

Programación orientada a objetos en Java

Ricardo Pérez López

IES Doñana, curso 2020/2021



Generado el 13 de abril de 2021 a las 13:03:00

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

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.
- ▶ Hay que indicar el nombre de la clase y pasarle al constructor los argumentos que necesite, entre paréntesis y separados por comas. Los paréntesis son obligatorios aunque no haya argumentos.
- ▶ Por ejemplo, si tenemos una clase **Triángulo** cuyo constructor espera dos argumentos (*ancho* y *alto*), podemos crear una instancia de esa clase de la siguiente forma:

```
jshell> new Triangulo(20, 30);  
$1 ==> Triangulo@ee7d9f1  
  
jshell> Triangulo t = new Triangulo(4, 2);  
t ==> Triangulo@726f3b58
```

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.lang.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.

- Al intentar invocar a un método desde una referencia nula, se lanza una excepción `NullPointerException`:

```
jshell> String s;  
s ==> null  
  
jshell> s.concat("hola")  
| Exception java.lang.NullPointerException  
|       at (#2:1)
```

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:
 - Menor que cero si `a < b`.
 - 0 si `a == b`.
 - Mayor que cero 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 Cadenas

2.2 Clases envolventes (*wrapper*)

2.1. Cadenas

2.1.1 Inmutables

2.1.2 Mutables

2.1.3 Conversión a `String`

2.1.4 Concatenación de cadenas

2.1.5 Comparación de cadenas

2.1.6 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

- ▶ Las cadenas inmutables son objetos de la clase `String`.
- ▶ 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]);
```

- ▶ Los métodos definidos en la clase `String` se pueden consultar en la API de Java:
<https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/lang/String.html>

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
```

2.2. Clases envolventes (*wrapper*)

2.2.1 *Boxing* y *unboxing*

2.2.2 *Autoboxing* y *autounboxing*

2.2.3 La clase `Number`

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>boolean</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 de las clases *wrapper* han quedado obsoletos.
- ▶ Actualmente, se recomienda usar uno de los métodos estáticos `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 = Integer.valueOf(4);  
        int res = 2 * i;           // El Integer se convierte en int  
        System.out.println(res);  
    }  
}
```

La clase `Number`

- ▶ La clase `java.lang.Number` en Java es una clase abstracta que representa la clase base (o superclase) de todas las clases que representan valores numéricos convertibles a valores primitivos de tipo `byte`, `double`, `float`, `int`, `long` y `short`.
- ▶ Esta clase es la superclase de todas las clases *wrapper* que representan valores numéricos: `Byte`, `Double`, `Float`, `Integer`, `Long` y `Short`.
- ▶ También es la superclase de las clases `java.math.BigInteger` y `java.math.BigDecimal` cuyas instancias representan, respectivamente, números enteros y reales de precisión arbitraria.

- ▶ La clase abstracta `Number` declara (o define) los siguientes métodos:

```
byte byteValue()  
abstract double doubleValue()  
abstract float floatValue()  
abstract int intValue()  
abstract long longValue()  
short shortValue()
```

- ▶ Los métodos `byteValue` y `shortValue` son concretos porque devuelven el valor de `intValue()` convertido, respectivamente, a `byte` o a `short` a través de un *casting* (`byte`) o (`short`).
- ▶ Los métodos abstractos se implementan luego en sus subclases.

► Ejemplo de uso:

```
jshell> Number n = Integer.valueOf(4);  
n ==> 4  
  
jshell> n.intValue()  
$2 ==> 4  
  
jshell> n.longValue()  
$3 ==> 4  
  
jshell> Number bi = new BigInteger("2318972398472987492874982234742");  
bi ==> 2318972398472987492874982234742  
  
jshell> bi.intValue()  
$5 ==> 1756076662           // No cabe en un int, así que se trunca
```

3. Arrays

- 3.1 Definición
- 3.2 Declaración
- 3.3 Creación
- 3.4 Inicialización
- 3.5 Acceso a elementos
- 3.6 Longitud de un *array*
- 3.7 Modificación de elementos
- 3.8 *Arrays* de tipos referencia
- 3.9 Subtipado entre *arrays*
- 3.10 `java.util.Arrays`
- 3.11 Copia y redimensionado de *arrays*
- 3.12 Comparación de *arrays*
- 3.13 *Arrays* multidimensionales

3.1. Definición

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.

3.2. Declaración

Declaración

- ▶ 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.

3.3. Creación

Creación

- ▶ 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).

3.4. Inicialización

Inicialización

- ▶ 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 }
```

3.5. Acceso a elementos

Acceso a elementos

- 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.ArrayIndexOutOfBoundsException`:

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


3.6. Longitud de un *array*

Longitud de un array

- 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  
| ^-----^
```

3.7. Modificación de elementos

Modificación de elementos

- 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;  
|      ^__^
```

3.8. Arrays de tipos referencia

Arrays de tipos referencia

- ▶ Los elementos de un *array* también pueden ser valores referencia.
- ▶ En ese caso, sus elementos serán objetos de una determinada clase.
- ▶ Inicialmente, los elementos referencia del *array* toman el valor `null`.
- ▶ Por ejemplo:

```
jshell> String[] cadenas = new String[5];  
cadenas ==> String[5] { null, null, null, null, null }
```

- ▶ En cada elemento de `cadenas` podremos meter una instancia de la clase `String`:

```
jshell> cadenas[2] = "hola";  
$2 ==> "hola"  
  
jshell> cadenas  
cadenas ==> String[5] { null, null, "hola", null, null }
```

- También podemos inicializar el *array* con objetos:

```
jshell> String[] cadenas = new String[] { "hola", "adiós", "prueba" };
cadenas ==> String[3] { "hola", "adiós", "prueba" }

jshell> cadenas[1]
$2 ==> "adiós"

jshell> cadenas.length
$3 ==> 3
```

- ▶ Hemos dicho que los elementos de un *array* de tipos referencia deben ser objetos de una determinada clase, que es la clase indicada al declarar el *array*.
- ▶ Pero por el principio de sustitución, esos elementos también pueden ser instancias de una subclase de esa clase.
- ▶ Por ejemplo, si tenemos la clase *Figura* y una subclase suya llamada *Triangulo*:

```
jshell> Figura[] figuras = new Figura[5];  
figuras ==> Figura[5] { null, null, null, null, null }
```

- ▶ En cada elemento de *figuras* podremos meter una instancia de la clase *Figura* o de cualquier subclase suya:

```
jshell> figuras[2] = new Triangulo(20, 30); // alto y ancho  
$2 ==> Triangulo@1b701da1  
  
jshell> figuras  
figuras ==> Figura[5] { null, null, Triangulo@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.

3.9. Subtipado entre *arrays*

Subtipado entre *arrays*

- ▶ Entre los tipos de *arrays* se define una relación de subtipado directo ($<_1$) similar a la que hemos visto hasta ahora.
- ▶ Resumiendo, las reglas que definen esa relación son las siguientes:
 - Si S y T son tipos referencia, entonces $S[] <_1 T[]$ si y sólo si $S <_1 T$.

Debido a esto, se dice que los *arrays* de Java son **covariantes** con los tipos referencia (hablaremos más sobre este tema cuando estudiemos los *tipos genéricos*).

- $\text{Object}[] <_1 \text{Object}$.
- Si P es un tipo primitivo, entonces $P[] <_1 \text{Object}$.

3.10. java.util.Arrays

java.util.Arrays

- ▶ La clase `java.util.Arrays` contiene varios métodos estáticos para manipular *arrays*, incluyendo ordenación y búsqueda.
- ▶ También contiene un método factoría estático que permite ver a los *arrays* como listas, lo que será de interés cuando veamos las listas en Java.
- ▶ Los métodos de esta clase lanzan todos una excepción `NullPointerException` cuando el *array* especificado es una referencia nula.
- ▶ Su documentación se encuentra en el API de Java:

<https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/util/Arrays.html>

3.11. Copia y redimensionado de *arrays*

3.11.1 `Object.clone`

3.11.2 `System.arraycopy`

3.11.3 `Arrays.copyOf`

Copia y redimensionado de *arrays*

- ▶ Para hacer una copia de un *array*, se pueden usar varios métodos:
 - `Object.clone`.
 - `System.arraycopy`.
 - `Arrays.copyOf`.
 - `Arrays.copyOfRange`.
- ▶ Todos los métodos hacen **copias superficiales** (*shallow copys*) del *array*.
- ▶ Para hacer una **copia profunda** (*deep copy*) es necesario escribir un trozo de código que lo haga.

Object.clone

- ▶ La clase `Object` proporciona el método `clone`.
- ▶ Como los *arrays* en Java también son objetos de la clase `Object`, se puede usar este método para copiar un *array*.
- ▶ Este método copia todos los elementos del *array*, así que no sirve si lo que se quiere es hacer una copia parcial del *array*, es decir, si se quieren copiar sólo algunos elementos (un *subarray*).
- ▶ Por ejemplo:

```
jshell> String[] s = new String[] { "hola", "pepe", "juan" };
s ==> String[3] { "hola", "pepe", "juan" }

jshell> String[] w = s.clone();
w ==> String[3] { "hola", "pepe", "juan" }

jshell> s == w
$3 ==> false
```


System.arraycopy

- ▶ El método `arraycopy` de la clase `java.lang.System` es la mejor forma de hacer una **copia parcial** del array.
- ▶ Ofrece una forma sencilla de especificar el número total de elementos a copiar así como los índices de origen y destino.
- ▶ Su signatura es:

```
static void arraycopy(Object src, int srcPos, Object dest, int destPos,  
                      int length)
```

- ▶ Por ejemplo, el siguiente código:

```
System.arraycopy(origen, 3, destino, 2, 5);
```

copiará 5 elementos del array `origen` en el array `destino`, empezando por el índice 3 de `origen` y desde el índice 2 de `destino`.

Arrays.copyOf

- ▶ Se puede usar el método `copyOf` de la clase `java.util.Arrays` cuando se desean copiar los primeros elementos del `array`.
- ▶ También se puede usar cuando se desea extender el `array` creando un nuevo `array` con los mismos elementos del `array` original pero con más elementos al final, los cuales se rellenarán con los valores iniciales por defecto del tipo de los elementos del `array`.
- ▶ El método `copyOf` está sobrecargado para poder copiar `arrays` de cualquier tipo primitivo.
- ▶ Además, es un método genérico (ya veremos en su momento lo que eso significa), lo que le permite copiar `arrays` de cualquier tipo referencia.

- La signatura de los métodos `copyOf` sigue el siguiente esquema:

```
static T[] copyOf(T[] original, int newLength)
```

donde `T` es el tipo de los elementos del array `original`.

- El método devuelve un nuevo array a partir del original, con tantos elementos como se haya indicado en `newLength`.
 - Cuando `newLength < original.length`, el array nuevo tendrá sólo los primeros `newLength` elementos del original.
 - Cuando `newLength > original.length`, el array nuevo tendrá todos los elementos del original, y se rellenará por el final con tantos valores como sea necesario para que el nuevo array tenga `newLength` elementos.
- Esos valores serán los valores iniciales por defecto del tipo `T`.

- Por ejemplo, si tenemos:

```
jshell> String s = new String[] { "hola", "pepe", "juan", "maría", "antonio" }  
s ==> String[5] { "hola", "pepe", "juan", "maría", "antonio" }
```

- Creamos un nuevo *array* con menos elementos:

```
jshell> Arrays.copyOf(s, 2)  
$2 ==> String[2] { "hola", "pepe" }
```

- Creamos otro *array* con más elementos:

```
jshell> Arrays.copyOf(s, 7)  
$3 ==> String[7] { "hola", "pepe", "juan", "maría", "antonio", null, null }
```

Los nuevos elementos toman el valor **null**, que es el valor por defecto para los tipos referencia.

3.12. Comparación de *arrays*

3.12.1 `Arrays.equals`

Comparación de *arrays*

- Para comprobar si dos *arrays* `a1` y `a2` son iguales, podríamos usar:
 - `a1 == a2`, pero no resulta muy adecuado porque ya sabemos que lo que comprueba es si son **idénticos**.
 - `a1.equals(a2)`, pero tampoco resulta adecuado porque la implementación del método `equals` en los *arrays* es la que se hereda de la clase `Object`, la cual es idéntica a `a1 == a2`.

```
jshell> String[] s = new String[] { "hola", "pepe", "juan" };
s ==> String[3] { "hola", "pepe", "juan" }

jshell> String[] w = s.clone();
w ==> String[3] { "hola", "pepe", "juan" }

jshell> s == w
$3 ==> false

jshell> s.equals(w)
$4 ==> false
```

Arrays.equals

- ▶ La mejor opción para comparar dos *arrays* es usar el método `equals` de la clase `java.util.Arrays`.
- ▶ El método `Arrays.equals` está sobrecargado para poder comparar *arrays* de cualquier tipo primitivo.
- ▶ Además, es un método genérico (ya veremos en su momento lo que eso significa), lo que le permite comparar *arrays* de cualquier tipo referencia.
- ▶ Las signaturas de los métodos `equals` siguen el siguiente esquema:

```
static boolean equals(T[] a, T[] a2)
static boolean equals(T[] a, int aFromIndex, int aToIndex, T[] b,
                    int bFromIndex, int bToIndex)
```

donde `T` es el tipo de los elementos del *array* `original`.

► Ejemplo de uso:

```
jshell> int[] a = new int[] { 4, 2, 6 };  
a ==> int[3] { 4, 2, 6 }  
  
jshell> int[] b = new int[] { 4, 2, 6 };  
b ==> int[3] { 4, 2, 6 }  
  
jshell> a.equals(b)  
$3 ==> false  
  
jshell> Arrays.equals(a, b)  
$4 ==> true
```



```

jshell> String[] s = new String[] { "hola", "pepe", "juan" };
s ==> String[3] { "hola", "pepe", "juan" }

jshell> Arrays.equals(s, a)
| Error:
| no suitable method found for equals(java.lang.String[],int[])
|   method java.util.Arrays.equals(long[],long[]) is not applicable
|     (argument mismatch; java.lang.String[]
|       cannot be converted to long[])
|   method java.util.Arrays.equals(int[],int[]) is not applicable
|     (argument mismatch; java.lang.String[]
|       cannot be converted to int[])
|   [...]
|   method java.util.Arrays.<T>equals(T[],T[],
|     java.util.Comparator<? super T>) is not applicable
|     (cannot infer type-variable(s) T
|       (actual and formal argument lists differ in length))
|   method java.util.Arrays.<T>equals(T[],int,int,T[],int,int,
|     java.util.Comparator<? super T>) is not applicable
|     (cannot infer type-variable(s) T
|       (actual and formal argument lists differ in length))
| Arrays.equals(s, a)
| ^-----^

```

3.13. Arrays multidimensionales

3.13.1 Declaración

3.13.2 Creación

3.13.3 Inicialización

3.13.4 `Arrays.deepEquals()`

Arrays multidimensionales

- ▶ Se denomina **dimensión** de un *array* al número de índices que se necesitan para acceder a un elemento del *array*.
- ▶ Los *arrays* que hemos visto hasta ahora necesitan un único índice para acceder a cada uno de sus elementos, por lo que su dimensión es 1.

A los *arrays* de dimensión 1 también se les denominan ***arrays* unidimensionales**.

- ▶ Los *arrays* de dimensión mayor que 1 se denominan, genéricamente, ***arrays* multidimensionales**.

Esos *arrays* necesitan más de un índice para acceder a sus elementos individuales.

- ▶ Los *arrays* multidimensionales más habituales son los de dimensión 2 (también llamados ***arrays* bidimensionales**) y de dimensión 3 (***arrays* tridimensionales**).

Declaración

- ▶ En Java, los *arrays* multidimensionales se forman internamente haciendo que los elementos de un *array* sean, a su vez, otros *arrays*.
- ▶ Es decir: en Java, los *arrays* multidimensionales son *arrays* de *arrays*.
- ▶ Por tanto, al declarar un *array* tenemos que usar una sintaxis equivalente a la que hemos usado hasta ahora, pero usando tantas parejas de corchetes como sean necesarios.
- ▶ Por ejemplo, la siguiente sentencia declara una variable `x` como una variable que puede contener un *array* bidimensional donde sus elementos son enteros:

```
int[][] x;
```

Cuando la variable apunte a un *array* de ese tipo, se podrá acceder a cada uno de sus elementos indicando dos índices, cada uno entre un par de corchetes:

```
System.out.println(x[4][3]);
```

- En este caso, tenemos un *array* bidimensional.
- Los *arrays* bidimensionales se pueden representar como una **matriz** de elementos organizados en filas y columnas.
- Esos elementos se pueden almacenar por filas:

	0	1	2	3	...
0	<code>x[0][0]</code>	<code>x[0][1]</code>	<code>x[0][2]</code>	...	
1	<code>x[1][0]</code>	<code>x[1][1]</code>	<code>x[1][2]</code>	...	
2	<code>x[2][0]</code>	<code>x[2][1]</code>	<code>x[2][2]</code>	...	
⋮	⋮	⋮	⋮	⋮	

- O por columnas:

	0	1	2	3	...
0	<code>x[0][0]</code>	<code>x[1][0]</code>	<code>x[2][0]</code>	...	
1	<code>x[0][1]</code>	<code>x[1][1]</code>	<code>x[2][1]</code>	...	
2	<code>x[0][2]</code>	<code>x[1][2]</code>	<code>x[2][2]</code>	...	
⋮	⋮	⋮	⋮	⋮	

Creación

- Al crear un *array* multidimensional tenemos que indicar, al menos, el tamaño de la primera dimensión del *array*:

```
jshell> new int[4][]  
$1 ==> int[4][] { null, null, null, null }
```

- Podemos indicar el tamaño de más dimensiones, siempre en orden de izquierda a derecha:

```
jshell> new int[4][3]  
$12 ==> int[4][] { int[3] { 0, 0, 0 }, int[3] { 0, 0, 0 },  
                  int[3] { 0, 0, 0 }, int[3] { 0, 0, 0 } }
```

Inicialización

- La inicialización de un *array* multidimensional se hace de forma análoga a la de un *array* unidimensional:

```
jshell> new int[][] { null, null, null, null }  
$1 ==> int[4][] { null, null, null, null }  
  
jshell> new int[][] { null, new int[] { 4, 2, 3 }, null, null }  
$2 ==> int[4][] { null, int[3] { 4, 2, 3 }, null, null }
```

- En éste último caso, podemos hacer:

```
jshell> int[][] x = new int[][] { null, new int[] { 4, 2, 3 }, null, null }  
x ==> int[4][] { null, int[3] { 4, 2, 3 }, null, null }  
  
jshell> x[0]  
$2 ==> null  
  
jshell> x[1]  
$3 ==> int[3] { 4, 2, 3 }  
  
jshell> x[1][2]  
$4 ==> 3
```

Arrays.deepEquals()

- El método `Arrays.equals` sirve para comprobar si dos *arrays* son iguales, pero sólo funciona con *arrays* unidimensionales.
- En cambio, el método `Arrays.deepEquals` permite comprobar si dos *arrays* multidimensionales son iguales:

```
jshell> int[][] x = new int[][] { null, new int[] { 4, 2, 3 }, null, null }  
x ==> int[4][] { null, int[3] { 4, 2, 3 }, null, null }  
  
jshell> int[][] y = new int[][] { null, new int[] { 4, 2, 3 }, null, null }  
y ==> int[4][] { null, int[3] { 4, 2, 3 }, null, null }  
  
jshell> Arrays.equals(x, y)  
$3 ==> false  
  
jshell> Arrays.deepEquals(x, y)  
$4 ==> true
```


4. Bibliografía

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.