Programación orientada a objetos en Java

Ricardo Pérez López

IES Doñana, curso 2020/2021

Generado el 22 de febrero de 2021 a las 13:13:00

Índice general

1. Uso básico de objetos

1.1. Instanciación

1.1.1. `new`

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

1.1.2. `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"
```

1.1.3. `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

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`.

1.2.1. `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

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`.

1.3.1. `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 sobreescribir dicho método al crear nuevas clases, ya que, de lo contrario, se comportaría igual que `==`.

1.3.2. `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`.

1.3.3. `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 sobreescribir 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

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*)

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 *wrapper*	Tipo primitivo
`java.lang.Boolean`	`bool`
`java.lang.Byte`	`byte`
`java.lang.Short`	`short`
`java.lang.Character`	`char`
`java.lang.Integer`	`int`
`java.lang.Long`	`long`
`java.lang.Float`	`float`
`java.lang.Double`	`double`

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
```

2.1.1. *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***.

2.1.2. *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

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`.

2.2.1. 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"`).

2.2.2. 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"
```

2.2.2.1. `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]);
```

2.2.3. 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 sobreescribiéndo éste.

Si es un valor primitivo, primero se convierte a instancia de su clase *wrapper* correspondiente.

2.2.4. 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"
```

2.2.5. 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
  ```

2.2.6. 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
```

3. *Arrays*

3.1. Definición

En Java, un ***array*** es un dato mutable compuesto por elementos (también llamados **componentes**) a los que se accede mediante ***indexación***, es decir, indicando la posición donde se encuentra almacenado el elemento deseado dentro del *array*.

Se parece a las **listas** de Python, con las siguientes diferencias:

- Cada *array* tiene una **longitud fija** (no puede crecer o encogerse de tamaño dinámicamente).
- Todos los elementos de un *array* deben ser del **mismo tipo**, el cual debe indicarse en la declaración del *array*.

Los *arrays* en Java pueden contener valores primitivos o referencias a objetos.

Los *arrays* de Java son **objetos** y, por tanto, son valores referencia.

Los *arrays* se declaran indicando el tipo del elemento que contienen, seguido de `[]`.

Por ejemplo, para declarar un *array* de enteros, se puede hacer:

```
int[] x;
```

Ahora mismo, `x` es una referencia a un objeto *array* que puede contener elementos de tipo `int`. Como la variable `x` aún no ha sido inicializada, el valor que contiene es la referencia nula (`null`):

```
jshell> int[] x;
x ==> null
```

Por tanto, `x` puede hacer referencia a un *array* de enteros, pero actualmente no hace referencia a ninguno.

A esa variable le podemos asignar una referencia a un objeto *array* del tipo adecuado.

Para ello, se puede crear un objeto *array* usando el operador **`new`** e indicando el tipo de los elementos y la longitud del *array* (entre corchetes):

```
jshell> x = new int[5];
x ==> int[5] { 0, 0, 0, 0, 0 }
```

A partir de este momento, la variable `x` contiene una referencia a un objeto *array* de cinco elementos de tipo `int` que, ahora mismo, tienen todos el valor `0`.

Como se puede observar, los elementos de un *array* siempre se inicializan a un **valor por defecto** cuando se crea el *array* (`0` en enteros, `0.0` en reales, **`false`** en *booleanos*, `'\000'` en caracteres y **`null`** en valores referencia).

También se pueden inicializar los elementos de un *array* en el momento en que se crea con **`new`**.

En ese caso:

- Se indican los valores de los elementos del *array* entre llaves.

- No se indica la longitud del array, ya que se deduce a partir de la lista de valores iniciales.

Por ejemplo:

```
jshell> int[] x = new int[] {6, 5, 27, 81};
x ==> int[4] { 6, 5, 27, 81 }
```

Para **acceder** a un elemento del *array* se usa el operador de *indexación* (los corchetes):

```
jshell> int[] x = new int[] {6, 5, 27, 81};
x ==> int[4] { 6, 5, 27, 81 }

jshell> x[3]
$2 ==> 81
```

Los elementos se indexan de 0 a $n - 1$, siendo n la longitud del *array*.

Si se intenta acceder a un elemento fuera de esos límites, se levanta una excepción `java.lang.ArrayIndexOutOfBound`

```
jshell> x[4]
|  Exception java.lang.ArrayIndexOutOfBoundsException: Index 4 out of bounds for length 4
|        at (#3:1)
```

Para conocer la longitud de un *array*, se puede consultar el atributo `length`:

```
jshell> x.length
4
```

Ese valor es constante y no se puede cambiar:

```
jshell> x.length = 44
|  Error:
|  cannot assign a value to final variable length
|  x.length = 43
|  ^------^
```

Para cambiar un elemento del *array* por otro, se puede usar la *indexación* combinada con la *asignación*:

```
jshell> x[2] = 99;
$5 ==> 99

jshell> x
x ==> int[4] { 6, 5, 99, 81 }
```

El compilador comprueba que el valor a asignar es del tipo correcto, e impide la operación si se ve obligado a hacer un *narrowing* para hacer que el tipo del valor sea compatible con el tipo del elemento:

```
shell> x[2] = 99.9;
|  Error:
|  incompatible types: possible lossy conversion from double to int
```

```
|  x[2] = 99.9;
|         ^--^
```

Los elementos de un *array* también pueden ser valores referencia.

En ese caso, sus elementos serán objetos de una determinada clase.

Por el principio de sustitución, esos objetos también podrán ser instancias de una de sus subclases.

Inicialmente, los elementos referencia de un *array* toman el valor **`null`**.

Por ejemplo:

```
jshell> Trabajador[] t = new Trabajador[5];
t ==> Trabajador[5] { null, null, null, null, null }
```

En cada elemento de `t` podremos meter una instancia de la clase `Trabajador` o de cualquier subclase suya:

```
jshell> t[2] = new Docente(...)
$8 ==> Docente@1b701da1

jshell> t
t ==> Trabajador[5] { null, null, Docente@1b701da1, null, null }
```

Si declaramos un *array* de tipo `Object[]`, estamos diciendo que sus elementos pueden ser de cualquier tipo referencia, lo que tiene ventajas e inconvenientes:

- Ventaja: los elementos del *array* podrán ser de cualquier tipo, incluyendo tipos primitivos (recordemos el *boxing/unboxing*).
- Inconveniente: no podremos aprovechar el comprobador de tipos del compilador para determinar si los tipos son los adecuados, por lo que tendremos que hacerlo a mano en tiempo de ejecución.

3.2. De tipos primitivos

3.2.1. Declaración

3.2.2. Creación

3.2.3. Inicialización

3.3. `.length`

3.4. De objetos

3.4.1. Declaración

3.4.2. Creación

3.4.3. Inicialización

Bibliografía

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.