

Programación orientada a objetos en Java

Ricardo Pérez López

IES Doñana, curso 2020/2021



Generado el 15 de febrero de 2021 a las 11:52:00

1. Uso básico de objetos
2. Clases y objetos básicos en Java

1. Uso básico de objetos

1.1 Instanciación

1.2 Referencias

1.3 Comparación de objetos

1.4 Destrucción de objetos y recolección de basura

1.1. Instanciación

1.1.1 `new`

1.1.2 `getClass()`

1.1.3 `instanceof`

new

- ▶ La operación `new` permite instanciar un objeto a partir de una clase.

getClass()

- ▶ El método `getClass()` devuelve la clase de la que es instancia el objeto sobre el que se ejecuta.
- ▶ Lo que devuelve es una instancia de la clase `java.class.Class`.
- ▶ Para obtener una cadena con el nombre de la clase, se puede usar el método `getSimpleName()` definido en la clase `Class`:

```
jshell> String s = "Hola";  
s ==> "Hola"  
  
jshell> s.getClass()  
$2 ==> class java.lang.String  
  
jshell> s.getClass().getSimpleName()  
$3 ==> "String"
```

instanceof

- ▶ El operador **instanceof** permite comprobar si un objeto es instancia de una determinada clase.
- ▶ Por ejemplo:

```
jshell> "Hola" instanceof String  
$1 ==> true
```

- ▶ Sólo se puede aplicar a referencias, no a valores primitivos:

```
jshell> 4 instanceof String  
| Error:  
| unexpected type  
|   required: reference  
|   found:    int  
| 4 instanceof String  
| ^
```

1.2. Referencias

1.2.1 `null`

Referencias

- ▶ Los objetos son accesibles a través de **referencias**.
- ▶ Las referencias se pueden almacenar en variables de **tipo referencia**.
- ▶ Por ejemplo, `String` es una clase, y por tanto es un tipo referencia. Al hacer la siguiente declaración:

```
String s;
```

estamos declarando `s` como una variable que puede contener una referencia a un valor de tipo `String`.

null

- ▶ El tipo `null` sólo tiene un valor: la referencia nula, representada por el literal `null`.
- ▶ El tipo `null` es compatible con cualquier tipo referencia.
- ▶ Por tanto, una variable de tipo referencia siempre puede contener la referencia nula.
- ▶ En la declaración anterior:

```
String s;
```

la variable `s` puede contener una referencia a un objeto de la clase `String`, o bien puede contener la referencia nula `null`.

- ▶ La referencia nula sirve para indicar que la variable no apunta a ningún objeto.

1.3. Comparación de objetos

1.3.1 `equals`

1.3.2 `compareTo`

1.3.3 `hashCode`

Comparación de objetos

- ▶ El operador `==` aplicado a dos objetos (valores de tipo referencia) devuelve `true` si ambos son **el mismo objeto**.
- ▶ Es decir: el operador `==` compara la **identidad** de los objetos para preguntarse si son **idénticos**.
- ▶ Equivale al operador `is` de Python.
- ▶ Para usar un mecanismo más sofisticado que realmente pregunte si dos objetos son **iguales**, hay que usar el método `equals`.

equals

- ▶ El método `equals` compara dos objetos para comprobar si son iguales.
- ▶ Debería usarse siempre en sustitución del operador `==`, que sólo comprueba si son idénticos.
- ▶ Equivale al `__eq__` de Python, pero en Java hay que llamarlo explícitamente (no se llama implícitamente al usar `==`).

```
jshell> String s = new String("Hola");  
s ==> "Hola"  
  
jshell> String w = new String("Hola");  
w ==> "Hola"  
  
jshell> s == w  
$3 ==> false  
  
jshell> s.equals(w)  
$4 ==> true
```

- ▶ La implementación predeterminada del método `equals` se hereda de la clase `Object` (que ya sabemos que es la clase raíz de la jerarquía de clases en Java, por lo que toda clase acaba siendo subclase, directa o indirecta, de `Object`).
- ▶ En dicha implementación predeterminada, `equals` equivale a `==`:

```
public boolean equals(Object otro) {  
    return this == otro;  
}
```

- ▶ Por ello, es importante sobrescribir dicho método al crear nuevas clases, ya que, de lo contrario, se comportaría igual que `==`.

compareTo

- Un método parecido es `compareTo`, que compara dos objetos de forma que la expresión `a.compareTo(b)` devuelve un entero:
 - `-1` si `a < b`.
 - `0` si `a == b`.
 - `1` si `a > b`.

hashCode

- ▶ El método `hashCode` equivale al `__hash__` de Python.
- ▶ Como en Python, devuelve un número entero (en este caso, de 32 bits) asociado a cada objeto, de forma que si dos objetos son iguales, deben tener el mismo valor de `hashCode`.
- ▶ Por eso (al igual que ocurre en Python), el método `hashCode` debe coordinarse con el método `equals`.
- ▶ A diferencia de lo que ocurre en Python, en Java **todos los objetos son hashables**. De hecho, no existe el concepto de *hashable* en Java, ya que no tiene sentido.
- ▶ Este método se usa para acelerar la velocidad de almacenamiento y recuperación de objetos en determinadas colecciones como `HashMap`, `HashSet` o `Hashtable`.

- ▶ La implementación predeterminada de `hashCode` se hereda de la clase `Object`, y devuelve un valor que depende de la posición de memoria donde está almacenado el objeto.
- ▶ Al crear nuevas clases, es importante sobrescribir dicho método para que esté en consonancia con el método `equals` y garantizar que siempre se cumple que:

Si `x.equals(y)`, entonces `x.hashCode() == y.hashCode()`.

```
jshell> "Hola".hashCode()  
$1 ==> 2255068
```

1.4. Destrucción de objetos y recolección de basura

Destrucción de objetos y recolección de basura

- Los objetos en Java no se destruyen explícitamente, sino que se marcan para ser eliminados cuando no hay ninguna referencia apuntándole:

```
jshell> String s = "Hola"; // Se crea el objeto y una referencia se guarda en «s»  
s ==> "Hola"  
  
jshell> s = null;           // Ya no hay más referencias al objeto, así que se marca  
s ==> null
```

- La próxima vez que se active el recolector de basura, el objeto se eliminará de la memoria.

2. Clases y objetos básicos en Java

2.1 Clases envolventes (*wrapper*)

2.2 Cadenas

2.3 Arrays

2.1. Clases envolventes (*wrapper*)

2.1.1 *Boxing y Unboxing*

Clases envolventes (*wrapper*)

- ▶ Las **clases envolventes** (también llamadas **clases *wrapper***) son clases cuyas instancias representan valores primitivos almacenados dentro de valores referencia.
- ▶ Esos valores referencia *envuelven* al valor primitivo dentro de un objeto.
- ▶ Se utilizan en contextos en los que se necesita manipular un dato primitivo como si fuera un objeto, de una forma sencilla y transparente.

- Existe una clase *wrapper* para cada tipo primitivo:

| Clase <i>wrapper</i> | Tipo primitivo |
|----------------------|---------------------|
| Boolean | <code>bool</code> |
| Byte | <code>byte</code> |
| Short | <code>short</code> |
| Character | <code>char</code> |
| Integer | <code>int</code> |
| Long | <code>long</code> |
| Float | <code>float</code> |
| Double | <code>double</code> |

- Los objetos de estas clases disponen de métodos para acceder a los valores envueltos dentro del objeto.

► Por ejemplo:

```
jshell> Integer x = new Integer(4);  
x ==> 4  
  
jshell> x.floatValue()  
$2 ==> 4.0  
  
jshell> Boolean y = new Boolean(true);  
y ==> true  
  
jshell> y.shortValue()  
| Error:  
| cannot find symbol  
|   symbol:   method shortValue()  
|   y.shortValue()  
|   ^-----^
```


2.2. Cadenas

2.2.1 Inmutables

2.2.2 Mutables

2.2.3 Conversión a `String`

2.2.4 Concatenación de cadenas

2.2.5 Comparación de cadenas

2.2.6 Diferencias entre literales cadena y objetos `String`

2.3. Arrays

2.3.1 De tipos primitivos

2.3.2 `.length`

2.3.3 De objetos

2.3.4 Subtipado entre *arrays*

2.3.5 `java.util.Arrays`

2.3.6 Copia y redimensionado de arrays

2.3.7 Comparación de *arrays*

2.3.8 Arrays multidimensionales

Arrays.deepEquals()

Gosling, James, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley. 2014. *The Java® Language Specification*. Java SE 8 edition. Upper Saddle River, NJ: Addison-Wesley.