

Programación orientada a objetos en Java

Ricardo Pérez López

IES Doñana, curso 2020/2021



Generado el 21 de febrero de 2021 a las 19:46:00

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*

2.1.2 *Autoboxing* y *autounboxing*

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
<code>java.lang.Boolean</code>	<code>bool</code>
<code>java.lang.Byte</code>	<code>byte</code>
<code>java.lang.Short</code>	<code>short</code>
<code>java.lang.Character</code>	<code>char</code>
<code>java.lang.Integer</code>	<code>int</code>
<code>java.lang.Long</code>	<code>long</code>
<code>java.lang.Float</code>	<code>float</code>
<code>java.lang.Double</code>	<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()  
|   ^-----^
```


- ▶ A partir de JDK 9, los constructores *wrapper* de tipo han quedado obsoletos.
- ▶ Actualmente, se recomienda que usar uno de los métodos `valueOf` para obtener un objeto *wrapper*.
- ▶ El método es un miembro estático de todas las clases *wrappers* y todas las clases numéricas admiten formas que convierten un valor numérico o una cadena en un objeto.
- ▶ Por ejemplo:

```
jshell> Integer i = Integer.valueOf(100);  
i ==> 100
```

Boxing y unboxing

- El **boxing** es el proceso de *envolver* un valor primitivo en una referencia a una instancia de su correspondiente clase *wrapper*. Por ejemplo:

```
jshell> Integer x = new Integer(4);  
x ==> 4  
  
jshell> x.getClass()  
$2 ==> class java.lang.Integer
```

- El **unboxing** es el proceso de extraer un valor primitivo a partir de una instancia de su correspondiente clase *wrapper*. Por ejemplo:

```
jshell> Integer i = Integer.valueOf(100);  
i ==> 100  
  
jshell> int j = i.intValue();  
j ==> 100
```

- A partir de JDK 5, este proceso se puede llevar a cabo automáticamente mediante el **autoboxing** y el **autounboxing**.

Autoboxing y autounboxing

- El **autoboxing** es el mecanismo que convierte automáticamente un valor primitivo en una referencia a una instancia de su correspondiente clase *wrapper*. Por ejemplo:

```
jshell> Integer x = 4;  
x ==> 4  
  
jshell> x.getClass()  
$2 ==> class java.lang.Integer
```

- El **autounboxing** es el mecanismo que convierte automáticamente una instancia de una clase *wrapper* en su valor primitivo equivalente. Por ejemplo:

```
public class Prueba {  
    public static void main(String[] args) {  
        Integer i = new Integer(4);  
        int res = cuadrado(i);    // Se envía un Integer  
        System.out.println(res);  
    }  
    public static cuadrado(int x) { // Se recibe un int  
        return x ** x;  
    }  
}
```

2.2. Cadenas

2.2.1 Inmutables (`String`)

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`

Cadenas

- ▶ En Java, las cadenas son objetos.
- ▶ Por tanto, son valores referencia, instancias de una determinada clase.
- ▶ Existen dos tipos de cadenas:
 - Inmutables: instancias de la clase `String`.
 - Mutables: instancias de las clases `StringBuffer` o `StringBuilder`.

Inmutables (String)

- ▶ Los objetos de la clase `String` son cadenas inmutables.
- ▶ Las cadenas literales (secuencias de caracteres encerradas entre dobles comillas `"`) son instancias de la clase `String`:

```
jshell> String s = "Hola";
```

- ▶ Otra forma de crear un objeto de la clase `String` es instanciando dicha clase y pasándole otra cadena al constructor. De esta forma, se creará un nuevo objeto cadena con los mismos caracteres que la otra cadena:

```
jshell> String s = new String("Hola");
```

- Si se usa varias veces el mismo literal cadena, el JRE intenta aprovechar el objeto ya creado y no crea uno nuevo:

```
jshell> String s = "Hola";  
s ==> "Hola"  
  
jshell> String w = "Hola";  
w ==> "Hola"  
  
jshell> s == w  
$3 ==> true
```

- Las cadenas creadas mediante instanciación, siempre son objetos distintos:

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

- Pregunta: ¿cuántos objetos cadena se crean en cada caso?

- ▶ Los objetos de la clase `String` disponen de métodos que permiten realizar operaciones con cadenas.
- ▶ Muchos de ellos devuelven una nueva cadena a partir de la original tras una determinada transformación.
- ▶ Algunos métodos interesantes son:
 - `length`
 - `indexOf`
 - `lastIndexOf`
 - `charAt`
 - `repeat`
 - `replace`
 - `startsWith`
 - `endsWith`
 - `substring`
 - `toUpperCase`
 - `toLowerCase`

- ▶ La clase `String` también dispone de **métodos estáticos**.
- ▶ El más interesante es `valueOf`, que devuelve la representación en forma de cadena de su argumento:

```
jshell> String.valueOf(4)
$1 ==> "4"

jshell> String.valueOf(2.3)
$2 ==> "2.3"

jshell> String.valueOf('a')
$3 ==> "a"
```

- ▶ No olvidemos que, en Java, los caracteres y las cadenas son tipos distintos:
 - Un carácter es un valor primitivo de tipo `char` y sus literales se representan entre comillas simples (`'a'`).
 - Una cadena es un valor referencia de tipo `String` y sus literales se representan entre comillas dobles (`"a"`).

Mutables

- ▶ Un objeto de la clase `String` no puede modificarse una vez creado.
- ▶ Es exactamente lo que ocurre con las cadenas en Python.
- ▶ En Java existen **cadenas mutables** que sí permiten su modificación después de haberse creado.
- ▶ Para ello, proporciona dos clases llamadas `StringBuffer` y `StringBuilder`, cuyas instancias son cadenas mutables.
- ▶ Las dos funcionan prácticamente de la misma forma, con la única diferencia de que los objetos `StringBuffer` permiten sincronización entre hilos mientras que los `StringBuilder` no.
- ▶ Cuando se está ejecutando un único hilo, es preferible usar objetos `StringBuilder` ya que son más eficientes.

- Se puede crear un objeto `StringBuilder` vacío o a partir de una cadena:

```
jshell> StringBuilder sb = new StringBuilder();           // Crea uno vacío
sb ==>

jshell> StringBuilder sb = new StringBuilder("Hola"); // O a partir de una cadena
sb ==> "Hola"
```

StringTokenizer

- ▶ La clase `StringTokenizer` permite romper una cadena en *tokens*.
- ▶ El método de *tokenización* consiste en buscar los elementos separados por delimitadores, que son los caracteres que separan los *tokens*.
- ▶ Esos delimitadores pueden especificarse en el momento de crear el *tokenizador* o bien *token* a *token*.
- ▶ Por ejemplo:

```
StringTokenizer st = new StringTokenizer("esto es una prueba");  
while (st.hasMoreTokens()) {  
    System.out.println(st.nextToken());  
}
```

produce la siguiente salida:

```
esto  
es  
una  
prueba
```

- ▶ La clase `StringTokenizer` se mantiene por compatibilidad pero su uso no se recomienda en código nuevo.
- ▶ En su lugar, se recomienda usar el método `split` de la clase `String` o el paquete `java.util.regex`.
- ▶ Por ejemplo:

```
String[] result = "esto es una prueba".split("\\s");  
for (int x = 0; x < result.length; x++)  
    System.out.println(result[x]);
```

Conversión a `String`

- ▶ La conversión de un objeto a `String` se realiza llamando al método `toString` del objeto.
- ▶ Todo objeto, sea de la clase que sea, tiene un método `toString` heredado de la clase `Object` y posiblemente sobrescribiendo éste.
- ▶ Si es un valor primitivo, primero se convierte a instancia de su clase *wrapper* correspondiente.

Concatenación de cadenas

- La operación de concatenación de cadenas se realiza con el operador `+`:

```
jshell> "hola " + "mundo"  
$1 ==> "hola mundo"
```

- También existe el método `concat`, que hace lo mismo:

```
jshell> "hola ".concat("mundo")  
$1 ==> "hola mundo"
```

Comparación de cadenas

- En las cadenas, las comparaciones se pueden realizar:
 - Con el operador `==`:

```
jshell> "hola" == "hola"  
true
```

No es conveniente, ya que comprueba si los dos objetos son el mismo.

- Con el método `equals`:

```
jshell> "hola".equals("hola")  
true
```

Comprueba si las dos cadenas tienen los mismos caracteres.

- Con el método `compareTo`:

```
jshell> "hola".compareTo("adiós")  
7
```


Diferencias entre literales cadena y objetos `String`

- ▶ Los literales cadena se almacenan en un *pool* de cadenas y se reutilizan siempre que se puede.
- ▶ Los objetos `String` van asociados a un literal cadena almacenado en el *pool*.
- ▶ Se puede acceder a ese literal del objeto cadena usando el método `intern`:

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

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.