @z = restric_to_range (z)
18  end
19 end
20
21 puts RestrictedPoint3D.new(-1,1,1).inspect
```

★ ¿Hay algún conflicto de nombres en las líneas 15, 16, ó 17?

Ejemplos

Ruby: Ejemplo con dos constructores

```
1 class RestrictedPoint3D
2
3   # Añadimos al código anterior
4
5   def self.new_3D(x,y,z) # método de clase
6     new(x,y,z)
7   end
8
9   def self.new_2D(x,y)   # método de clase
10    new(x,y,0)
11  end
12
13  private_class_method :new # pasa a ser privado
14 end
15
16 puts RestrictedPoint3D.new_3D(-1,101,-2000).inspect
17 puts RestrictedPoint3D.new_2D(1,99).inspect
18 puts RestrictedPoint3D.new_3D(50,51,52).inspect
19 puts RestrictedPoint3D.new_3D(-2000,50,2000).inspect
20 # puts RestrictedPoint3D.new(-1,1,1).inspect # Error, new es ahora privado
```

¡Mal ejemplo!

Ruby: Error frecuentemente cometido por los estudiantes

```
1 class RestrictedPoint3D
2
3   # Forma ERRÓNEA de implementar estos constructores
4
5   def self.new_3D(x,y,z) # método de clase
6     @x = restric_to_range (x)
7     @y = restric_to_range (y)
8     @z = restric_to_range (z)
9   end
10
11   def self.new_2D(x,y) # método de clase
12     @x = restric_to_range (x)
13     @y = restric_to_range (y)
14     @z = 0
15   end
16
17   private_class_method :new # pasa a ser privado
18 end
```

★ ¿Cuáles son los errores? ¿Por qué son errores?

Estos errores, en los exámenes, serán **penalizados**

Ejemplos

Ruby: initialize con un número variable de parámetros

```
1 def initialize (x, y, *z)
2   # *z es un array con el resto de parámetros que se pasen
3
4   @x = restric_to_range (x)
5   @y = restric_to_range (y)
6   if (z.size != 0) then
7     z_param = z[0]
8   else
9     z_param = 0
10  end
11  @z = restric_to_range (z_param)
12 end
13
14 # En algún lugar fuera de la clase ...
15
16 puts RestrictedPoint3D.new(1,2,3,4,5,6).inspect
17
18 # los parámetros extra son ignorados
```

Ejemplos

Ruby: initialize con valores por defecto

```
1 def initialize (x, y, z=0)
2   # el parámetro z tiene un valor por defecto
3
4   @x=restric_to_range(x)
5   @y=restric_to_range(y)
6   @z=restric_to_range(z)
7 end
8
9 # En algún lugar fuera de la clase ...
10
11 puts RestrictedPoint3D.new(1,2).inspect
12 puts RestrictedPoint3D.new(1,2,3).inspect
```

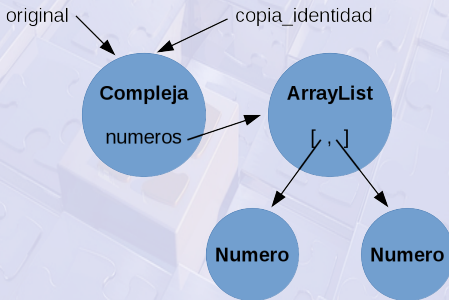
Ejemplos

Ruby: Parámetros nombrados con valores por defecto

```
1 # Parámetros nombrados con valores por defecto
2
3 def initialize (x:, y:, z:0)
4   @x = restric_to_range (x)
5   @y = restric_to_range (y)
6   @z = restric_to_range (z)
7 end
8
9 # En algún lugar fuera de la clase ...
10
11 puts RestrictedPoint3D.new(x:-1, y:101, z:-2000).inspect
12
13 # Puedo cambiar el orden
14 puts RestrictedPoint3D.new(y:2, z:3, x:1).inspect
15
16 puts RestrictedPoint3D.new(x:1, y:99).inspect
```

Constructor de copia

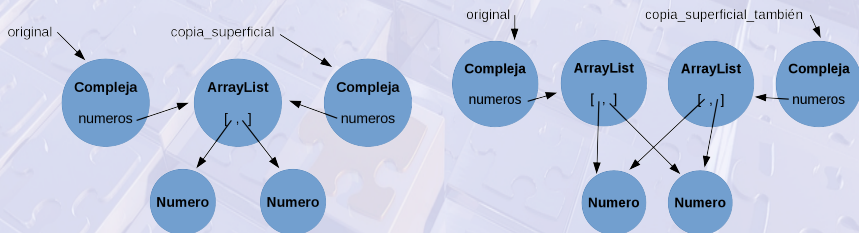
- Un constructor de copia recibe un objeto como parámetro
 - ▶ Construye otro objeto (distinta identidad)
 - ▶ Con el mismo estado (inicialmente) que el objeto recibido
- Puede haber diferentes niveles de copia (superficial y profunda)
 - ▶ Ya conocemos cómo actúa la asignación `copia = original`



Constructor de copia

Copia superficial

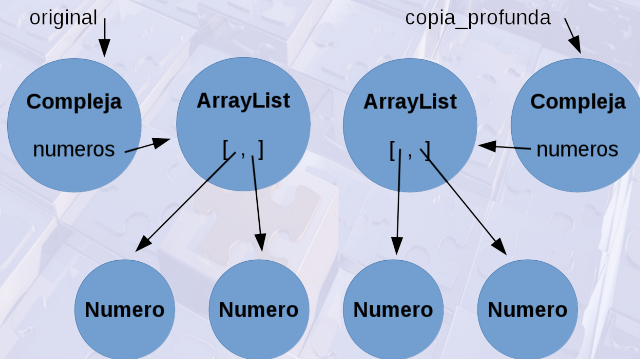
- En una copia superficial, aunque se crea un objeto distinto, en algún nivel se referencia un mismo objeto



Constructor de copia

Copia profunda

- En una copia profunda, se copia toda la jerarquía



Constructor de copia en Java

Java: Constructor de copia (superficial)

```
1 class Persona {  
2     private String nombre;  
3     // otros atributos  
4  
5     public Persona (String unNombre) {  
6         nombre = unNombre;  
7         // se inicializan el resto de atributos  
8     }  
9  
10    public Persona (Persona otraPersona) {  
11        nombre = otraPersona.nombre;  
12        // se asignan el resto de atributos  
13        // tomando los valores de los atributos del parámetro  
14    }  
15 }
```


Constructor de copia en Ruby

Ruby: Constructor de copia (superficial)

```
1 class Persona
2   attr_reader :nombre
3
4   # A los atributos que no tengan consultor básico hay que añadirsele
5
6   attr_reader :otro_atributo
7
8   def initialize (nombre, otro_atributo)
9     @nombre = nombre
10    @otro_atributo = otro_atributo
11  end
12
13  def self.constructor_copia (persona)
14    new(persona.nombre, persona.otro_atributo)
15  end
16 end
```

Memoria dinámica y pila

- En Java y Ruby todos los objetos **se crean** en **memoria dinámica** (*heap*)
- En ambos lenguajes las variables contienen referencias a objetos (punteros)
 - ▶ Hay algunas excepciones como los tipos primitivos de Java (`int`, `float`, etc.)
 - ▶ Los `String` también tienen un tratamiento distinto
- Cuando se devuelve el valor de una variable, **se está devolviendo una referencia a un objeto**
- ¿Cómo **se libera** la memoria?
 - ▶ Java y Ruby disponen de un **recolector de basura** que libera automáticamente la memoria utilizada por objetos no referenciados

Memoria dinámica y pila:

El lenguaje C++

- En C++, el programador puede decidir si crea los objetos en la pila o en el *heap*
- También es el responsable de la liberación de la memoria reservada en el *heap* para un objeto

C++: Destructor

```

1 class A {
2 };
3
4 class B {
5     private:
6         A *atributo;
7
8     public:
9         B() {
10             atributo = new A();
11         }
12
13         ~B() {
14             // destructor
15             delete (atributo);
16         }
17 };

```

C++: Pila y Heap

```

1 void unaFuncion () {
2     A a;           // En la pila
3     B *b = new B(); // En el heap
4
5     // . . .
6
7     delete (b);    // se libera del heap
8     // al salir se libera la pila
9 }
10
11 int main () {
12     unaFuncion();
13 }

```

Constructores

→ *Diseño* ←



- Los constructores no cuestan dinero



- El tiempo perdido entendiendo un código enrevesado, sí
- Con los constructores, y en general, con cualquier método,
 - ▶ Sobrecargarlos (si el lenguaje lo permite) de manera **que cada constructor/método haga una cosa muy concreta**
 - ▶ Si el lenguaje no admite sobrecarga, añadir constructores/métodos con distintos nombres
 - ▶ Si en un constructor/método, se debe hacer un procesamiento diferente según el número y tipo de los parámetros recibidos, tal vez haya que sobrecargarlo (o crear más constructores/métodos)
- **Los diseños e implementaciones simples son fáciles de mantener**

Construcción de objetos

Prof. Francisco Velasco Anguita

Dpto. Lenguajes y Sistemas Informáticos
Universidad de Granada

Programación y Diseño Orientado a Objetos

Doble Grado en Ingeniería Informática
y Administración y Dirección de Empresas
(Curso 2024-2025)