

# **Tema 3**

# **Programación básica y utilización de objetos en Java**

**Programación (Desarrollo de aplicaciones web)    Carlos Alberto Cortijo Bon**



Esta obra está bajo una [Licencia Creative Commons Atribución-NoComercial-SinDerivadas 4.0 Internacional](https://creativecommons.org/licenses/by-nc-nd/4.0/)

## Índice

1. Operadores de Java.....	1
1.1. Operador ternario.....	2
1.2. Operador unario de complemento.....	4
1.3. Conjunción y disyunción lógica.....	4
1.4. Desplazamiento de bits.....	5
1.5. El operador de asignación.....	7
2. Clases <i>wrapper</i> para tipos básicos.....	8
2.1. Creación de objetos de tipos <i>wrapper</i> .....	8
2.2. Métodos de las clases <i>wrapper</i> para lectura de datos de tipos básicos.....	9
3. Tipos enumerados.....	9
3.1. Iteración sobre todos los valores de un tipo enumerado.....	12
4. Repaso de la programación estructurada.....	12
5. Métodos estáticos.....	14
6. Métodos que no devuelven valores (void).....	16
7. Encapsulación de algoritmos en métodos.....	18
8. Métodos que devuelven cadenas de caracteres (String).....	21
9. Paso de <i>arrays</i> a métodos.....	23
9.1. Métodos que devuelven <i>arrays</i> .....	24
9.2. Modificación y copia de los contenidos de un <i>array</i> .....	25
10. Sentencias avanzadas para control de bucles: <i>break</i> y <i>continue</i> .....	27
11. Bucles anidados.....	32
12. Clases y objetos.....	34
12.1. Constructores y variables de instancia.....	36
12.2. Creación de objetos.....	36
12.3. El valor <i>null</i> .....	36
12.4. Métodos.....	37
12.5. Acceso a variables de instancia y métodos de una clase.....	37
12.6. Variables de clase y métodos estáticos.....	38
13. Biblioteca estándar de clases de Java.....	39
14. La clase <i>String</i> para cadenas de caracteres.....	41
15. La clase <i>Math</i> para operaciones matemáticas.....	46
16. La clase <i>Random</i> para generación de números aleatorios.....	47
17. Gestión de excepciones.....	49

# 1. Operadores de Java

Los operadores sirven para construir expresiones. Una expresión se puede evaluar y como resultado se obtiene un valor. Por ejemplo: cuando se evalúa la expresión  $3*4+6$ , da como resultado 18. En una expresión pueden intervenir variables. Si el valor de una variable *a* es 4, cuando se evalúa la expresión  $3*a+6$ , también da como resultado 18. El valor puede ser de diferentes tipos. La expresión  $a > 0$  da un valor de tipo boolean (*true* o *false*, dependiendo del valor de *a*) cuando se evalúa.

## Evaluación de expresiones

En general, las expresiones se evalúan de izquierda a derecha, pero solo si los operadores tienen la misma prioridad. Por ejemplo: para evaluar la expresión  $3+5-6$ , dado que los operadores  $+$  y  $-$  tienen igual prioridad, se haría primero la operación  $3+5$ , que daría 8 como resultado, y después  $8-6$ , con lo que se tendría como resultado final 2. Pero si la expresión fuera  $3+5*6$ , dado que el operador  $*$  tiene mayor prioridad que el operador  $+$ , se evaluaría primero  $5*6$ , que daría 30 como resultado, y después  $3+30$ , que daría 33 como resultado final. Los paréntesis se utilizan para alterar este orden. Si se quiere sumar 3 y 5 y multiplicar el resultado por 6, se utilizaría la expresión  $(3+5)*6$ .

En la siguiente tabla se muestran los operadores disponibles en Java, de mayor a menor precedencia. Se muestran resaltados los que no se vieron en el tema anterior.

<b>Postfijos unarios</b>	<i>expr</i> ++ <i>expr</i> --
<b>Prefijos unarios</b>	++ <i>expr</i> -- <i>expr</i> + <i>expr</i> - <i>expr</i> ~ !
<b>Multiplicativos</b>	* / %
<b>Aditivos</b>	+ -
<b>De desplazamiento de bits</b>	<< >> >>>
<b>Relacionales</b>	< > <= >= instanceof
<b>De igualdad</b>	== !=
<b>Conjunción a nivel de bits</b>	&
<b>Disyunción exclusiva (XOR) a nivel de bits</b>	^
<b>Disyunción inclusiva a nivel de bits</b>	
<b>Conjunción lógica</b>	&&
<b>Disyunción lógica</b>	
<b>Operador ternario</b>	<i>condición</i> ? <i>valor1</i> : <i>valor2</i>
<b>Operador de asignación</b>	= += -= *= /= %= &= ^=  = <<= >>= >>>=

Los que no se han visto en el tema anterior son los siguientes:

- El operador ternario *condición* ? *valor1* : *valor2*.
- Operadores a nivel de bit.
  - Operador unario de complemento: ~*valor*.
  - Conjunción y disyunción (inclusiva y exclusiva) a nivel de bits.
    - Operadores binarios. *valor1* & *valor2*, *valor1* | *valor2* y *valor1* ^ *valor2*.

- Operadores unarios. Asignan a `valor1` el resultado de aplicar el correspondiente operador binario con `valor1` y `valor2`.

`valor1 &= valor2` equivale a `valor1 = valor1 & valor2`

`valor1 |= valor2` equivale a `valor1 = valor1 | valor2`

`valor1 ^= valor2` equivale a `valor1 = valor1 ^ valor2`

- Desplazamiento a nivel de bits.

- Operadores binarios. `valor >> desp`, `valor >>> desp` y `valor << desp`.

- Operadores unarios de asignación. Asignan a `valor` el resultado de aplicar el correspondiente operador binario con `valor` y `desp`.

`valor >>= desp` equivale a `valor1 = valor1 >> desp`

`valor >>>= desp` equivale a `valor1 = valor1 >>> desp`

`valor <<= desp` equivale a `valor1 = valor1 << desp`

- El operador `instanceof`, que se verá en el siguiente tema.
- El operador de asignación. Se ha visto la sentencia de asignación, pero en realidad, la asignación es también un operador, y este nuevo aspecto se verá a continuación.

## 1.1. Operador ternario

El operador ternario se llama así porque opera con tres operandos: una condición y dos valores. Tiene la siguiente forma:

**`condición ? valor1 : valor2`**

El resultado de evaluar esta expresión es:

- Si se cumple **`condición`**: **`valor1`**.
- En otro caso (si no se cumple **`condición`**): **`valor2`**.

Se suele utilizar para asignar un valor u otro a una variable dependiendo de si se cumple o no una condición.

Por ejemplo, supongamos que se aplica un porcentaje de descuento u otro a un artículo dependiendo de si la cantidad comprada es mayor que 50 o no. Se podría hacer con:

```
float descuento = (cantidad < 50) ? 5 : 10;
```

Esto sería equivalente a:

```
float descuento;

// Se asigna un valor a cantidad

if(cantidad < 50) {
    descuento = 5;
} else {
    descuento = 10;
}
```

También se podría asignar un porcentaje de descuento a un cliente dependiendo de si es preferente o no.

```
bool clPref;  
  
// Se asigna un valor a clPref (true o false)  
  
float descuento = clPref ? 10 : 5;
```

### Actividad 3.1

A partir del siguiente programa, crear uno que escriba lo siguiente, utilizando `printf` y el operador ternario.

```
1 es un número impar.  
2 es un número par.  
...  
49 es un número impar.  
50 es un número par.
```

```
boolean par = false;  
for (int i = 1; i < 10; i++){  
    par = !par;  
    System.out.println(par);  
}
```

### Actividad 3.2

\*

Completa el siguiente programa que utiliza el operador ternario para escribir resultado o resultados dependiendo del valor de una variable `int numResult`. Esta puede tomar valores de 0 en adelante.

```
for(int numResult = 0; i < 10 ; i++) {  
  
    System.out.printf("%d resultado%s.\n", numResult, ■ ? ■ : ■);  
  
}
```

Se puede utilizar repetidamente este operador para evitar sentencias `if ... else if ... else` en las que se asigna un valor u otro a una variable dependiendo de si se cumplen o no determinadas condiciones. Con el siguiente código, se asigna a una variable `comp` un valor 1, -1 o 0 dependiendo de si, el valor de **a** es mayor que el de **b**, el de **b** mayor que el de **a**, o ambos son iguales.

```
int comp = (a > b) ? 1 : (a < b) ? -1 : 0;
```

Posiblemente la anterior expresión queda más clara formateada de la siguiente manera:

```
int comp =  
    (a > b) ? 1 :  
    (a < b) ? -1 :  
    0;
```

## 1.2. Operador unario de complemento

Cuando se aplica a un valor de tipo entero, devuelve un valor que es el resultado de cambiar cada uno de los dígitos binarios que lo componen: los ceros por unos y los unos por ceros. Este es el complemento a uno del valor.

A continuación se muestra el resultado de aplicar este operador a un valor de tipo `int`, que se representa internamente con 32 bits.

El valor que se asigna a una variable de este tipo se especifica normalmente como un número decimal en un programa de Java, pero se puede especificar el valor en binario anteponiéndole el prefijo `0b`, como se hace en el siguiente programa, cuya salida se muestra también. Para que quede más claro cómo se realizan las operaciones y su resultado, este programa utiliza mecanismos algo avanzados para formatear su salida. Pero no es necesario entenderlos, solo entender el resultado de la operación a nivel de bit que realiza con el valor `num` de tipo `int`:

```
int num = 0b10001110;

System.out.printf("  %s\n~:%s\n",
    String.format("%1$32s", Integer.toUnsignedString(num, 2)).replace(' ', '0'),
    String.format("%1$32s", Integer.toUnsignedString(~num, 2)).replace(' ', '0')
);
```

0000000000000000000000000000000010001110  
~:1111111111111111111111111111111101110001

## 1.3. Conjunción y disyunción lógica

Las operaciones de conjunción y disyunción lógica se aplican a nivel de bits, sobre números de tipo `int` (de 32 bits) o de tipo `long` (de 64 bits). En las siguientes tablas se muestra el resultado de cada operación dependiendo de los valores de los dos bits a los que se aplica.

Conjunción lógica (AND)	Disyunción lógica (OR)	Disyunción lógica exclusiva (XOR)																											
<table border="1"> <tr><td>&amp;</td><td>0</td><td>1</td></tr> <tr><td>0</td><td>0</td><td>0</td></tr> <tr><td>1</td><td>0</td><td>1</td></tr> </table>	&	0	1	0	0	0	1	0	1	<table border="1"> <tr><td> </td><td>0</td><td>1</td></tr> <tr><td>0</td><td>0</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>1</td></tr> </table>		0	1	0	0	1	1	1	1	<table border="1"> <tr><td>^</td><td>0</td><td>1</td></tr> <tr><td>0</td><td>0</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>0</td></tr> </table>	^	0	1	0	0	1	1	1	0
&	0	1																											
0	0	0																											
1	0	1																											
	0	1																											
0	0	1																											
1	1	1																											
^	0	1																											
0	0	1																											
1	1	0																											
El bit resultante es 1 solo cuando ambos bits son 1.	El bit resultante es 1 cuando al menos uno de los bits es 1.	El bit resultante es 1 solo cuando ambos bits son distintos (uno es 1 y el otro 0).																											

Los tipos numéricos enteros (como `int` y `long`) se representan internamente como números binarios, es decir, como una secuencia de bits. Los operadores se aplican a los bits situados en igual posición (es decir, con igual peso) en ambos números, para obtener el correspondiente bit del número resultante.

A continuación se muestra el resultado de aplicar estos operadores a dos valores de tipo `int`, que se representan internamente con 32 bits.

Los valores se especifican en binario, anteponiendo el prefijo `0b`. Se muestra la salida del programa, que utiliza mecanismos algo avanzados para formatear su salida. Pero no es necesario entenderlos, solo entender el resultado de las operaciones a nivel de bit que realiza con dos valores `num1` y `num2` de tipo `int`:

- La conjunción lógica: `num1 & num2`.
- La disyunción lógica inclusiva: `num1 | num2`.
- La disyunción lógica exclusiva: `num1 ^ num2`.

<pre>int num1 = 0b10001110; int num2 = 0b11100011; System.out.printf(" %s\n %s\n",     String.format("%1\$16s", Integer.toUnsignedString(num1, 2)).replace(' ', '0'),     String.format("%1\$16s", Integer.toUnsignedString(num2, 2)).replace(' ', '0')); System.out.println(String.format("&amp;:%1\$16s",     Integer.toUnsignedString(num1 &amp; num2, 2)).replace(' ', '0')); System.out.printf("%d &amp; %d = %d\n", num1, num2, num1 &amp; num2); System.out.println(String.format(" :%1\$16s",     Integer.toUnsignedString(num1   num2, 2)).replace(' ', '0')); System.out.printf("%d   %d = %d\n", num1, num2, num1   num2); System.out.println(String.format("^:%1\$16s",     Integer.toUnsignedString(num1 ^ num2, 2)).replace(' ', '0')); System.out.printf("%d ^ %d = %d\n", num1, num2, num1 ^ num2);</pre>	<pre>0000000010001110 0000000011100011  &amp;:0000000010000010 142 &amp; 227 = 130   :0000000011101111 142   227 = 239  ^:0000000011011011 142 ^ 227 = 109</pre>
---	--

Para todos los operadores anteriores se puede utilizar una forma abreviada cuando el primero de los dos operandos es la misma variable a la que se le está asignando el resultado de la operación.

Expresión	Equivalente a
<i>num</i> &= <i>desp</i>	<i>num</i> = <i>num</i> & <i>desp</i>
<i>num</i>  = <i>desp</i>	<i>num</i> = <i>num</i>   <i>desp</i>
<i>num</i> ^= <i>desp</i>	<i>num</i> = <i>num</i> ^ <i>desp</i>

## 1.4. Desplazamiento de bits

Las operaciones de desplazamiento de bits se aplican sobre un único operando, y operan también a nivel de bit. Son las siguientes:

<< Desplazamiento a la izquierda	>>> Desplazamiento a la derecha sin signo
<p>Desplaza todos los bits a la izquierda el número de veces que se indique, rellenando con ceros por la derecha. Equivale a multiplicar por 2 si se considera el número sin signo (para Java, el dígito más significativo indica el signo, que es negativo si es 1, y positivo si es 0).</p> <pre>int num = 0b11; for (int i = 0; i &lt;= 32; i++) {     System.out.printf("%s (%s)\n",         String.format("%1\$32s",             Integer.toUnsignedString(num, 2))             .replace(' ', '0'),         Integer.toUnsignedString(num)); }; num &lt;&lt;= 1; }</pre>	<p>Desplaza todos los bits a la derecha el número de veces que se indique, rellenando con ceros por la derecha. Equivale a dividir por 2 si se considera el número sin signo. (para Java, el dígito más significativo indica el signo, que es negativo si es 1, y positivo si es 0).</p> <pre>int num = 0b11000000000000000000000000000000; for (int i = 0; i &lt;= 32; i++) {     System.out.printf("%s (%s)\n",         String.format("%1\$32s",             Integer.toUnsignedString(num, 2))             .replace(' ', '0'),         Integer.toUnsignedString(num)); }; num &gt;&gt;= 1; }</pre>
<pre>0000000000000000000000000000000011 (3) 0000000000000000000000000000000010 (6) 000000000000000000000000000000001100 (12) 0000000000000000000000000000000011000 (24) 00000000000000000000000000000000110000 (48) 000000000000000000000000000000001100000 (96) 0000000000000000000000000000000011000000 (192) 00000000000000000000000000000000110000000 (384) 000000000000000000000000000000001100000000 (768) 0000000000000000000000000000000011000000000 (1536) 00000000000000000000000000000000110000000000 (3072) 000000000000000000000000000000001100000000000 (6144) 0000000000000000000000000000000011000000000000 (12288) 00000000000000000000000000000000110000000000000 (24576) 000000000000000000000000000000001100000000000000 (49152) 0000000000000000000000000000000011000000000000000 (98304) 00000000000000000000000000000000110000000000000000 (196608)</pre>	<pre>1100000000000000000000000000000000 (3221225472) 0110000000000000000000000000000000 (1610612736) 0011000000000000000000000000000000 (805306368) 0001100000000000000000000000000000 (402653184) 0000110000000000000000000000000000 (201326592) 0000011000000000000000000000000000 (100663296) 0000001100000000000000000000000000 (50331648) 0000000110000000000000000000000000 (25165824) 0000000011000000000000000000000000 (12582912) 0000000001100000000000000000000000 (6291456) 0000000000110000000000000000000000 (3145728) 0000000000011000000000000000000000 (1572864) 0000000000001100000000000000000000 (786432) 0000000000000110000000000000000000 (393216) 0000000000000011000000000000000000 (196608) 0000000000000001100000000000000000 (98304) 0000000000000000110000000000000000 (49152)</pre>

[illegible]

>> Desplazamiento a la derecha con signo			
Desplaza todos los bits a la derecha el número de veces que se indique, rellenando por la derecha con el mismo valor del dígito más significativo. Se dice con signo porque, para Java, el dígito más significativo indica el signo del número: negativo si es 1, positivo si es 0.			
<pre>int num = 0b11000000000000000000000000000000; for (int i = 0; i &lt;= 32; i++) {     System.out.printf("%s (%s)\n",         String.format("%1\$32s",             Integer.toUnsignedString(num, 2))             .replace(' ', '0'),         Integer.toUnsignedString(num)     );     num &gt;&gt;= 1; }</pre>		<pre>int num = 0b01100000000000000000000000000000; for (int i = 0; i &lt;= 32; i++) {     System.out.printf("%s (%s)\n",         String.format("%1\$32s",             Integer.toUnsignedString(num, 2))             .replace(' ', '0'),         Integer.toUnsignedString(num)     );     num &gt;&gt;= 1; }</pre>	
11000000000000000000000000000000	(3221225472)	01100000000000000000000000000000	(1610612736)
11100000000000000000000000000000	(3758096384)	00110000000000000000000000000000	(805306368)
11110000000000000000000000000000	(4026531840)	00011000000000000000000000000000	(402653184)
11111000000000000000000000000000	(4160749568)	00001100000000000000000000000000	(201326592)
11111100000000000000000000000000	(4227858432)	00000110000000000000000000000000	(100663296)
11111110000000000000000000000000	(4261412864)	00000011000000000000000000000000	(50331648)
11111111000000000000000000000000	(4278190080)	00000001100000000000000000000000	(25165824)
11111111100000000000000000000000	(4286578688)	00000000110000000000000000000000	(12582912)
11111111110000000000000000000000	(4290772992)	00000000011000000000000000000000	(6291456)
11111111111000000000000000000000	(4292870144)	00000000001100000000000000000000	(3145728)
11111111111100000000000000000000	(4293918720)	00000000000110000000000000000000	(1572864)
11111111111110000000000000000000	(4294443008)	00000000000011000000000000000000	(786432)
11111111111111000000000000000000	(4294705152)	00000000000001100000000000000000	(393216)
11111111111111100000000000000000	(4294836224)	00000000000000110000000000000000	(196608)
11111111111111110000000000000000	(4294901760)	00000000000000011000000000000000	(98304)
11111111111111111000000000000000	(4294934528)	00000000000000001100000000000000	(49152)
11111111111111111100000000000000	(4294950912)	00000000000000000110000000000000	(24576)
11111111111111111110000000000000	(4294959104)	00000000000000000011000000000000	(12288)
11111111111111111111000000000000	(4294963200)	00000000000000000001100000000000	(6144)
11111111111111111111100000000000	(4294965248)	00000000000000000000110000000000	(3072)
11111111111111111111110000000000	(4294966272)	00000000000000000000011000000000	(1536)
11111111111111111111111000000000	(4294966784)	00000000000000000000000110000000	(768)
11111111111111111111111100000000	(4294967040)	00000000000000000000000011000000	(384)
11111111111111111111111110000000	(4294967168)	00000000000000000000000001100000	(192)
11111111111111111111111110000000	(4294967232)	00000000000000000000000000110000	(96)
11111111111111111111111111000000	(4294967264)	00000000000000000000000000011000	(48)
11111111111111111111111111100000	(4294967280)	00000000000000000000000000001100	(24)
11111111111111111111111111110000	(4294967288)	00000000000000000000000000000110	(12)
11111111111111111111111111111000	(4294967292)	00000000000000000000000000000011	(6)
11111111111111111111111111111100	(4294967294)	00000000000000000000000000000011	(3)
11111111111111111111111111111111	(4294967295)	00000000000000000000000000000001	(1)
11111111111111111111111111111111	(4294967295)	00000000000000000000000000000000	(0)
11111111111111111111111111111111	(4294967295)	00000000000000000000000000000000	(0)

Para todos los operadores anteriores se puede utilizar una forma abreviada cuando el primero de los dos operandos es la misma variable a la que se le está asignando el resultado de la operación.



Expresión	Equivalente a
<code>num &lt;=&lt; desp</code>	<code>num = num &lt;&lt; desp</code>
<code>num &gt;=&gt; desp</code>	<code>num = num &gt;&gt; desp</code>
<code>num &gt;&gt;&gt;= desp</code>	<code>num = num &gt;&gt;&gt; desp</code>

**Actividad 3.3**

\*\*\*

Crea un programa que pida los cuatro bytes de una dirección IP y una longitud de máscara de red, que debe estar entre 8 y 24. El programa debe dar la dirección IP de la red. Por ejemplo: si la dirección IP es 192.168.123.14 y la máscara de red es de 20 bits, entonces la dirección IP expresada en binario es 11000000.10101000.01111011.00001110, y la máscara 11111111.11111111.11110000.00000000, y la dirección de red el resultado de hacer la conjunción lógica (AND) de ambos números, es decir: 11000000.10101000.01110000.00000000, que pasado a decimal es 192.168.112.0, que es la respuesta que debe dar el programa.

## 1.5. El operador de asignación

La sentencia de asignación es realmente un operador que tiene como efecto añadido la asignación de un valor a una variable. El valor resultante de la evaluación del operador es el propio valor que se asigna. Esto permite asignar el mismo valor a varias variables en una misma sentencia, como se muestra en el siguiente ejemplo:

```
boolean fin = encontrado = false;
int xMin = xMax = yMin = yMax = 0;
```

La primera sentencia funciona de la siguiente manera:

- A **fin** se le asigna el resultado de evaluar la expresión **encontrado = false**.
- Por tanto, se evalúa la expresión **encontrado = false**. A la variable **encontrado** se le asigna el valor **false**, y este mismo valor es el resultado de evaluar la expresión.
- Por tanto, finalmente, a la variable **fin** se le asigna el valor **false**, que es además el resultado final de evaluar la expresión **fin = encontrado = false**.

Este mecanismo permite encadenar tantas asignaciones de variables como se quiera. En el segundo ejemplo se usa repetidamente este mecanismo para asignar el valor 0 a cuatro variables: **xMin**, **xMax**, **yMin** e **yMax**. Con el resultado final de evaluar la expresión no se hace nada, pero lo importante es que al evaluarla se ha asignado el valor 0 a cuatro variables.

Se podría haber hecho lo mismo con el siguiente código, en el que se separa la declaración de las variables y la asignación de un valor a ellas.

```
boolean fin, encontrado;
int xMin, xMax, yMin, yMax;

fin = encontrado = false;
xMin = xMax = yMin = yMax = 0;
```

Más adelante se verá otro posible uso de esta dualidad de la asignación como sentencia y como expresión. Por ahora, baste un avance. Se pueden leer todos los bytes de un fichero, uno a uno, con el siguiente código.

```
FileInputStream fis = new FileInputStream("fichero.txt") {
    int unByte;
    while ((unByte = fis.read()) != -1) {
```

```
System.out.printf("%3d(%c)\n", unByte, (char) unByte);  
}
```

En el código resaltado, se utiliza un operador de asignación para asignar el valor del byte leído del fichero a la variable `unByte`. Pero además se compara este valor con `-1`, que indicaría que se ha llegado al final del fichero.

## 2. Clases *wrapper* para tipos básicos

En Java existen clases correspondientes para cada uno de los tipos básicos, según se muestra en la siguiente tabla. Se conocen como clases *wrapper* o envoltorio.

	Tipo básico	Clase	Longitud en bits
Entero	<code>int</code>	<code>Integer</code>	32
Entero largo	<code>long</code>	<code>Long</code>	64
Entero corto	<code>short</code>	<code>Short</code>	16
Número real	<code>float</code>	<code>Float</code>	32
Número real con doble precisión	<code>double</code>	<code>Double</code>	64
Byte	<code>byte</code>	<code>Byte</code>	8
Carácter Unicode	<code>char</code>	<code>Character</code>	16
Booleano	<code>boolean</code>	<code>Boolean</code>	1

### 2.1. Creación de objetos de tipos *wrapper*

Estas clases *wrapper* tienen un tratamiento especial. Se puede usar el operador de asignación para asignar valores de un tipo básico a un objeto de la clase *wrapper* correspondiente (*autoboxing*), y viceversa (*unboxing*).

Autoboxing	Unboxing
<code>Integer i1 = 4;</code>	<code>int i2 = i1;</code>
<code>Long l1 = 4L;</code>	<code>long l2 = l1;</code>
<code>Short s1 = 4;</code>	<code>short s2 = s1;</code>
<code>Float f1 = 4.56f;</code>	<code>float f2 = f1;</code>
<code>Double d1 = 4.56;</code>	<code>double d2 = d1;</code>
<code>Byte b1 = (byte) 234;</code>	<code>byte b2 = b1;</code>
<code>Character c1 = 'ñ';</code>	<code>char c2 = c1;</code>
<code>Boolean b11 = true;</code>	<code>boolean b12 = b11;</code>

Nota: se puede también utilizar la sintaxis habitual en Java para crear objetos de una clase *wrapper*. Por ejemplo:

```
Integer i3 = new Integer(4).
```

Pero esto está *deprecated* (considerado obsoleto) y por tanto se podría eliminar en versiones futuras.

## 2.2. Métodos de las clases *wrapper* para lectura de datos de tipos básicos

Todas las clases *wrapper* tienen métodos con nombre `parseTTT`, donde *TTT* es el nombre de la clase, que permiten obtener un valor del tipo básico asociado a partir de un `String` que contiene su representación como cadena de caracteres. Si esto no es posible, porque la cadena de caracteres no tiene un formato apropiado, producen una excepción de la clase `NumberFormatException`.

Estos métodos son muy útiles para leer datos desde los argumentos de línea de comandos, porque estos se obtienen como valores de tipo `String` desde el parámetro `String[] args` del método `main`.

El siguiente programa de ejemplo intenta convertir el primer argumento de línea de comandos a un `int`, y el segundo a un `double`. Si el formato de cualquiera de ellos no es correcto, se genera una excepción de tipo `NumberFormatException`. Pero se captura, se muestra un mensaje de error, y se termina la ejecución del programa con `return`.

```
package parsingtiposbasicos;

public class ParsingTiposBasicos {

    public static void main(String[] args) {

        if (args.length < 2) {
            System.out.println("ERROR: indicar un número entero y uno con decimales.");
            return;
        }

        try {
            int n = Integer.parseInt(args[0]);
            System.out.printf("1-Número entero: %d\n", n);
        } catch (NumberFormatException e) {
            System.out.printf("Valor incorrecto: %s, no es un entero.\n", args[0]);
            return;
        }

        try {
            double d = Double.parseDouble(args[1]);
            System.out.printf("2-Número real: %f\n", d);
        } catch (NumberFormatException e) {
            System.out.printf("Valor incorrecto: %s, no es un número real.\n", args[1]);
            return;
        }

    }

}
```

## 3. Tipos enumerados

Un tipo enumerado es un tipo de datos especial que admite un conjunto cerrado de posibles valores predefinidos y constantes. Por ejemplo:

- Las direcciones IZQUIERDA, DERECHA, ARRIBA y ABAJO.
- Los días de la semana LUNES, MARTES, MIERCOLES, JUEVES, VIERNES, SABADO y DOMINGO.
- Los reinos de los seres vivos eucariotas: ANIMALES, VEGETALES, HONGOS, PROTOZOOS, ALGAS.

Tipos enumerados básicos

Un tipo enumerado se define con la palabra reservada `enum`. Un tipo enumerado para las direcciones se podría definir de la siguiente forma.

```
enum Direccion {
    IZQUIERDA, DERECHA, ARRIBA, ABAJO
}
```

Se puede declarar una variable de este tipo y asignarle un valor de la siguiente forma:

```
Direccion dir = Direccion.IZQUIERDA;
```

Los tipos enumerados se prestan mucho a su uso con sentencias `switch` en las que se hace una cosa distinta para cada uno de sus posibles valores. Para ellas hay una sintaxis más tradicional (que de hecho es la misma que existe en los lenguajes de programación C y C++), y una nueva (*rule switch*), que se introdujo a partir de una determinada versión de Java. Se muestra un ejemplo a continuación, expresado con ambas variantes.

Sintaxis clásica	Sintaxis <i>rule switch</i>
<pre>Direccion dir = Direccion.ABAJO; int x = 0; int y = 0;</pre>	
<pre>switch (dir) {     case IZQUIERDA:         x--;         break;     case DERECHA:         x++;         break;     case ARRIBA:         y--;         break;     case ABAJO:         y++;         break; }</pre>	<pre>switch (dir) {     case IZQUIERDA -&gt;         x--;     case DERECHA -&gt;         x++;     case ARRIBA -&gt;         y--;     case ABAJO -&gt;         y++; }</pre>
<pre>System.out.printf("Fin: (%d,%d)\n", x, y);</pre>	

#### Actividad 3.4

\*

Crea un programa con un *array* con tipo base `Direccion`, que contenga varias direcciones (elementos de tipo `Direccion`) que representen sucesivos desplazamientos desde una posición inicial. Por ejemplo:

```
Direccion[] desplazamientos = {
    Direccion.ARRIBA,
    Direccion.DERECHA,
```

```

    Direccion.DERECHA,
    Direccion.ABAJO,
    Direccion.IZQUIERDA
}

```

Define dos variables `x` e `y` con un valor inicial cualquiera. Utiliza un bucle `for` para obtener una a uno a uno los desplazamientos contenidos en el *array*, y dentro de él una sentencia `switch` como la anterior para calcular la nueva posición, a partir de la anterior, después de cada desplazamiento. Haz que el programa escriba al principio la posición inicial y al final la posición final.

Pueden utilizarse cláusulas `case` con más de un valor, y cláusulas `default`, como se muestra en los siguientes ejemplos.

El primero es con el tipo `Direccion` ya visto.

Sintaxis clásica	Sintaxis <i>rule switch</i>
<pre> Direccion dir = Direccion.IZQUIERDA;  switch (dir) {     case IZQUIERDA:     case DERECHA:         System.out.println("Horizontal.");         break;     case ARRIBA:     case ABAJO:         System.out.println("Vertical.");         break; } </pre>	<pre> switch (dir) {     case IZQUIERDA, DERECHA -&gt;         System.out.println("Horizontal.");     case ARRIBA, ABAJO -&gt;         System.out.println("Vertical."); } </pre>

El siguiente es con un nuevo tipo `DiaSemana` para los días de la semana.

Sintaxis clásica	Sintaxis <i>rule switch</i>
<pre> package dias;  public class Dias {      enum DiaSemana {         LUNES, MARTES, MIERCOLES,         JUEVES, VIERNES, SABADO, DOMINGO     }      public static void main(String[] args) {          DiaSemana dia = DiaSemana.VIERNES;          switch (dia) {             case SABADO:             case DOMINGO:                 System.out.println("Festivo");                 break; </pre>	<pre> switch (dia) {     case SABADO, DOMINGO -&gt;         System.out.println("Festivo");     default -&gt;         System.out.println("Laborable"); } </pre>

```

        default:
            System.out.println("Laborable");
    }
}
}

```

### 3.1. Iteración sobre todos los valores de un tipo enumerado

El método `values()` devuelve un *array* con todos los posibles valores del tipo enumerado (`Direccion`). El siguiente programa itera sobre este *array* para mostrar todos los valores. Para cada uno, el método `name()` devuelve una descripción textual, y el método `ordinal()` devuelve un número de orden.

```

enum Direccion {
    IZQUIERDA, DERECHA, ARRIBA, ABAJO
}

for (Direccion dir : Direccion.values()) {
    System.out.printf("%d: %s\n", dir.ordinal(), dir.name());
}

```

La salida del programa anterior es la siguiente.

```

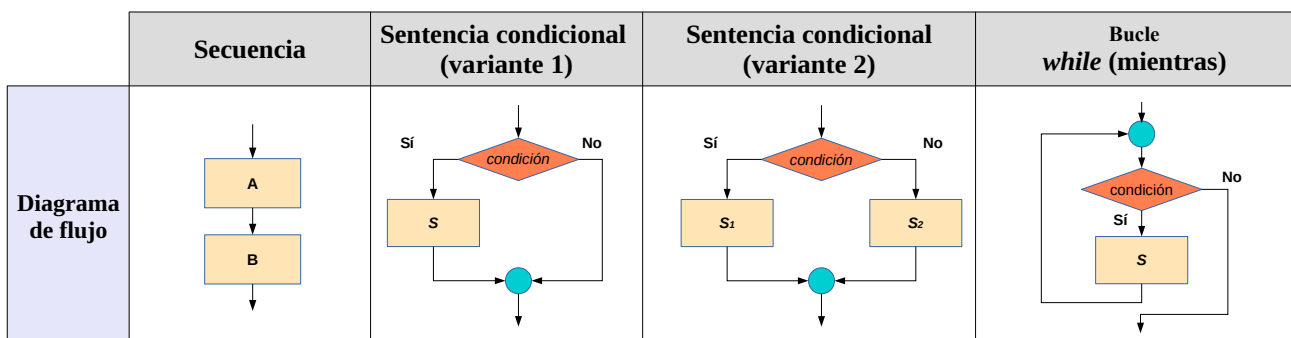
0: IZQUIERDA
1: DERECHA
2: ARRIBA
3: ABAJO

```

## 4. Repaso de la programación estructurada

En el tema anterior se vieron los aspectos fundamentales de la programación estructurada en lenguaje Java.

La **programación estructurada**, como ya se ha comentado, se basa en la utilización de tres tipos de estructuras básicas para combinar sentencias: la secuencia, la sentencia condicional, y el bucle *while*.



	Secuencia	Sentencia condicional (variante 1)	Sentencia condicional (variante 2)	Bucle <i>while</i> (mientras)
Código en Java	A; B;	<b>if</b> ( <i>condición</i> ) { <i>S</i> ; }	<b>if</b> ( <i>condición</i> ) { <i>S</i> <sub>1</sub> ; } <b>else</b> { <i>S</i> <sub>2</sub> ; }	<b>while</b> ( <i>condición</i> ) { <i>S</i> ; }

Las sentencias de Java anteriores tienen correspondencia directa con las estructuras básicas de la programación estructurada. Pero además, en Java y en general en todos los lenguajes de programación existen otras sentencias que no son estrictamente necesarias, pero sí muy útiles, porque permiten expresar muchos programas de manera más sencilla y concisa, y mejoran la claridad y legibilidad del código.

Son otros dos tipos de bucles (**for** y **do ... while**) y otros tipos de sentencias condicionales (sentencias **if ... else if ... else** y sentencias de selección múltiple **switch**).

	Bucle <b>for</b>	Bucle <b>do ... while</b>
Diagrama de flujo		<pre> graph TD     Start(( )) --&gt; S[S]     S --&gt; Cond{Condición}     Cond -- Sí --&gt; S     Cond -- No --&gt; Exit(( ))         </pre>
Código en Java	<b>for</b> ( <i>Inic</i> ; <i>cond</i> ; <i>Fin</i> ) { <i>S</i> ; }	<b>do</b> { <i>S</i> ; } <b>while</b> ( <i>condición</i> );
Código equivalente en Java	<i>Inic</i> ; <b>while</b> ( <i>cond</i> ) { <i>S</i> ; <i>Fin</i> ; }	<i>S</i> ; <b>while</b> ( <i>condición</i> ) { <i>S</i> ; }

Sentencia <b>if ... else if ...</b>	Sentencia <b>if ... else if ... else</b>	Sentencia de selección múltiple <b>switch</b>
--	---	--

Código en Java	<pre>if (cond1) {     S<sub>1</sub>; } else if (cond2) {     S<sub>2</sub>; }</pre>	<pre>if (cond1) {     S<sub>1</sub>; } else if (cond2) {     S<sub>2</sub>; } else {     S<sub>3</sub>; }</pre>	<pre>switch (v) {     case val1:         S<sub>1</sub>;         break;     case val2:         S<sub>2</sub>;         break;     case val3:         S<sub>3</sub>;         break;     case val4:         S<sub>4</sub>;         break; }</pre>	<pre>switch (v) {     case val1:         S<sub>1</sub>;         break;     case val2:     case val3:         S<sub>2</sub>;         break;     case val4:         S<sub>3</sub>;         break;     default:         S<sub>4</sub>; }</pre>
Código equivalente en Java	<pre>if (cond1) {     S<sub>1</sub>; } else {     if (cond2) {         S<sub>2</sub>;     } }</pre>	<pre>if (cond1) {     S<sub>1</sub>; } else {     if (cond2) {         S<sub>2</sub>;     } else {         S<sub>3</sub>;     } }</pre>	<pre>if (v==val1) {     S<sub>1</sub>; } else if (v==val2) {     S<sub>2</sub>; } else if (v==val3) {     S<sub>3</sub>; } else if (v==val4) {     S<sub>4</sub>; }</pre>	<pre>if (v == val1) {     S<sub>1</sub>; } else if (v==val2    v==val3) {     S<sub>2</sub>; } else if (v==val4) {     S<sub>3</sub>; } else {     S<sub>4</sub>; }</pre>
Diagrama de flujo				

## 5. Métodos estáticos

Hay un aspecto nuevo de la programación estructurada que se va a introducir a continuación, y es la posibilidad de encapsular bloques de código en una unidad identificada por un nombre, que tiene unos parámetros para recibir valores de entrada, y que puede devolver un valor de un tipo determinado.

El siguiente programa define y utiliza dos métodos:

- `doble`, que tiene como parámetro un número entero, y devuelve un entero que es el doble del valor que se le pasa en este parámetro.



- suma, que tiene como parámetros dos números enteros, y devuelve un entero que es la suma de los dos valores que se le pasan en esos parámetros.

Un método devuelve un valor con la sentencia **return**.

```
public class PruebaDosMetodos {  
  
    static int doble(int n) {  
        return 2*n;  
    }  
  
    static int suma(int n1, int n2) {  
        return n1 + n2;  
    }  
  
    public static void main(String[] args) {  
  
        int num = 4;  
  
        int dNum = doble(num);  
  
        System.out.printf("El doble de %d es %d.\n", num, dNum);  
  
        System.out.printf("El doble de 2 es %d.\n", 2, doble(2));  
  
        int a = 5;  
        int b = 7;  
  
        System.out.println(suma(a, b));  
        System.out.println(suma(a, doble(b)));  
        System.out.println(doble(suma(a, 9)));  
  
    }  
}
```

#### Parámetros vs argumentos

Los argumentos son los valores que se pasan a los parámetros de un método en el momento de llamarlo. Por ejemplo, en la sentencia `int dNum = doble(num)` del programa anterior, se llama al método `doble` con el argumento 4 para el parámetro `n`. Es decir, con 4, que es el valor que tiene la variable `num` en el momento de realizar la llamada.

#### Recuerda

Todos los métodos que se van a crear serán estáticos (**static**). Por ahora no interesa lo que significa, se verá más adelante.

#### Recuerda: el método `main`

Los programas de Java que se ha visto hasta ahora tienen un método **public static void main(String[] args)**. Ya se ha visto que es el punto de entrada de los programas de Java, y tiene un parámetro `args` de tipo `String[]` que sirve para pasarle los argumentos de línea de comandos. El tipo **void** significa que no devuelve ningún valor. El siguiente programa, por ejemplo, escribe todos los argumentos de línea de comandos, encerrados entre corchetes.

```
public class EscribeArgs {  
  
    public static void main(String[] args) {
```

```
for(String arg: args) {  
    System.out.printf("[%s] ", arg);  
}  
  
System.out.println("");  
}  
}
```

### Actividad 3.5

\*

¿Se podría crear, en el mismo programa anterior, un método con el mismo nombre `suma` que el ya existente, y que tenga dos parámetros de tipo `double` y devuelva un valor de tipo `double`? Intenta crearlo y utilizarlo en un programa. Asegúrate de pasarle valores del tipo apropiado para que se use el nuevo método y no el antiguo.

### Actividad 3.6

\*

Crea un programa que lea dos números reales y utilice un método `suma` para obtener su suma. Este método es igual que el creado para el ejercicio anterior pero suma dos números de tipo `double` en lugar de dos números de tipo `int`.

### Actividad 3.7

\*

Utiliza el método `suma` para calcular la suma de 4 con todos los números del 10 al 20 (es decir: 4+10, 4+11, ..., 4+20).

### Actividad 3.8

\*

Crea un método al que se le puedan pasar dos valores de tipo `double` y un operador. El operador podrá ser "+", "-", "\*" ó "/". El método devolverá un valor de tipo `double` que, dependiendo del operador, será la suma, diferencia, producto o división de los dos operadores.

Si se pasa al método algún otro valor para el operador, debe devolver `Double.NaN`, que indica que la operación que se ha pedido no tiene sentido y, por tanto, no se puede dar ningún resultado válido (NaN significa "Not a Number").

Se podría probar el método con un código como este:

```
System.out.println(calcular(6.4, "+", 6));  
System.out.println(calcular(3, "-", 1.2));  
System.out.println(calcular(4, "*", 54.56));  
System.out.println(calcular(6.4, "/", 6));
```

## 6. Métodos que no devuelven valores (`void`)

Un método que se define con tipo `void` no devuelve ningún valor. El siguiente programa utiliza un método `escribeRepetido` de tipo `void` que no devuelve ningún valor. Escribe una cadena de caracteres dada un número de veces dado. Puede escribir o no un salto de línea después. La cadena de caracteres que hay que escribir, el número de veces que se escribe, y la opción de escribir o no un salto de línea al final se pasan como argumentos en el momento de realizar la llamada, y dentro del método, están disponibles en parámetros.

```
public class Prueba {
```

```

static void escribeRepetido(String cad, int n, boolean saltoLinea) {
    for (int i = 0; i < n; i++) {
        System.out.print(cad);
    }
    if (saltoLinea) {
        System.out.println("");
    }
}

static void main(String[] args) {

    int ancho = 10;

    escribeRepetido("-", ancho, true);

    for (int i = 2; i < ancho; i++) {
        escribeRepetido("|", 1, false);
        escribeRepetido(".", ancho-2, false);
        escribeRepetido("|", 1, true);
    }

    escribeRepetido("-", ancho, true);
}
}

```

**Actividad 3.9**

\*

Crea un programa que utilice el método `escribeRepetido` del ejemplo anterior para escribir lo siguiente:

```

*
**
***
****
*****

```

**Actividad 3.10**

\*\*

Crea un método `escribeCuñaIzq` que utilice el método `escribeRepetido` para escribir figuras como la anterior, pero con un tamaño dado por un parámetro de tipo `int`.

Con distintos valores para ese parámetro escribiría lo siguiente. Crea un programa que lo verifique.

1	2	3	4	5	6
*	*	*	*	*	*
	**	**	**	**	**
		***	***	***	***
			****	****	****
				*****	*****
					*****

**Actividad 3.11**

\*

Cambia el método creado en la actividad anterior para que escriba no necesariamente `"*"`, sino cualquier texto que

se le pase en un parámetro.

### Actividad 3.12

\*\*

Crea un método `escribeCuñaDcha` que utilice el método `escribeRepetido` para escribir figuras como las siguientes, que se muestran para distintos valores de un parámetro de tipo `n`.

Ayuda: en cada fila hay que escribir primero una cantidad determinada de espacios y después una cantidad determinada de asteriscos.

1	2	3	4	5	6
*	* **	* ** ***	* ** *** ****	* ** *** **** *****	* ** *** **** ***** *****

### Actividad 3.13

\*\*\*

Crea un método `escribePiramide` que utilice el método `escribeRepetido` para escribir figuras como las siguientes, que se muestran para distintos valores de un parámetro de tipo `n`.

Ayuda: en cada fila hay que escribir primero una cantidad determinada de espacios, después una cantidad determinada de asteriscos, y después la misma cantidad de espacios que al principio.

1	2	3	4	5	6
*	* ***	* *** *****	* *** ***** *****	* *** ***** ***** *****	* *** ***** ***** ***** ***** *****

Crea un método `escribeRombo` que utilice el método `escribeRepetido` para escribir figuras como las siguientes, que se muestran para distintos valores de un parámetro de tipo `n`.

1	2	3	4
*	* *** *	* *** ***** *** *	* *** ***** ***** ***** *** *

### Actividad 3.14

\*

Cambia alguno de los métodos anteriores para que devuelva el número de caracteres impresos.

## 7. Encapsulación de algoritmos en métodos

Los métodos estáticos sirven para encapsular algoritmos, de manera que se puedan ejecutar fácilmente con distintos valores para las entradas.

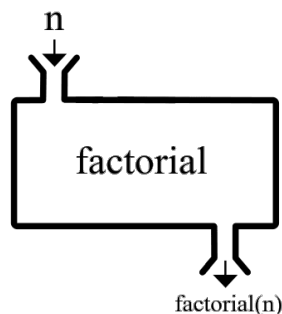
Por ejemplo, el siguiente programa calcula el factorial del número 5. Este valor está almacenado previamente en una variable `int n`.

```
public static void main(String[] args) {  
  
    int n = 5;  
  
    long fact = 1;  
  
    int i = n;  
    while(i > 1) {  
        fact *= i;  
        i--;  
    }  
  
    System.out.printf("!\%d = \%d\n", n, fact);  
}
```

Se puede crear un método para el cálculo del factorial. Este se puede invocar repetidamente para calcular el factorial de distintos números. El siguiente programa muestra el factorial de 5, como el anterior, pero utiliza el método factorial para su cálculo. Y después lo utiliza para el cálculo del factorial de varios otros números.

```
public class PruebaFact {  
  
    static long factorial(int n) {  
        long fact = 1;  
        int i = n;  
        while (i > 1) {  
            fact *= i;  
            i--;  
        }  
        return fact;  
    }  
  
    public static void main(String[] args) {  
  
        int n;  
  
        n = 5;  
        System.out.printf("!\%d = \%d\n", n, factorial(n));  
  
        n = 20;  
        System.out.printf("!\%d = \%d\n", n, factorial(n));  
  
        System.out.printf("!\%10 = \%d\n", factorial(10));  
  
    }  
}
```

Un método se puede ver como una caja negra que encapsula un algoritmo. Desde otros métodos (por ejemplo, el método `main`, pero puede ser cualquier método) se puede utilizar el algoritmo llamando al método con valores para las entradas, y obteniendo el valor de salida.



Algoritmo	Método de Java
<b>Algoritmo: factorial</b> Entradas: <code>int n</code> <pre> long fact = 1; int i = n; while(i &gt; 1) {     fact = fact * i;     i--; } </pre> Salida: <code>long fact</code>	<pre> long factorial(int n) {     long fact = 1;     int i = n;     while( i &gt; 1) {         fact *= i;     }     return fact; } </pre>

### Actividad 3.15

\*

Crea un programa que escriba los factoriales de los números del 1 al 20, utilizando el método **factorial**. La salida del programa debe ser de la siguiente forma:

```

0! = 1
1! = 1
2! = 2
3! = 6
...
20! = 2432902008176640000

```

### Actividad 3.16

\*

Crea un procedimiento `salario` que devuelva el salario semanal de un trabajador dado su tipo de salario y el número de horas que ha trabajado durante la semana.

Si el salario es de tipo A, se pagan 6€ por hora, hasta 40 horas, y a partir de ellas a 9€ por hora.

Si el salario es de tipo B, se pagan 7€ por hora, hasta 40 horas, y a partir de ellas a 9,5€ por hora.

Si el salario es de tipo C, se pagan 8€ por hora, hasta 40 horas, y a partir de ellas a 10€ por hora.

Utilízalo para calcular el salario de trabajadores de tipos A, B y C, tanto con horas extras (más de 40 horas trabajadas en la semana) como sin ellas (40 o menos horas trabajadas en la semana).

Hacer que funcione con el siguiente código en el método `main`.

```

System.out.println(salario("A", 30));
System.out.println(salario("A", 46));
System.out.println(salario("B", 40));
System.out.println(salario("B", 50));
System.out.println(salario("C", 38));
System.out.println(salario("C", 52));

```

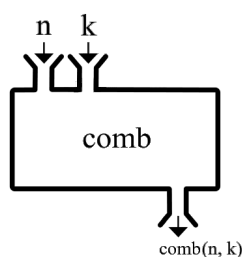
## Actividad 3.17

\*\*

Crea un programa que lea un número entero de la entrada estándar. Si no se introduce un número entero o se introduce un número entero negativo, debe escribir un mensaje de error y terminar su ejecución. En otro caso, debe escribir el factorial del número introducido.

La lectura y validación del número entero debe hacerse igual que en los programas de ejemplo del tema anterior.

Para calcular el valor de un número combinatorio  $\binom{n}{k} = \frac{n!}{k!(n-k)!}$  se puede utilizar el método `factorial`. Se puede crear un método `comb` para ello, que a su vez utilice el método `factorial`.



Algoritmo	Función de Java
<b>Algoritmo: comb</b> Entradas: <code>int n, int k</code> <hr/> <code>comb = fact(n) / fact(k) / fact(n-k)</code> Salida: <code>long comb</code>	<pre>long comb(int n, int k) {     return fact(n) / fact(k) /         fact(n-k); }</pre>

## Actividad 3.18

\*

Crea un programa que lea un número entero de la entrada estándar. Si no se introduce un número entero o se introduce un número entero negativo, debe escribir un mensaje de error y terminar su ejecución. En otro caso, debe escribir los números combinatorios `comb(n, 0)`, `comb(n, 1)`, `comb(n, 2)`, ..., `comb(n, n)`, donde `n` es el número que se ha introducido.

Por ejemplo: si se introduce 5, debe escribir: 1 5 10 10 5 1

## Actividad 3.19

\*\*

Implementar de manera más eficiente el método `comb`.  $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ , pero también se puede calcular de la siguiente manera:  $\binom{n}{k} = \frac{n \cdot (n-1) \cdot (n-2) \dots (n-k+1)}{k!}$ . Es decir, el producto de `k` números empezando desde `n` y

hacia abajo de uno en uno, dividido por el factorial de `k`. Por ejemplo:  $\binom{7}{3} = \frac{7 \cdot 6 \cdot 5}{3 \cdot 2 \cdot 1} = 35$ , es decir, el producto de 3 números, empezando desde 7 y hacia abajo, dividido por el producto de 3 números, de 3 a 1.

Se sugiere

Probar esta implementación con el código del método `main` de alguna de las actividades anteriores, y verificar que se obtienen los mismos resultados. Se puede copiar el código de prueba para todos ellos en el método `main`, y ejecutarlo para verificar que se obtienen los mismos resultados que con la anterior implementación.

## 8. Métodos que devuelven cadenas de caracteres (**String**)

No solo se pueden utilizar los métodos para realizar cálculos numéricos. Un método puede devolver un valor de cualquier tipo. Por ejemplo, el siguiente método devuelve una cadena compuesta de un número de palotes dado en un parámetro.

```
static String palotes(int n) {
    String palotes = "";
    for(int i=0; i<n; i++) {
        palotes += "|";
    }
    return palotes;
}
```

**Actividad 3.20**

\*

Crea un método al que se le pase un número entero y que devuelva en un `String` una flecha formada por caracteres. La dirección de la flecha será hacia la derecha o hacia la izquierda dependiendo del signo del número, y la longitud vendrá dada por su valor absoluto. Se muestra en la tabla lo que debe devolver el método para distintos valores del parámetro.

-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6
<-----	<----	<---	<--	<-	<		>	->	-->	--->	---->	----->

**Actividad 3.21**

\*

Crea un método al que se le pase una cadena de caracteres en un `String` y devuelva la misma cadena al revés. Es decir: si se le pasa "método", devolverá "odotém". Debe comenzarse con una cadena vacía (`String res = ""`). En cada iteración del bucle se le debe concatenar al final un nuevo carácter de la cadena, que se puede obtener con `res.charAt(posición)` o `res.substring(posición_inicial, pos_final)`.

**Actividad 3.22**

\*\*

Crea un método al que se le pase una cadena de caracteres en un `String` y devuelva el valor booleano `true` si la cadena es un palíndromo, y el valor booleano `false` si no lo es. Un palíndromo es una cadena de caracteres que es igual leída del derecho que del revés. Por ejemplo: "reconocer". Se puede utilizar el método creado en el apartado anterior y después devolver `true` si ambas cadenas son iguales, y `false` en caso contrario. Se puede saber si dos cadenas `str1` y `str2` son iguales con `str1.equals(str2)`.

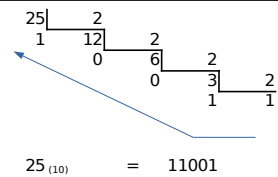
**Actividad 3.23**

\*\*\*

Crea un método al que se le pase un número entero y devuelva el número binario equivalente, utilizando un bucle `while`. El número se debe devolver en forma de `String`.

El método debe funcionar con números negativos. Si para 6 devuelve 110, para -6 debe devolver -110. Se explica el algoritmo para números positivos.

Se divide repetidamente por 2 en un bucle. En cada iteración del bucle se obtiene un nuevo dígito decimal en el resto y se continúa con el cociente, mientras sea mayor que 1. El número equivalente será un 1 seguido de todos los restos que se han ido obteniendo, en orden inverso.





## Actividad 3.24

\*\*\*

Utilizar el método `comb` para crear un método `String binNewton(String a, String b, int n)` que devuelva el desarrollo de  $(a+b)^n$  de acuerdo a la fórmula del binomio de Newton, donde  $a$  y  $b$  son cadenas de caracteres que pueden representar expresiones o números, y  $n$  es un número natural.

$$(a+b)^n = \binom{n}{0} a^n b^0 + \binom{n}{1} a^{n-1} b^1 + \binom{n}{2} a^{n-2} b^2 + \dots + \binom{n}{n-1} a^1 b^{n-1} b^1 + \binom{n}{n} a^0 b^n$$

Por ejemplo:  $(a+b)^3 = \binom{3}{0} a^3 b^0 + \binom{3}{1} a^2 b^1 + \binom{3}{2} a^1 b^2 + \binom{3}{3} a^0 b^3$

Se muestra un código de prueba para el método `main` y el resultado que se espera para su ejecución.

```
System.out.println(binNewton("a", "b", 1, false));
System.out.println(binNewton("a", "b", 2, false));
System.out.println(binNewton("x", "y", 3, false));
System.out.println(binNewton("a", "(-b)", 4, false));
1*a^1*b^0+1*a^0*b^1
1*a^2*b^0+2*a^1*b^1+1*a^0*b^2
1*x^3*y^0+3*x^2*y^1+3*x^1*y^2+1*x^0*y^3
1*a^4*(-b)^0+4*a^3*(-b)^1+6*a^2*(-b)^2+4*a^1*(-b)^3+1*a^0*(-b)^4
```

(Posibles mejoras). no escribir los factores 1, ni las potencias con exponente 0, dado que son iguales a 1 ( $a^0=1$ ). No escribir los exponentes 1 ( $a^1=a$ ). Con esto, para los ejemplos anteriores se obtendría:

```
a+b
a^2+2*a*b+b^2
a^3+3*a^2*b+3*a*b^2+b^3
a^4+4*a^3*(-b)+6*a^2*(-b)^2+4*a*(-b)^3+(-b)^4
```

Hacer todo esto utilizando el operador ternario `expr ? valorSi : valorSiNo`.

## 9. Paso de *arrays* a métodos

No hay ningún inconveniente en pasar un *array* como argumento a un método en Java. El *array* debe pasarse para un parámetro definido con el tipo apropiado. En el siguiente ejemplo se pasa un *array* de números enteros a un método `sumaArray` que devuelve la suma de los elementos del *array*.

```
package operarray;

public class OperArray {

    public static long sumaArray(int[] nums) {
        long suma = 0;
        for (int num: nums) {
            suma += num;
        }
        return suma;
    }

    public static void main(String[] args) {
        int[] arr = {1, 4, -2, 0, 6, -5, 3};

        System.out.printf("Suma: %d.\n",
            sumaArray(arr));
    }
}
```

## Actividad 3.25

\*\*

Haz los cambios necesarios en el programa anterior para que la función `sumaArray` devuelva la suma de un array de `double`, y se pruebe el método con una llamada desde el método `main`. El método debe devolver un valor de tipo `double`.

## 9.1. Métodos que devuelven *arrays*

También se pueden devolver *arrays* desde métodos. La clase de ejemplo vista anteriormente se completa con un método que recibe un *array* en un parámetro, y que devuelve un *array*. Cuando se devuelve un *array*, este ha de crearse dentro del método utilizando **new**.

```
package operarray;

public class OperArray {

    // Devuelve suma de todos los elementos de un array
    public static long sumaArray(int[] nums) {
        long suma = 0;
        for (int num: nums) {
            suma += num;
        }
        return suma;
    }

    // Devuelve array con n enteros, de 0 a n-1
    public static int[] rangoHasta(int n) {
        int[] rango = new int[n];
        for (int i = 0; i < n; i++) {
            rango[i] = i;
        }
        return rango;
    }

    // Devuelve array con n enteros, de 0 a n-1
    public static void escribeArray(int[] nums) {
        System.out.print("{");
        if (nums.length > 0) {
            System.out.print(nums[0]);
        }
        for (int i = 1; i < nums.length; i++) {
            System.out.printf(", %d", nums[i]);
        }
        System.out.print("}");
    }

    public static void main(String[] args) {

        int[] arr = {1, 4, -2, 0, 6, -5, 3};

        System.out.print("arr: ");
        escribeArray(arr);
        System.out.println();

        System.out.printf("Suma: %d.\n", sumaArray(arr));

        int n = 10;
    }
}
```

```
int[] rango = rangoHasta(10);
System.out.printf("rango de 0 a %d: ", n-1);
escribeArray(rango);
System.out.println();

}

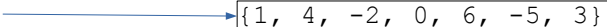
}
```

### El valor `null`

Una variable de tipo *array* contiene una referencia a una posición de la memoria en la que están los datos del *array*. Por ejemplo, cuando se ejecuta el siguiente código:

```
int[] arr = {1, 4, -2, 0, 6, -5, 3};
```

A la variable `arr` se le asigna una referencia a una posición de memoria en la que están los contenidos del *array* y alguna información adicional para uso interno, como por ejemplo la longitud.

`arr`      

Incluso si el *array* se crea con longitud 0, con `int[] arr = {}`, se reserva una zona de memoria para él, y `arr` contiene una referencia a ella. `arr` es una referencia a un *array* que existe, aunque su longitud sea 0.

A una variable de tipo *array* se le puede asignar el valor `null`. Este es un valor especial que significa que no existe el *array*, es decir, que no apunta a ninguna dirección de memoria en la que se puedan encontrar sus contenidos.

Un método que devuelve un *array* puede devolver el valor `null` para indicar que no tiene sentido el *array* que se solicita.

Por ejemplo: `rangoHasta(-5)` podría devolver `null`, porque el método devuelve un *array* cuyos elementos van de 0 al número solicitado, de uno en uno y en orden creciente.

### Actividad 3.26

Añade a la clase de ejemplo anterior un método `sumaArrays` que devuelva la suma de dos *arrays*. Si cualquiera de ellos es `null` o tienen distintas longitudes, el método debe devolver el valor `null`.

### Actividad 3.27

Prueba qué pasa cuando a los métodos de la clase de ejemplo anterior `OperArray` se les pasa algún *array* con valor `null`. Haz los cambios necesarios para gestionar adecuadamente estos casos y verifícalos.

El valor `null` se suele utilizar para representar algún valor no existente o no relevante en una determinada situación. Piensa algún caso en que podría tener sentido que alguno de los métodos anteriores devolviera el valor `null`. Cambia el método en consecuencia y verifica su funcionamiento en ese caso.

## 9.2. Modificación y copia de los contenidos de un *array*

Se pueden modificar los contenidos de un *array* en un método al que se le pasa el *array* en un parámetro. El siguiente método suma un número entero a todas las posiciones de un *array*.

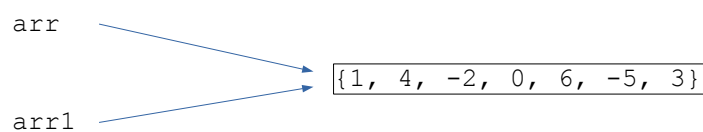
```
static void sumaEntero(int[] arr, int num) {
    for (int i = 0; i < arr.length; i++) {
        arr[i] += num;
    }
}
```

El siguiente programa utiliza este método para sumar 4 a todas las posiciones de un *array*, y el método `escribeArray` para mostrar sus contenidos antes y después de realizar esta operación.

<pre>int[] arr = {1, 4, -2, 0, 6, -5, 3}; escribeArray(arr); System.out.println(); sumaEntero(arr, 4); escribeArray(arr); System.out.println();</pre>	<pre>{1, 4, -2, 0, 6, -5, 3} {5, 8, 2, 4, 10, -1, 7}</pre>
---	--

Al método `escribeArray` se le pasa una referencia al *array* `arr`. Cualquier variable de un tipo no básico contiene una referencia a un objeto. Y los *arrays*, en Java, son un tipo particular de objetos.

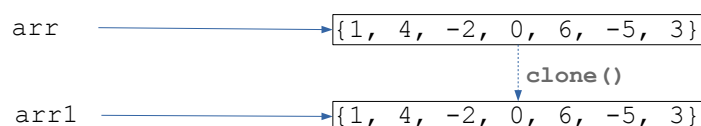
Con la sentencia `int[] arr1 = arr` no se crea un nuevo *array* `arr`, sino que se asigna a `arr1` una referencia al mismo objeto que `arr`.



Por ello, como se puede comprobar con el siguiente programa, las operaciones hechas con `arr1` se pueden ver en `arr`, porque ambas variables contienen una referencia al mismo objeto.

<pre>int[] arr = {1, 4, -2, 0, 6, -5, 3}; int[] arr1 = arr; sumaEntero(arr1, 4); escribeArray(arr); System.out.println(); escribeArray(arr1); System.out.println();</pre>	<pre>{5, 8, 2, 4, 10, -1, 7} {5, 8, 2, 4, 10, -1, 7}</pre>
---	--

En otros casos puede interesar no modificar los contenidos en el *array* original, sino en una copia, dejando el *array* original inalterado. Se puede hacer una copia de un *array* con el método `clone`, como en el siguiente ejemplo. Con `clone` se crea un objeto nuevo al que referencia `arr1`, y `arr` sigue apuntando al mismo objeto, que queda inalterado.



<pre>int[] arr = {1, 4, -2, 0, 6, -5, 3}; int[] arr1 = arr.clone(); sumaEntero(arr1, 4); escribeArray(arr); System.out.println(); escribeArray(arr1); System.out.println();</pre>	<pre>{1, 4, -2, 0, 6, -5, 3} {5, 8, 2, 4, 10, -1, 7}</pre>
---	--

## Actividad 3.28

\*\*

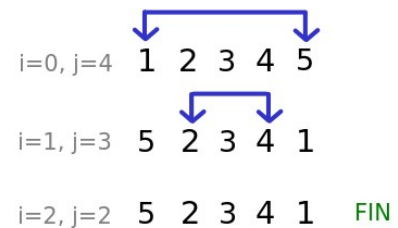
Añade a la clase de ejemplo anterior `OperArray` un método `invertir` que invierta los contenidos de un `array`. Si se le pasa, por ejemplo, un `array` cuyos contenidos son {1, 2, 3}, debe cambiar sus contenidos para que sean {3, 2, 1}. El método debe realizar los cambios sobre el propio `array`, no crear y devolver un `array` nuevo.

Asegúrate de que el método hace lo apropiado cuando se le pasa un `array null`.

En el tema anterior se planteó como ejercicio un programa que hiciera esto mismo. Ahora se pide crear un método para ello.

Se puede utilizar un bucle en el que se mantienen dos índices, cada uno de los cuales hace referencia a una posición del `array`. El primero empieza con valor 0, haciendo por tanto referencia a la primera posición del `array`. El segundo empieza haciendo referencia a la última posición del `array`. En cada iteración se intercambian los elementos del `array` referenciados por ambos índices, se incrementa el valor del primero, y se decrementa el valor del último. El bucle se ejecuta mientras el valor del primero sea menor que el valor del último.

Contenido inicial: 1 2 3 4 5



## Actividad 3.29

\*

En el método `main`, escribe un programa de prueba para la clase `OperArray` que, utilizando métodos estáticos de esta clase,

1. Crea un `array` con los números del 4 al 24.
2. Crea una copia de este `array`, utilizando `clone`.
3. Da la vuelta a los contenidos de esta copia.
4. Escribe el `array` original y su copia, que tendrá los mismos contenidos pero al revés.

## 10. Sentencias avanzadas para control de bucles: `break` y `continue`

Las sentencias `break` y `continue` se utilizan para simplificar la programación de algunos problemas que se resuelven con un bucle. No son estrictamente necesarias. Es decir, que cualquier programa que utiliza una de estas sentencias se podría reescribir para hacer lo mismo sin ellas. Pero sí son muy convenientes para crear programas más simples y, por tanto, más legibles.

Supongamos que se quiere escribir un programa que determina si un número entero `num` mayor o igual que 2 es primo. Lo será si no tiene ningún divisor entre 2 y `num-1`. El siguiente programa utiliza un bucle de `div=2` a `num-1` para buscar divisores de un número entero `num`. Se asume que el número es primo (`esPrimo = true`) mientras no se encuentre un divisor, y si se encuentra se asigna `false` a `esPrimo`.

```
int num = new Scanner(System.in).nextInt();

boolean esPrimo = true;
for(int div = 2; div < num; div++) {
    if(num % div == 0) {
        esPrimo = false;
    }
}
System.out.printf("%d primo: %b\n", num, esPrimo);
```

Nota: En realidad, no hace falta buscar divisores hasta `num-1`, ni siquiera hasta `num/2`, pero se deja así el bucle por simplicidad.

Pero basta con encontrar un divisor para llegar a la conclusión de que el número no es primo. El bucle podría perfectamente terminar justo después de la asignación `esPrimo = false`. Se puede modificar el programa para que así sea, añadiendo la condición `esPrimo` para continuar con el bucle. Esto mejora mucho la eficiencia del programa, al evitar iteraciones innecesarias del bucle.

#### Mejora 1: guarda en el bucle (condición adicional para seguir con su ejecución)

```
int num = new Scanner(System.in).nextInt();

boolean esPrimo = true;
for(int div = 2; div < num && esPrimo; div++) {
    if(num % div == 0) {
        esPrimo = false;
    }
}
System.out.printf("%d primo: %b\n", num, esPrimo);
```

Pero se puede también utilizar la **sentencia break** para terminar inmediatamente la ejecución de un bucle. Esto simplifica el programa.

#### Mejora 2: sentencia break para salir del bucle

```
int num = new Scanner(System.in).nextInt();

boolean esPrimo = true;
for(int div = 2; div < num; div++) {
    if(num % div == 0) {
        esPrimo = false;
        break;
    }
}
System.out.printf("%d primo: %b\n", num, esPrimo);
```

La sentencia `break` se puede utilizar con cualquier tipo de bucle.

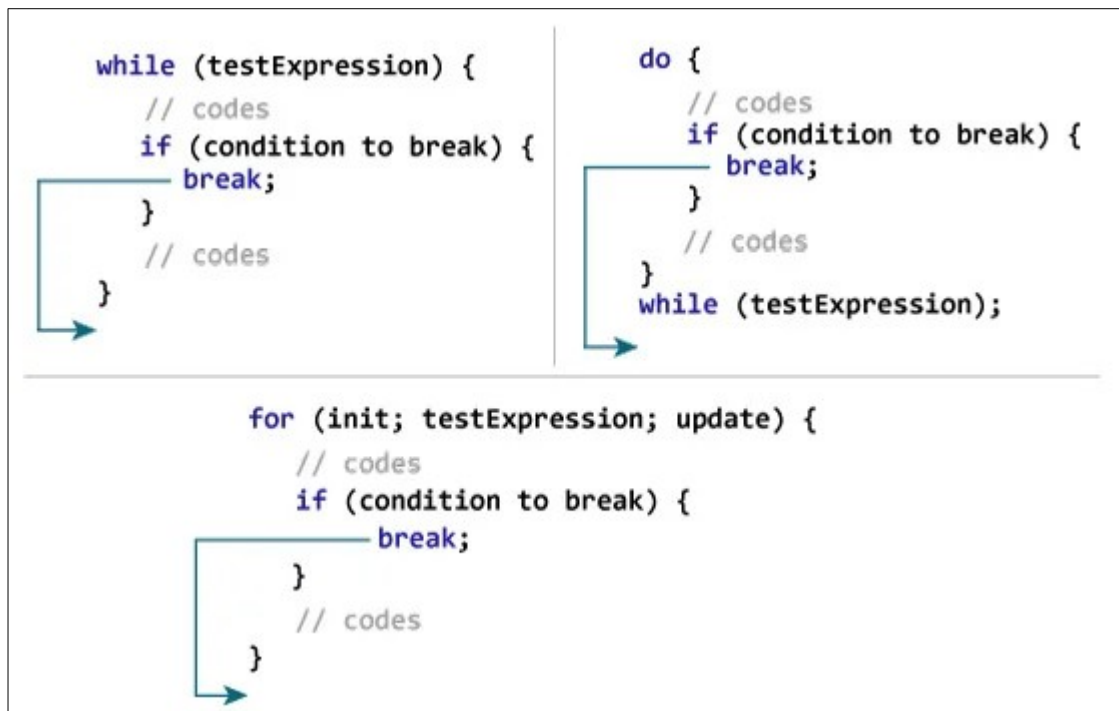


Figura 3.1: Sentencia **break** para bucles

(fuente: <https://www.programiz.com/java-programming/break-statement>)

### Actividad 3.30

\*\*

Crea un programa que verifique si un *array* de números enteros está ordenado. Utiliza una sentencia **break** para terminar en cuanto un número sea menor que el anterior. El programa debe escribir un mensaje indicando si el *array* esta ordenado o no. Si no lo está, debe indicar los dos números que rompen la ordenación y en qué posición están.

Ayuda: utilizar un bucle para obtener los valores en todas las posiciones del *array*, de la primera a la última. Utilizar una variable `previo` que tendrá el valor de la anterior posición. Al final de cada iteración del bucle, se le asignará el valor de la posición actual. Antes de la primera iteración se le puede asignar el valor de la primera posición del *array*, dado que su utilidad es detectar cualquier valor que sea menor que el anterior.

La sentencia **continue** hace que se pase inmediatamente a la siguiente ejecución del bucle.

Se podría mejorar la eficiencia del programa anterior con una sentencia **continue** para evitar dividir por números pares mayores de 2. El número 2 es primo. Si un número mayor que 2 es divisible por 2, entonces no es primo, y entonces no será tampoco divisible por ningún número par, lo que permite evitar la mitad de divisiones. Se puede utilizar una variable booleana `par` que cambie de valor con cada iteración del bucle, para evitar dividir por 2.

### Mejora 2: sentencia **continue** para evitar algunas ejecuciones del bucle

```

int num = new Scanner(System.in).nextInt();

boolean esPrimo;

if (num == 2) {
    esPrimo = true;
} else if (num % 2 == 0) {
    esPrimo = false;
} else {

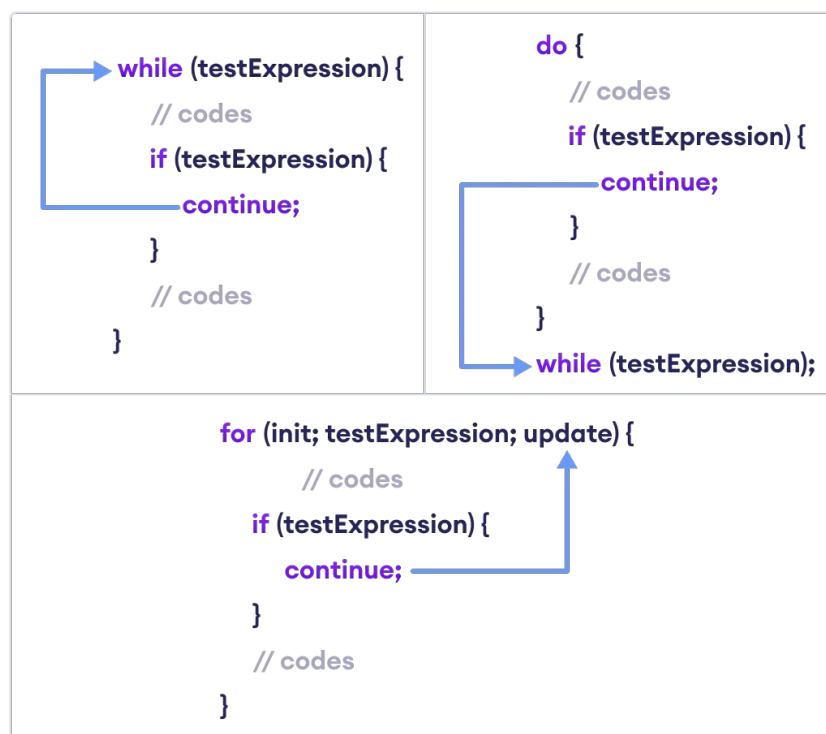
```

```
esPrimo = true
boolean par = true;
for (int div = 3; div < num; div++) {
    par = !par;
    if (par) {
        continue;
    }
    if (num % div == 0) {
        esPrimo = false;
        break;
    }
}

System.out.printf("%d primo: %b\n", num, esPrimo);
```

(Por cierto, hubiera sido más sencillo incrementar aumentar `div` de dos en dos en el bucle `for`: `div += 2` en lugar de `div++`).

La sentencia `continue` se puede utilizar también con cualquier tipo de bucle.



**Figura 3.2: Sentencia `continue` para bucles**

(fuente: <https://www.programiz.com/java-programming/continue-statement>)

Las sentencias `break` y `continue` se pueden utilizar para un tipo de programas muy habitual, los que leen una entrada tras otra de la entrada estándar y realizan alguna acción en base a la entrada introducida. Normalmente estos programas permiten terminar la ejecución cuando se introduce un valor particular (para



esto se puede utilizar `break`). Y muchas veces no hacen nada cuando se introducen determinados valores (para esto se puede utilizar `continue`).

El siguiente programa, por ejemplo, muestra la longitud de cada cadena de caracteres que introduce el usuario. Las cadenas vacías se ignoran. Cuando se introduce un punto, termina la ejecución del programa.

El bucle en sí es un bucle sin fin, pero termina con `break` cuando el usuario introduce un punto. Cuando se introduce una línea vacía, se ejecuta directamente `continue` para leer y gestionar la siguiente.

```
Scanner s = new Scanner(System.in);

while(true) {

    System.out.print("Introducir cadena: ");

    String linea = s.nextLine();

    if(linea.length() < 1) {
        continue;
    }

    if(linea.equals(".")) {
        break;
    }

    System.out.printf("Longitud: %d\n", linea.length());
}
```

Es una práctica habitual utilizar bucles sin fin, como el anterior (`while(true) { ... }`) con sentencias `break`. También se pueden crear bucles sin fin de tipo `for y do ... while`.

Bucles sin fin		
<code>while(true)</code> { } }	<code>for(;;)</code> { } }	<code>do</code> { } } <code>while(true)</code>

Por supuesto, debe asegurarse de que el usuario tiene la oportunidad de terminar la ejecución, si se trata de un proceso interactivo o, si no lo es, que siempre se acabará ejecutando la sentencia `break`.

### Actividad 3.31

\*

Crea un programa que lea un número entero tras otro. El programa debe mostrar un `prompt` o texto antes de quedar a la espera de leer un número entero. El programa terminará cuando se introduzca un cero. Si se introduce algo que no sea un número entero, se mostrará un mensaje de error y se pasará inmediatamente a leer el siguiente número. Se puede utilizar para ello el método `nextInt()` de `Scanner`, capturando las excepciones de tipo `InputMismatchException` que se puedan producir si no se introduce un número entero. Si se introduce un número entero positivo, se debe mostrar la suma de todos los números positivos introducidos hasta el momento. Si se introduce un número entero negativo, el programa mostrará un mensaje indicando que se ha introducido un número negativo y pasará inmediatamente a leer el siguiente. Cuando se introduce cero, el programa mostrará la suma de todos los números introducidos y su media aritmética, por supuesto con decimales, si los tiene.

## 11. Bucles anidados

Un bucle anidado es un bucle dentro de otro.

Ya se ha visto algún ejemplo. El siguiente programa muestra las tablas de multiplicar del 2 al 9.

<pre>public static void main(String[] args) {     for (int i = 1; i &lt; 10; i++) {         for (int j = 1; j &lt; 10; j++) {             System.out.printf("%2d ", i*j);         }         System.out.println();     } }</pre>	<pre>1  2  3  4  5  6  7  8  9 2  4  6  8 10 12 14 16 18 3  6  9 12 15 18 21 24 27 4  8 12 16 20 24 28 32 36 5 10 15 20 25 30 35 40 45 6 12 18 24 30 36 42 48 54 7 14 21 28 35 42 49 56 63 8 16 24 32 40 48 56 64 72 9 18 27 36 45 54 63 72 81</pre>
---	--

En este ejemplo, ambos bucles son independientes. Se podría cambiar el programa para que solo se mostraran productos donde el primer factor (i) es mayor o igual que el segundo (j).

<pre>public static void main(String[] args) {     for (int i = 2; i &lt; 10; i++) {         for (int j = 1; j &lt;= i; j++) {             System.out.printf("%2d ", i*j);         }         System.out.println();     } }</pre>	<pre>1 2  4 3  6  9 4  8 12 16 5 10 15 20 25 6 12 18 24 30 36 7 14 21 28 35 42 49 8 16 24 32 40 48 56 64 9 18 27 36 45 54 63 72 81</pre>
---	--

El siguiente programa de ejemplo ordena los contenidos de un *array* de números aleatorios. Para ello, intercambia el valor de la primera posición por cualquier valor menor que haya en el *array* en posiciones posteriores. Después hace lo mismo con la segunda posición, y sucesivamente con todas, hasta llegar a la penúltima. Para ello utiliza dos bucles anidados, el exterior con i tomando valores desde el índice del primer elemento (0) hasta el índice del penúltimo elemento (la longitud del *array* menos 2), y el interior tomando valores de i+1 al índice del último elemento (la longitud del *array* menos 1).

```
Random r = new Random();

int TAM_ARRAY = 20;
int[] nums = new int[TAM_ARRAY];

// Rellena array con valores al azar de 0 a 99
for (int i = 0; i < TAM_ARRAY; i++) {
    nums[i] = r.nextInt(100);
}

// Ordena array
for (int i = 0; i < nums.length - 1; i++) {
    for (int j = i + 1; j < nums.length; j++) {
        if (nums[i] > nums[j]) { // Si nums[i] > nums[j] Intercambia nums[i] y nums[j]
            int aux = nums[i];
            nums[i] = nums[j];
            nums[j] = aux;
        }
    }
}
```

```

    }
}

for (int i = 0; i < TAM_ARRAY; i++) {
    System.out.printf("%d ", nums[i]);
}
System.out.println("");

```

El siguiente programa de ejemplo utiliza el método de la criba de Eratóstenes para encontrar los números primos menores que uno dado. Se muestra primero el algoritmo y después el programa que lo implementa en Java.

Se marcarán los números que no son primos. En principio no hay ninguno marcado como no primo.

Se consideran uno a uno los números a partir de 2. Es decir, se hace un bucle desde `candidato=2` hasta `candidato=MAX_PRIMO`, sumando 1 a `candidato` cada vez. En este ejemplo, `MAX_PRIMO` es 100. En realidad, no hace falta seguir mientras `candidato < MAX_PRIMO`, basta con seguir mientras `candidato < Math.sqrt(MAX_PRIMO)`.

El primer candidato para ser considerado como primo es 2. No está marcado. Se deja sin marcar y se marcan como no primos sus múltiplos.

Es decir, se marcan como no primos desde `mult=candidato+candidato`, sumando `mult` cada vez, mientras `mult<MAX_PRIMO`.

El siguiente candidato para ser considerado como primo es 3. Se hace igual: se deja sin marcar y se marcan como no primos sus múltiplos.

El siguiente candidato es 4. Como ya está marcado, se pasa al siguiente (`continue`). Con ello se pasa al 5. Este no está marcado como no primo, por lo que se marcan sus múltiplos como no primos.

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

El siguiente candidato es 6. Como ya está marcado, se pasa al siguiente (`continue`), que es 7. Este no está marcado como no primo, por lo que se marcan sus múltiplos como no primos.

Se pasa sucesivamente a los siguientes candidatos hasta encontrar uno no marcado (con `continue`). Se pasa a 8, 9, y 10, que están marcados, y se llega a 11, no marcado. Este número ya es mayor que la raíz cuadrada de 100, que es 10. Por tanto, ya se han marcado todos los números no primos menores o iguales que 100.

Quedan sin marcar los números primos menores o iguales que 100.

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

```
final int MAX_PRIMO = 1000;

boolean noPrimo[] = new boolean[MAX_PRIMO];
// Inicialmente, todos a false.
// Esto significa que no se ha demostrado que no son primos.

for (int candidato = 2; candidato < Math.sqrt(MAX_PRIMO); candidato++) {
    if (noPrimo[candidato]) {
        continue; // Ya marcado como no primo, probar con siguiente
    }
    // candidato es primo, marcar sus múltiplos como no primos
    for (int mult = candidato + candidato; mult < MAX_PRIMO; mult += candidato) {
        noPrimo[mult] = true;
    }
}

for (int i = 2; i < MAX_PRIMO; i++) { // Escribir los no marcados como primos.
    if (!noPrimo[i]) {
        System.out.print(i + " ");
    }
}
```

## 12. Clases y objetos

En Java cabe distinguir entre tipos básicos y clases. Los tipos básicos tienen valores elementales que se almacenan en memoria en uno o varios bytes. En Java son `int`, `long`, `short`, `float`, `double`, `byte`, `char`, `boolean`.

Los objetos pertenecen a una clase. O dicho de otra forma, un objeto es una instancia de una clase a la que pertenece. Una clase incluye tanto datos como métodos, que son operaciones que se ejecutan sobre una instancia de la clase.

Ya se han utilizado clases. Por ejemplo, la clase `String`. Y se han utilizado sus métodos `length` (para obtener la longitud de un `String`) y `charAt` (para obtener el carácter que hay en una posición de un `String`). Esta es una clase perteneciente a la biblioteca de clases estándares de Java. Estas están disponibles para su uso en cualquier programa. Pero también se pueden crear nuevas clases. Pero se pueden crear nuevas clases.

Se muestra a continuación la definición de una clase de ejemplo. Más adelante se verán las clases en detalle. Por ahora solo se quiere mostrar un ejemplo para tener una idea básica de su estructura y funcionamiento y cómo se definen y utilizan sus métodos.

Esta es la definición de una clase `Personaje`.

```
class Personaje {

    // miembros de datos
    private final String nomPers;
    private int x;
    private int y;
```

```

// constructor
public Personaje(String nombre, int x, int y) {
    this.nomPers = nombre;
    this.x = x;
    this.y = y;
}

// métodos
public String getNomPers() {
    return nomPers;
}

public int getX() {
    return x;
}

public int getY() {
    return y;
}

public void avanzaArriba(int numPasos) {
    y -= numPasos;
}

public void avanzaDerecha(int numPasos) {
    x += numPasos;
}

public void avanzaAbajo(int numPasos) {
    y += numPasos;
}

public void avanzaIzquierda(int numPasos) {
    y -= numPasos;
}
}

```

Los nombres tanto de clases como de variables siguen la convención de los nombres en forma de joroba de camello. Las jorobas serían las iniciales de palabras, que están siempre en mayúsculas. La primera letra es un caso particular. Está en mayúsculas en los nombres de clases, y en minúsculas en los nombres de variables. Algunos ejemplos:

Nombres de clases	Nombres de variables
Personaje	Personaje pers
MedioTransporte	Personaje personaje
	Personaje unPersonaje
	MedioTransporte avion1
	MedioTransporte unAvion
	MedioTransporte elMelillero
	MedioTransporte elBarcoDelArroz

## 12.1. Constructores y variables de instancia

Las variables de instancia de una clase tienen un valor distinto para cada objeto o instancia de la clase. En la clase `Personaje` del ejemplo anterior, son `nomPers`, `x` e `y`.

Los constructores son un tipo especial de métodos. Se invocan en el momento en que se crea un objeto o instancia de una clase. Tienen el mismo nombre de la clase, y pueden tener parámetros. En este caso particular, pero bastante típico por lo demás, lo único que hace el constructor es asignar valores a los miembros de datos de la clase a partir de valores pasados como parámetros al constructor. Para diferenciar entre el parámetro `x` del constructor y el miembro de datos `x` de la clase, este último se representa como `this.x`. Esto significa la variable de instancia `x` del propio objeto sobre el que se ejecuta el método, `this`.

```
class Personaje {  
  
    // miembros de datos  
    private final String nomPers;  
    private int x;  
    private int y;  
  
    // constructor  
    public Personaje(String nombre, int x, int y) {  
        this.nomPers = nombre;  
        this.x = x;  
        this.y = y;  
    }  
    (...)  
}
```

Cuando una variable se define como `final`, su valor no se puede modificar una vez que se asigna por primera vez. Es el caso de la variable de instancia `nomPers`. Este es el nombre del personaje, y no se puede cambiar, una vez que se le asigna un valor en el constructor. El valor de las variables `x` e `y`, en cambio, sí se puede modificar, y se hace en los métodos `avanzaArriba`, `avanzaDerecha`, `avanzaAbajo` y `avanzaIzquierda`.

## 12.2. Creación de objetos

Para construir un objeto de la clase, o una instancia de la clase, se utiliza `new`. Con ello se crea un objeto de la clase y se ejecuta un constructor sobre él. Con el siguiente ejemplo, se crean varios objetos de la clase `Personaje`.

```
Personaje gato = new Personaje("Jynx", 5, 3);  
Personaje raton1 = new Personaje("Pixy", 10, 2);  
Personaje raton2 = new Personaje("Dixy", 20, 14);
```

## 12.3. El valor `null`

El valor `null` es un valor especial que se puede asignar a cualquier variable de una clase cualquiera, que representa un objeto no existente o no conocido.

También se puede asignar este valor a una variable de tipo `array`. Esto es porque, de hecho, las variables son objetos en Java.

Cuando se intenta acceder a una variable de instancia o a un método de un objeto con valor `null`, se produce una excepción de la clase `NullPointerException`. Por ello, es conveniente verificar previamente su valor, como en el siguiente ejemplo.

```
int[] nums;
(...)
if(nums == null) {
    System.out.println("Array null");
} else {
    System.out.printf("Longitud del array: %d\n", nums.length);
}
```

## 12.4. Métodos

Los métodos de una clase se ejecutan sobre un objeto de la clase. Ya se ha hablado de un tipo particular de métodos, los constructores, que se ejecutan cuando se crea una instancia de una clase.

Un método puede devolver un valor de un tipo determinado que se especifica antes del nombre del método. Para ello se usa una sentencia `return` con el valor que devuelve el método. Un método para el que se especifica un tipo `void` no devuelve ningún valor, y en él no se puede utilizar la sentencia `return`.

<pre>public int getX() {     return x; }</pre>	<pre>public void avanzaDerecha(int numPasos) {     x += numPasos; }</pre>
--	---

La clase anterior incluye varios métodos cuyo nombre empieza por `get`, y que devuelven cada uno el valor de una variable de instancia o miembro de datos, pero no modifican el valor de ninguna variable de instancia del objeto. Este es un tipo frecuente de métodos, conocidos como *getters*. En el contexto particular de estos métodos, el nombre de la variable de instancia no se puede confundir con ningún parámetro ni ninguna variable local del método. Por ello no se antepone `this` a su nombre, pero podría hacerse. El *getter* para la variable de instancia `x` incluiría, entonces, la sentencia `return this.x`.

Los métodos *getter* anteriores permiten obtener información acerca del objeto, y en particular acerca de su estado. El estado de un objeto viene dado por los valores de sus variables de instancia. En este caso son `nombre`, `x` e `y`.

Los métodos `avanzaArriba`, `avanzaDerecha`, `avanzaAbajo` y `avanzaIzquierda` se declaran con tipo `void` y, por tanto, no devuelven ningún valor. Por otra parte, sí modifican los valores de variables de instancia.

## 12.5. Acceso a variables de instancia y métodos de una clase

Para referirse a algo de una clase, sean una variable de instancia o un método, se añade un punto y el nombre de la variable de instancia o del método. Se muestra a continuación un ejemplo en el que se crea un objeto de la clase `Personaje` y se ejecutan varios de sus métodos. Una vez creado el objeto, se obtienen y muestran las coordenadas de la posición inicial, se realiza un movimiento y se obtienen y muestran las coordenadas de la nueva posición.

```
Personaje gato = new Personaje("Jynx", 5, 3);
int posX = gato.getX();
int posY = gato.getY();

System.out.printf("Posición inicial: (%d, %d)\n", posX, posY);

gato.avanzaDerecha(1);
```

```
System.out.printf("Posición final: (%d, %d)\n", gato.getX(), gato.getY());
System.out.printf("Posición final: (%d, %d)\n", gato.x, gato.y);
```

Lo único que no funciona del código anterior es la última sentencia, en la que se accede directamente a las variables de instancia `x` e `y`. No se puede acceder a ellas desde fuera de la clase porque se han declarado como **private**. Si no se hubiera hecho así, se podría no solo obtener el valor de esas variables de instancia, lo que podría no ser tan grave, sino también modificarlos, lo que sí sería indeseable. Porque se quiere que solo se pueda cambiar la posición de la manera en que lo hacen los métodos `avanzaArriba`, `avanzaDerecha`, `avanzaAbajo` y `avanzaIzquierda`.

En cambio, los métodos que permiten consultar el estado de un objeto de la clase y modificarlo se declaran como **public**. Con esto se permite el acceso a ellos desde fuera del propio objeto.

### Actividad 3.32

Añade a la clase `Personaje` un método `avanza(Direccion pCard, int numPasos)` que mueva el personaje un número de pasos determinados hacia el punto cardinal dado. Utiliza el tipo enumerado `Direccion` mostrado en un programa de ejemplo anterior. En el método `main`, crea un programa de prueba que cree un objeto en una posición inicial, muestre la posición inicial, mueva el objeto varias veces utilizando los métodos anteriores (utiliza cada uno al menos una vez), y por último muestre la nueva posición. La trayectoria debe venir en un `array` de elementos del tipo enumerado `Direccion`, que representa una secuencia de movimientos.

## 12.6. Variables de clase y métodos estáticos

Cuando en una clase se define una variable o un método como **static**, no es una variable o un método de instancia, sino de clase. Una variable estática no tiene un valor diferente para cada instancia de la clase, sino un valor único para ella. En otras palabras, su valor es un atributo de la clase, y no de ninguna instancia particular suya. Un método estático o de clase solo puede acceder a variables o métodos estáticos o de clase.

La clase `Integer` de la biblioteca estándar de clases de Java tiene algunas variables de clase o **static**, como se puede ver en su documentación en <https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Integer.html>.

### Field Summary

#### Fields

Modifier and Type	Field	Description
static final int	<b>BYTES</b>	The number of bytes used to represent an int value in two's complement binary form.
static final int	<b>MAX_VALUE</b>	A constant holding the maximum value an int can have, $2^{31}-1$ .
static final int	<b>MIN_VALUE</b>	A constant holding the minimum value an int can have, $-2^{31}$ .
static final int	<b>SIZE</b>	The number of bits used to represent an int value in two's complement binary form.
static final Class<Integer>	<b>TYPE</b>	The Class instance representing the primitive type int.

Algunas de estas variables estáticas o de clase son:

<b>static final int</b> <code>MIN_VALUE</code>	Mínimo valor que puede tomar el valor entero representado por una instancia de la clase.
<b>static final int</b> <code>MAX_VALUE</code>	Máximo valor que puede tomar el valor entero representado por una instancia de la clase.
<b>static final int</b> <code>BYTES</code>	Número de bytes utilizados para representar un entero.

Estos valores se definen en variables estáticas porque son atributos de la clase y no de ninguna instancia suya particular. Además se definen como **final**, es decir, que su valor no puede cambiar, de manera que son constantes de clase.



se. Sus nombres, siguiendo las reglas de nomenclatura habituales en Java, tienen todas las letras en mayúsculas y con las palabras separadas por guiones bajos.

Para hacer referencia a ellas, dado que son variables de clase y no de instancia, se utiliza el nombre de la clase seguido de un punto, como se muestra en el siguiente ejemplo.

```
System.out.println("Los números enteros que se pueden representar en Java están  
entre %d y %d.\n", Integer.MIN_INT, Integer.MAX_INT);
```

También se pueden definir métodos como **static**. Estos no se ejecutan sobre ningún objeto de la clase. En ellos, por tanto, no se puede utilizar el identificador **this**. Y como ya se ha dicho, solo pueden utilizar variables y métodos estáticos.

La clase `Integer` tiene también algunos métodos estáticos. Por ejemplo, los que permiten obtener un valor entero a partir de su representación en forma de cadena de caracteres. Son estáticos porque el valor que devuelven solo depende de los valores que se les pasan en sus parámetros, y no del estado de ninguna instancia particular de la clase.

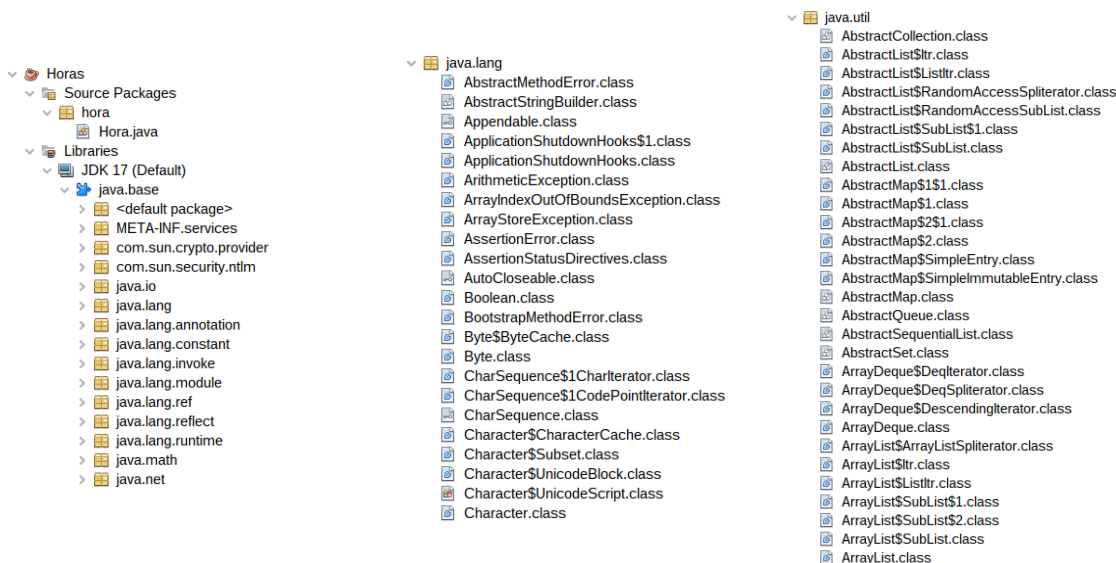
<code>static int parseInt(String s)</code>	Devuelve el número entero que representa un <code>String</code> .
<code>static int parseUnsignedInt(String s)</code>	Devuelve el número entero sin signo que representa un <code>String</code> .
<code>static int parseInt(String s, int radix)</code>	Devuelve el número entero que representa un <code>String</code> . El método anterior del mismo nombre asume que el número está en base 10. Con este método se puede indicar la base de numeración.

Se hace referencia a un método estático con el nombre de la clase seguido de un punto y del nombre del método, como se muestra en el siguiente ejemplo.

```
int n = Integer.parseInt("81");
```

### 13. Biblioteca estándar de clases de Java

La biblioteca estándar de clases de Java forma parte del JDK de Java y está estructurada en una jerarquía de paquetes. Esta jerarquía se puede visualizar dentro de cualquier proyecto, en el apartado “Libraries”, dentro del JDK utilizado, y dentro de él en el módulo “java.base”. Dentro de cada paquete se pueden ver las clases incluidas en él.



## Directivas `import`

Para utilizar en una clase otra clase de un paquete distinto al de la propia clase (lo que es siempre el caso cuando se uti-

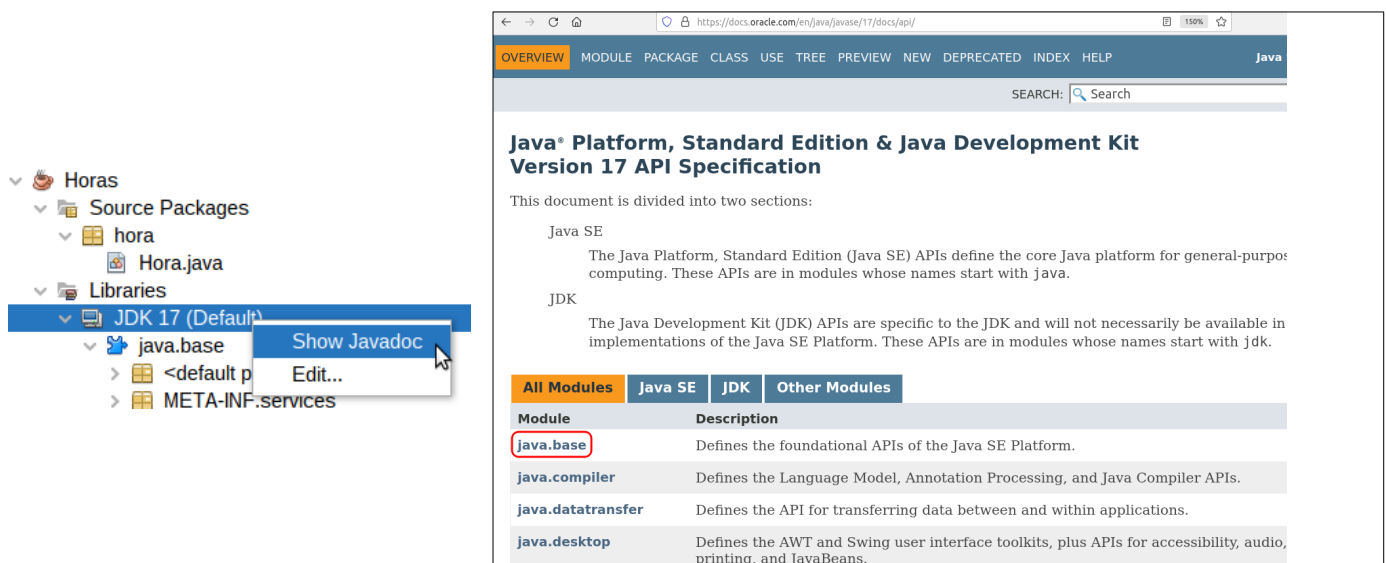
liza una clase de la biblioteca estándar de clases de Java), hay que incluir al principio una directiva `import`.

Ya se han visto algunas en los programas de ejemplo:

```
import java.util.InputMismatchException
import java.util.Scanner
```

Esto no es necesario para las clases dentro de `java.lang`, donde están clases básicas del lenguaje, como por ejemplo `String` y las clases *wrapper* para los tipos básicos: `Integer`, `Long`, `Short`, `Float`, `Double`, `Byte`, `Character` y `Boolean`.

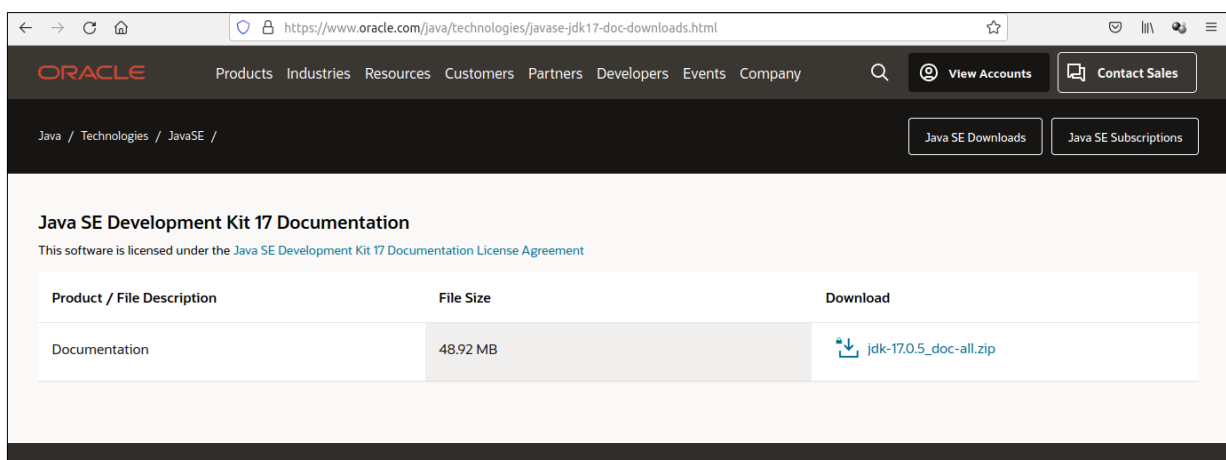
La biblioteca estándar de clases de Java está documentada en un conjunto de documentos en formato HTML, disponibles públicamente en Internet. Estos son los llamados Javadocs. Pulsando con el botón derecho del ratón, se puede mostrar esta documentación oficial. En este ejemplo, se puede ver que corresponde a la versión del JDK utilizada en el proyecto, la 17. La documentación de las clases de la biblioteca estándar de clases de Java está dentro de la del módulo `java.base`.



The screenshot shows an IDE's project explorer on the left with a tree structure: 'Horas' > 'Source Packages' > 'hora' > 'Hora.java' and 'Libraries' > 'JDK 17 (Default)' > 'java.base'. A right-click context menu is open over 'java.base', showing options 'Show Javadoc' and 'Edit...'. On the right, a web browser displays the 'Java Platform, Standard Edition & Java Development Kit Version 17 API Specification' page. The page includes a search bar, a table of modules, and a description of the 'java.base' module.

Module	Description
<b>java.base</b>	Defines the foundational APIs of the Java SE Platform.
java.compiler	Defines the Language Model, Annotation Processing, and Java Compiler APIs.
java.datatransfer	Defines the API for transferring data between and within applications.
java.desktop	Defines the AWT and Swing user interface toolkits, plus APIs for accessibility, audio, printing, and JavaBeans.

También se pueden descargar los Javadocs para consultarlos en local, sin necesidad de conexión a Internet. Para ello se puede hacer la búsqueda, por ejemplo, por texto “Javadocs Java 17 download”. Esto conducirá, seguramente, a la página web <https://www.oracle.com/java/technologies/javase-jdk17-doc-downloads.html>.



The screenshot shows the Oracle website page for downloading Java SE Development Kit 17 Documentation. The page includes the Oracle logo, navigation links, and a table with download information.

Product / File Description	File Size	Download
Documentation	48.92 MB	<a href="#">jdk-17.0.5_doc-all.zip</a>

Se puede descargar el fichero y entonces se tienen los siguientes contenidos dentro del nuevo directorio que se crea. Abriendo `index.html` en un navegador, se tiene acceso a los mismos contenidos disponibles en la web, pero en local, sin necesidad de acceder a Internet.

**Java Platform, Standard Edition & Java Development Kit Version 17 API Specification**

This document is divided into two sections:

- Java SE**  
The Java Platform, Standard Edition (Java SE) APIs define the core Java platform for general-purpose computing. These APIs are in modules whose names start with java.
- JDK**  
The Java Development Kit (JDK) APIs are specific to the JDK and will not necessarily be available in all implementations of the Java SE Platform. These APIs are in modules whose names start with jdk.

All Modules	Java SE	JDK	Other Modules
<b>Module</b>	<b>Description</b>		
java.base	Defines the foundational APIs of the Java SE Platform.		
java.compiler	Defines the Language Model, Annotation Processing, and Java Compiler APIs.		
java.datatransfer	Defines the API for transferring data between and within applications.		
java.desktop	Defines the AWT and Swing user interface toolkits, plus APIs for accessibility, audio, imaging, printing, and JavaBeans.		

## 14. La clase String para cadenas de caracteres

Los objetos de la clase `String` contienen cadenas de caracteres.

La longitud en caracteres del `String` se puede obtener con el método `length()`.

La clase `String` de Java almacena internamente las cadenas de caracteres en codificación UTF-16. Esta es una codificación de Unicode. Otras codificaciones de Unicode son UTF-8 y UTF-32.

Se puede consultar la documentación de esta clase buscando por su nombre. Cuando se introduce `String` salen muchos resultados. Pero interesa la clase `java.lang.String`. El prefijo `java.lang` indica que es una clase de la biblioteca estándar de clases de Java.

**Classes and Interfaces**

- java.lang.**String**
- javax.print.DocFlavor.**STRING**
- java.lang.**StringBuffer**
- java.io.**StringBufferInputStream**
- java.lang.**StringBuilder**
- java.text.**StringCharacterIterator**
- java.lang.invoke.**StringConcatException**
- java.lang.invoke.**StringConcatFactory**
- javax.swing.text.**StringContent**
- java.lang.**StringIndexOutOfBoundsException**
- java.util.**StringJoiner**
- javax.management.monitor.**StringMonitor**
- javax.management.monitor.**StringMonitorMBean**
- java.io.**StringReader**
- javax.naming.**StringRefAddr**
- com.sun.jdi.**StringReference**

Hay diversas maneras de crear un `String`. La más directa ya se ha utilizado en muchos ejemplos:

```
String saludo = "Hola";
```

También se puede crear un objeto de tipo `String` con `new`, aunque no se suele hacer de esta manera.

```
String saludo = new String("Hola");
```

Y también se puede crear a partir de un *array* de *char*:

```
char data[] = {'H', 'o', 'l', 'a'};
String str = new String(data);
```

Si dentro de la cadena de caracteres aparece el carácter ' ' (comillas dobles) debe anteponérsele una barra invertida (\), para indicar que se trata de este carácter, y no del final de la cadena. Por ejemplo:

```
string saludo = "Dijo \"hasta pronto\", pero no lo hemos vuelto a ver.";
```

Con la clase *String* se puede utilizar el operador + para concatenar cadenas, como en el siguiente ejemplo.

```
String nombre = "Titanio";
String simbolo = "Ti";
int numAtomico = 22;
double masaAtomica = 47.867;
String descElem = "Elemento " + nombre + " (" + simbolo + "), número atómico: "
    + numAtomico + ", masa atómica: " + masaAtomica;
```

Se puede utilizar el truco que se muestra a continuación para convertir datos de diversos tipos a un *String*.

```
int numAtomico = 22;
double masaAtomica = 47.867;
String numAtomicoS = "" + numAtomico;
String masaAtomicaS = "" + masaAtomica;
```

Una vez que se sabe crear objetos de tipo *String* por distintos medios, se plantea la cuestión de la igualdad entre distintos *String*.

Primero conviene precisar algunos términos y poner en cuestión algunas posibles ideas preconcebidas.

#### Operador == para tipos básicos y objetos

Con tipos básicos como *int*, *double* o *float*, el operador == devuelve *true* siempre que las dos cosas que se comparan tengan el mismo valor. Es decir, comprueba la **igualdad** de las dos cosas que se comparan.

<pre>int a = 2; int b = 2;  System.out.printf("2 == 2: %b\n", 2 == 2); System.out.printf("a == 2: %b\n", a == 2); System.out.printf("a == b: %b\n", a == b);</pre>	<pre>2 == 2: true a == 2: true a == b: true</pre>
--	---

Las cosas que se comparan en las sentencias de ejemplo anteriores (las variables *a* y *b* y el literal 2) son iguales, es decir, tienen igual valor, pero no son la misma cosa, es decir, no son idénticas. Las variables *a* y *b* ocupan distintos lugares en la memoria, y el valor literal 2 ocupa otro lugar en la memoria o en un registro del microprocesador.

También se puede utilizar el operador == con objetos de la clase *String*.

<pre>String a = "hola"; String b = "hola"; String c = new String("hola");  System.out.printf("\"hola\" == \"hola\": %b\n", "hola" == "hola"); System.out.printf("a == \"hola\": %b\n", a == "hola"); System.out.printf("a == b: %b\n", a == b); System.out.printf("c == \"hola\": %b\n", c == "hola"); System.out.printf("a == c: %b\n", a == c);</pre>	<pre>"hola" == "hola": true a == "hola": true a == b: true c == "hola": false a == c: false</pre>
---	---

En este caso particular, podría esperarse que el operador == devolviera siempre *true* para comparaciones entre estas variables y el literal "hola". Pero no siempre es así. No lo es, en particular, con la variable *c*.

Eso es porque el operador ==, cuando se utiliza con objetos (de la clase *String* en los ejemplos anteriores), comprueba la **identidad**, y no la igualdad de los objetos. Es decir, comprueba que son el mismo objeto, que están en la misma posición de memoria.

Si se quiere verificar la igualdad entre objetos, hay que utilizar el **método equals ()** en lugar del operador ==.

```
String a = "hola";
String b = "hola";
String c = new String("hola");
```

```
System.out.printf("%b\n", "hola".equals("hola"));
System.out.printf("%b\n", a.equals("hola"));
System.out.printf("%b\n", a.equals(b));
System.out.printf("%b\n", c.equals("hola"));
System.out.printf("%b\n", a.equals(c));
```

```
true
true
true
true
true
```

Pero entonces, ¿Por qué el operador `==` devuelve `true` con las comparaciones entre `a`, `b` y `"hola"`? Las variables de tipo `String` son en realidad referencias a un lugar de la memoria en el que está almacenado un objeto de esta clase. Cuando se asigna un valor a una variable de tipo `String`, la máquina virtual de Java, en general, no crea un nuevo objeto `String` en memoria si ya existe uno con el mismo valor, sino que le asigna una referencia al que ya existe. Pero no se puede confiar en que siempre sea o vaya a seguir siempre siendo así. Por lo tanto, para comparar el valor de dos `String`, siempre hay que utilizar el método `equals()`.

Algunos de los métodos más interesantes de la clase `String` son los siguientes. Se ejecutan sobre un `String` dado, que en adelante se denominará “el `String`”, “la cadena” o `this`.

**Cuadro 3.1: Métodos de la clase `String`**

<code>int length()</code>	Devuelve la longitud en caracteres (no en bytes).
<code>char charAt(int index)</code>	Devuelve el carácter ( <code>char</code> ) en la posición dada ( <code>index</code> ). La primera posición es 0.
<code>int compareTo(String anotherString)</code>	Compara el <code>String</code> con otro dado ( <code>anotherString</code> ). Devuelve: <ul style="list-style-type: none"> <li>• Un número positivo si es mayor en orden alfabético.</li> <li>• Un número negativo si es menor en orden alfabético.</li> <li>• 0 si son iguales</li> </ul>
<code>int compareToIgnoreCase(String str)</code>	Igual que el anterior, pero no diferencia entre mayúsculas y minúsculas.
<code>boolean contains(String str)</code>	Devuelve: <ul style="list-style-type: none"> <li>• <code>true</code> si <code>this</code> contiene a <code>str</code>, es decir, si <code>str</code> es una subcadena de <code>this</code>.</li> <li>• <code>false</code> en caso contrario.</li> </ul>
<code>boolean startsWith(String prefix)</code>	Devuelve: <ul style="list-style-type: none"> <li>• <code>true</code> si <code>this</code> comienza con <code>prefix</code></li> <li>• <code>false</code> en caso contrario.</li> </ul>
<code>boolean endsWith(String suffix)</code>	Devuelve: <ul style="list-style-type: none"> <li>• <code>true</code> si <code>this</code> termina con <code>suffix</code>.</li> <li>• <code>false</code> en caso contrario.</li> </ul>
<code>boolean equals(Object anObject)</code>	Ya se ha hablado de este método. Devuelve: <ul style="list-style-type: none"> <li>• <code>true</code> si <code>anObject</code> es un <code>String</code> y es igual (no necesariamente idéntico) a <code>this</code>.</li> <li>• <code>false</code> en caso contrario.</li> </ul>
<code>boolean equalsIgnoreCase(Object anObject)</code>	Lo mismo, pero sin diferenciar entre mayúsculas y minúsculas.
<code>int indexOf(int ch)</code>	Varios métodos que devuelven la primera posición en que aparece un carácter <code>ch</code> o un <code>String str</code> . Los que tienen el parámetro <code>fromIndex</code> buscan a partir de la posición dada por ese índice. El índice 0 representa la primera posición. Si no aparece, devuelven -1.
<code>int indexOf(int ch, int fromIndex)</code>	
<code>int indexOf(String str)</code>	
<code>int indexOf(String str, int fromIndex)</code>	
<code>int lastIndexOf(int ch)</code>	Varios métodos que devuelven la última posición en que aparece un carácter <code>ch</code> o un <code>String str</code> . Los que tienen el parámetro <code>fromIndex</code> buscan hacia atrás a partir de la posición dada por ese índice. El índice 0 representa la primera posición. Si no aparece, devuelven -1.
<code>int lastIndexOf(int ch, int fromIndex)</code>	
<code>int lastIndexOf(String str)</code>	
<code>int lastIndexOf(String str, int fromIndex)</code>	

<code>boolean isEmpty()</code>	Devuelve: <ul style="list-style-type: none"> <li>• <code>true</code> si la cadena tiene longitud 0. Es decir, si es igual a <code>""</code>.</li> <li>• <code>false</code> en caso contrario.</li> </ul>
<code>boolean isBlank()</code>	Devuelve: <ul style="list-style-type: none"> <li>• <code>true</code> si la cadena solo tiene caracteres que representan un espacio en blanco.</li> <li>• <code>false</code> en caso contrario.</li> </ul> <p>El carácter más habitual que representa un espacio en blanco es <code>' '</code>, pero no es el único en Unicode. Para más información se puede consultar la documentación del método <code>public static boolean isWhitespace(int codePoint)</code> de la clase <code>Character</code>.</p>
<code>String repeat(int count)</code>	Devuelve el resultado de concatenar el <code>String</code> el número de veces dado por <code>count</code> .
<code>String replace(char oldChar, char newChar)</code>	Devuelve un <code>String</code> resultante de sustituir todas las ocurrencias del <code>oldChar</code> por el <code>newChar</code> .
<code>String replace(CharSequence target, CharSequence replacement)</code>	Devuelve un <code>String</code> resultante de sustituir la primera ocurrencia de la cadena de caracteres <code>target</code> por la cadena de caracteres <code>replacement</code> . Se puede pasar un <code>String</code> para ambos parámetros.
<code>String replaceAll(String regex, String replacement)</code>	Devuelve un <code>String</code> resultante de sustituir todas las ocurrencias de <code>regex</code> por <code>replacement</code> . <code>regex</code> puede ser una cadena literal o una expresión regular. Más adelante se estudiarán las expresiones regulares. Por ahora baste decir que una cadena literal es un caso particular de expresión regular.
<code>String replaceFirst(String regex, String replacement)</code>	Igual que el método anterior, pero solo se sustituye la primera ocurrencia de <code>regex</code> .
<code>String[] split(String regex)</code>	Divide el <code>String</code> en varios, sirviendo cada ocurrencia de <code>regex</code> como separador entre ellos, y devuelve un <i>array</i> con todos ellos. Como ya se ha comentado, una cadena de caracteres literal es un caso particular de expresión regular.
<code>String strip()</code>	Devuelven el <code>String</code> resultante de eliminar los espacios en blanco que pueda tener al principio y al final.
<code>String trim()</code>	Es preferible <code>strip</code> porque, al contrario que <code>trim</code> , está preparado para Unicode. Reconoce como caracteres en blanco no solo <code>" "</code> y los otros que <code>trim</code> reconoce como tales, sino otros tipos de caracteres en blanco de Unicode.
<code>String stripLeading()</code>	Funciona como <code>strip</code> , pero solo elimina espacios en blanco del principio.
<code>String stripTrailing()</code>	Funciona como <code>strip</code> , pero solo elimina espacios en blanco del final.
<code>String substring(int beginIndex)</code>	Devuelve la subcadena que comienza en el índice dado y llega hasta el final.
<code>String substring(int beginIndex, int endIndex)</code>	Devuelve la subcadena que comienza en el primer índice dado y llega hasta el índice anterior al índice dado.
<code>String toLowerCase()</code>	Devuelve el <code>String</code> en minúsculas.
<code>String toUpperCase()</code>	Devuelve el <code>String</code> en mayúsculas.

@PEND. Método `format`. Quizá en cuadro aparte, para relacionarlo con las cadenas de formato de `printf`. Se puede mencionar que los placeholders se pueden referir a argumentos por la posición que ocupan en la lista de argumentos. De esta forma, se puede utilizar más de una vez el mismo sin repetirlo en la lista de argumentos. Dar un ejemplo comparando ambos mecanismos para hacer una misma cosa. Con el segundo mecanismo, se evita repetir argumentos.

**Actividad 3.33**

\*\*

Crea un programa que encuentre todas las ocurrencias de un `String` dentro de otro. Utilizar para ello el método `indexOf(String str, int fromIndex)` en un bucle, en el que se busque la cadena desde un índice o posición cada vez mayor. La primera vez debe llamarse con `fromIndex=0`. Las siguientes veces debe llamarse con la posición siguiente a la devuelta la vez anterior. Se termina cuando se obtiene `-1`, lo que significa que no se ha encontrado el `String`.

**Actividad 3.34**

\*

Obtener la cadena de caracteres (`String`) inversa a una dada. Por ejemplo: la cadena inversa de "ordenador" es "rodanedro". ¿Se te ocurre más de una manera de hacerlo?

**Actividad 3.35**

\*

Averiguar si una cadena es un palíndromo. Un palíndromo es una cadena de caracteres que se lee igual del derecho que del revés. Por ejemplo: "reconocer". Puedes utilizar lo que has hecho para el anterior ejercicio. O puedes hacerlo de manera más eficiente, sin necesidad de crear una cadena de caracteres nueva.

**Actividad 3.36**

\*

Igual que el ejercicio anterior, pero eliminando antes todos los espacios. De esa forma, se reconocerán como palíndromos "se van sus naves" o "yo hago yoga hoy".

**Actividad 3.37**

\*

Como el ejercicio anterior, pero ignorando signos de puntuación y sustituyendo previamente vocales acentuadas por las correspondientes sin acentuar. De esta forma, deberían reconocerse como palíndromos todos los que se muestran en [https://es.wikipedia.org/wiki/Pal%C3%ADndromo#En\\_espa%C3%B1ol](https://es.wikipedia.org/wiki/Pal%C3%ADndromo#En_espa%C3%B1ol). Haz un programa lo más genérico que sea posible. Convierte previamente todo a mayúsculas o minúsculas para que una letra mayúscula y otra minúscula se reconozcan como iguales. Utiliza un `array` de `char` para especificar todos los caracteres que hay que eliminar previamente, donde estarán todos los símbolos de puntuación. Puedes utilizar dos `String` para las correspondencias entre vocales con acentos u otros signos diacríticos y las mismas sin ellos. Por ejemplo: "áéíóúü" y "aeiouu". O bien dos `arrays` de tipo `char`, o dos `arrays` de tipo `String`.

**Actividad 3.38**

\*\*

Crea un programa que sustituya cadenas `%num` por cadenas dentro de un `array` de `String` `strArr`. Debe sustituir `%1` por `strArr[0]`, `%2` por `strArr[1]`, y así hasta `%n` por `strArr[n-1]`, donde `n` es la longitud del `array` `strArr`. Utiliza el método `replace` de la clase `String`.

Por ejemplo: se define inicialmente un `array` `String[] strArr` como sigue:

```
String[] strArr={ "Isabel", "Juan", "estaba muy feliz", "había comprado una bici" }
```

Y una variable `String` `texto` como sigue:

```
String texto="%1 le dijo a %2 que %3. %2 preguntó por qué %3. %1 respondió que %4." }
```

El programa debe sustituir las ocurrencias de `%1`, `%2`, `%3` y `%4` por los correspondientes `String`, para obtener un `String` con el siguiente contenido:

```
"Isabel le dijo a Juan que estaba muy feliz. Juan preguntó por qué estaba muy feliz.
Isabel respondió que había comprado una bici."
```



## 15. La clase Math para operaciones matemáticas

La clase `Math` contiene muchos métodos estáticos (**static**) para realizar muchas operaciones con datos de los distintos tipos básicos numéricos existentes en Java. A saber: `double`, `float`, `double` e `int`, y. Se muestran a continuación algunos de ellos.

		double	float	int	long
<code>abs(x)</code>	Valor absoluto	✓	✓	✓	✓
<code>max(x, y)</code>	Máximo de dos números	✓	✓	✓	✓
<code>min(x, y)</code>	Mínimo de dos números	✓	✓	✓	✓
<code>round(x)</code>	Valor entero más cercano al argumento, <code>long</code> si es un <code>double</code> o <code>int</code> si es un <code>float</code> .	✓	✓		
<code>ceil(x)</code>	Número más próximo por debajo que es igual a un entero.	✓			
<code>floor(x)</code>	Número más próximo por encima que es igual a un entero.	✓			
<code>rint(x)</code>	Valor más cercano al argumento, de tipo <code>double</code> e igual a un número entero.	✓			
<code>signum(x)</code>	Signo: 0, 1.0 o -1.0, según el argumento sea 0, positivo, o negativo.	✓	✓		
<code>sqrt(x)</code>	Raíz cuadrada	✓			
<code>cbrt(x)</code>	Raíz cúbica	✓			
<code>random()</code>	Devuelve un valor aleatorio mayor o igual que 0 y menor que 1.	✓			

### import static

Los métodos estáticos del paquete `Math` se pueden referenciar de varias maneras.

1. Anteponiendo el nombre del paquete. Por ejemplo: `Math.sqrt(2)`.
2. Con **import static** `java.lang.Math.sqrt` al principio, no hace falta anteponer el nombre de la clase en las llamadas a los métodos estáticos. Se podría usar directamente `sqrt(2)`.

Si se usan varios métodos de la clase `Math`, podría ser más cómodo importar estáticamente todos los métodos con **import static** `java.lang.Math.*`. Entonces se pueden utilizar todos los métodos sin anteponerles el nombre de la clase: `sqrt(2)`, `abs(-2)`, `max(a, b)`.

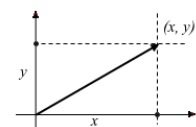
En general, la práctica más habitual es anteponer el nombre de la clase al del método. Por ejemplo: `Math.sqrt(2)`.

### Actividad 3.39

Crea un programa que lea por teclado las coordenadas `x` y `y` de un punto. Ambas pueden ser negativas. Si la coordenada `x` es negativa, el punto está a la izquierda del eje vertical (`y`). Si la coordenada `y` es negativa, el punto está por debajo del eje horizontal (`x`).

Una vez leídos los valores de `x` y `y`, el programa debe calcular y mostrar:

- Distancia al eje `x`, y distancia al eje `y`.
- Distancias máxima y mínima a los ejes. Para ello debe utilizarse un método de la clase `Math`, no una sentencia condicional `if`.
- De qué eje está más cerca el punto, si el eje `x` o el eje `y`.
- Distancia del punto al origen de coordenadas.





El siguiente programa de ejemplo utiliza la generación de números aleatorios para obtener una aproximación del valor del número  $\pi$ .

Para ello calcula muchos puntos (x, y) con valores de x e y al azar entre 0 y 1. Si la distancia del punto al origen de coordenadas es mayor que 1, entonces el punto está fuera del sector circular. En otro caso, está dentro. Con un número grande de puntos elegidos al azar, la proporción de los que caen fuera del área será aproximadamente igual a la proporción con respecto al área total que supone la del sector circular.

```
package aproximapi;

public class AproximaPI {

    public final static int NUM_PUNTOS = 100000000;

    public static void main(String[] args) {

        int nDentro = 0;
        int nTotal = 0;

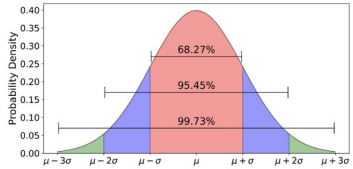
        for (int i = 0; i < NUM_PUNTOS; i++) {
            double x = Math.random();
            double y = Math.random();
            nTotal++;
            if (Math.pow(x, 2) + Math.pow(y, 2) < 1.0f) {
                nDentro++;
            }
        }
        System.out.println(4 * (double) nDentro / nTotal);
    }
}
```

## 16. La clase Random para generación de números aleatorios

La clase Random permite obtener números aleatorios. Los números no son realmente aleatorios, sino que se generan a partir de un valor inicial o semilla. A partir de una misma semilla o valor inicial, se obtiene siempre la misma secuencia de números. Por ello se dice que los números son pseudoaleatorios. Si como semilla o valor inicial se toma uno que dependa de la hora, y que tenga en cuenta hasta los microsegundos, cada vez que se genere una secuencia se obtendrá una secuencia diferente.

**Cuadro 3.2: Métodos de la clase Random**

Random() Random(long seed)	Creadores. El segundo establece una semilla determinada. El primero asigna un valor para la semilla que con mucha probabilidad será distinto que para cualquier otra invocación del constructor.
-------------------------------	--

<pre> int nextInt(int bound) int nextInt(int origin, int bound) int nextInt() boolean nextBoolean() double nextDouble(...) double nextFloat(...) double nextLong(...) </pre>	<p>Métodos que devuelven valores aleatorios del tipo en cuestión, que aparece en su nombre, con una distribución uniforme. Esto último quiere decir que todos los valores en el rango de posibles valores tienen igual probabilidad de ser obtenidos.</p> <p>Si se especifica un límite superior <code>bound</code>, se obtiene un valor entre 0 y ese, pero sin llegar a él. Si se especifica un límite inferior <code>origin</code>, se obtienen valores mayores o iguales que él.</p> <p>Existen métodos <code>nextDouble</code>, <code>nextFloat</code> y <code>nextLong</code> análogos a <code>nextInt</code>.</p> <p><code>nextDouble()</code> y <code>nextFloat()</code> devuelven un número entre 0 y 1.</p> <p><code>nextInt()</code> devuelve un número entre <math>-2^{32}</math> y <math>2^{32}-1</math>.</p>
<pre> void nextBytes(byte[] bytes) </pre>	<p>Rellena un <code>array</code> proporcionado con bytes con valores aleatorios.</p>
<pre> double nextGaussian() double nextGaussian(double mean,     double stddev) </pre>	<p>Devuelven un número con una distribución de probabilidad normal. La probabilidad es más alta para números más cercanos a la media, y disminuye conforme se alejan de ella. Con el primer método, la media <math>\mu</math> (<code>mean</code>) es 0 y la desviación estándar <math>\sigma</math> (<code>stddev</code>) es 1.</p> <p>Con el segundo se puede especificar la media y la desviación estándar.</p> 

**Actividad 3.40**

\*

Escribe un programa que simule la tirada de dos dados de 6 caras repetidamente. Para cada tirada escribirá el número que ha salido en ambos dados. El programa terminará cuando en una tirada salga el mismo número en ambos dados. Al final escribirá el número de tiradas que se han hecho.

**Actividad 3.41**

\*\*

Escribe un programa que escriba 20 números enteros aleatorios entre 1 y 90, separados por espacios. Para terminar, escribirá el máximo, el mínimo y la media aritmética de todos ellos. No utilizar ningún `array` (no es necesario ni supone ninguna ventaja).

**Actividad 3.42**

\*

Escribir un programa que simule 1000 lanzamientos de una moneda imperfecta. La probabilidad de que salga cara será del 55%. Al final debe mostrarse el número de veces que ha salido cara, que debería, normalmente, estar en torno a 550. Ayuda: generar para cada lanzamiento un número al azar entre 0 y 1. Considerar que ha salido cara si el número es menor que 0,55.

**Actividad 3.43**

\*\*

Escribir un programa que genere un número aleatorio entre 1 y 100, que será un número secreto que debe adivinar el usuario. Este introducirá repetidamente un número, hasta que acierte el número secreto. Si el usuario se cansa, puede terminar la partida introduciendo un 0. Para cada número introducido por el usuario, el programa debe decir si el número secreto es menor o mayor. Cuando el usuario acierte el número secreto, debe mostrar el número de intentos realizados hasta adivinar el número secreto.

**Actividad 3.44**

\*\*

Escribir un programa que simule el movimiento de una persona que va a lo largo de un camino con anchura 7. La posición de la persona con respecto a los márgenes del camino viene dada por un número. Inicialmente, está en medio del camino. Con cada paso que da, existe la misma probabilidad de que siga recto, de que se mueva hacia la izquierda, o de que se mueva hacia la derecha. Se puede generar un número entre 0 y 2 para elegir la dirección de avance para el próximo paso. La persona no se puede salir del camino, de manera que si está en un margen e intenta salirse del camino, se moverá hacia delante. Simular el movimiento de la persona durante 20 pasos. Se muestra un ejemplo de la salida del programa desde el inicio de la ejecución y durante varios pasos consecutivos.

```
|  *  |
|  *  |
|  *  |
|  *  |
|  *  |
|  *  |
|  *  |
|  *  |
```

**Actividad 3.45**

\*\*

Igual que el ejercicio anterior, pero en cada punto del camino hay un hoyo en el que puede caer, situado en una posición aleatoria, y entonces el programa termina. Se muestra un ejemplo de ejecución. Se puede hacer que el camino sea más ancho, para que tarde más en caer. Se muestra un ejemplo a continuación.

Escribir un programa que simule el movimiento de una persona que va a lo largo de un camino con anchura 7. La posición de la persona con respecto a los márgenes del camino viene dada por un número. Inicialmente, está en medio del camino. Con cada paso que da, existe la misma probabilidad de que siga recto, de que se mueva hacia la izquierda, o de que se mueva hacia la derecha. La persona no se puede salir del camino, de manera que si está en un margen e intenta salirse del camino, se moverá hacia delante. Simular el movimiento de la persona durante 20 pasos. Se muestra un ejemplo de la salida del programa desde el inicio de la ejecución y durante varios pasos consecutivos. En el último paso no se muestra ni la persona ni el hoyo, que están en la misma posición, sino una X para indicar que la persona ha caído en el hoyo.

```
|  # *  |
|  #   *  |
|   # *  |
|       * #  |
|  #   *  |
|       X  |
```

**Actividad 3.46**

\*\*\*

Igual que el ejercicio anterior, pero la persona tiene tendencia hacia ir hacia uno de los lados. Es decir, que la probabilidad (a elegir) de que vaya hacia este lado será mayor de 1/3. La probabilidad de que vaya recta o hacia el otro lado será igual. Por ejemplo, si hay una probabilidad de un 50% de que vaya hacia uno de los lados, el 50% restante se repartirá entre que vaya recta y que vaya hacia el otro lado. Antes de empezar la ejecución del programa, se pedirá al usuario que introduzca un número entre 0 y 1, que será la probabilidad de que vaya hacia la izquierda de la pantalla (que sería hacia la derecha según su dirección de avance).

@PEND: Clase Date, DateTime, LocalTime.

## 17. Gestión de excepciones

Algunas operaciones realizadas en Java pueden provocar excepciones. Ya se han visto algunos ejemplos.

Cuando se accede a un índice de un *array* *arr* fuera de los límites, es decir, menor que 0 o mayor que *arr.length-1*.

```
int[] arr = {5, 3, 5, 6};
System.out.println(arr[arr.length]);
```

La ejecución del código anterior hace que el programa termine su ejecución y muestre el siguiente mensaje.

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 4 out of
bounds for length 4
    at prueba.Prueba.main(Prueba.java:12)
```

Durante la ejecución del programa se ha producido una excepción del tipo *ArrayIndexOutOfBoundsException* y, como no la ha gestionado, se termina su ejecución.

También se produce una excepción si se utiliza *Integer.parseInt()* con un *String* que no representa un número entero.

```
String[] strArr = {"5", "-4", "dos", "3"};

int suma = 0;
for(String unStr: strArr) {
    suma += Integer.parseInt(unStr);
}

System.out.printf("La suma es: %d\n", suma);
```

Cuando se ejecuta el anterior programa se produce una excepción.

```
Exception in thread "main" java.lang.NumberFormatException: For input string: "dos"
    at
java.base/java.lang.NumberFormatException.forInputString(NumberFormatException.java:67)
    at java.base/java.lang.Integer.parseInt(Integer.java:668)
    at java.base/java.lang.Integer.parseInt(Integer.java:786)
    at prueba.Prueba.main(Prueba.java:18)
```

Leyendo atentamente la información que se muestra, se puede saber el tipo de excepción que se ha producido (*NumberFormatException*), en qué línea del código fuente se ha producido (la línea 18 del fichero *Prueba.java*), y por qué (se ha intentado convertir a un entero el *String* "dos").

Se puede capturar y gestionar esta excepción es con un bloque **try ... catch** alrededor de esta línea de código. Lo que se quiere hacer cuando un *String* del *array* no representa un número es escribir un mensaje informativo y, por lo demás, ignorarlo.

```
String[] strArr = {"5", "-4", "dos", "3"};

int suma = 0;
for(String unStr: strArr) {
    try {
        suma += Integer.parseInt(unStr);
    } catch (NumberFormatException ex) {
        System.out.printf("%s no es un número entero.\n", unStr);
    }
}

System.out.printf("La suma es: %d\n", suma);
```

Cuando en el bloque `try` se produce una excepción de tipo `NumberFormatException`, se ejecuta el bloque `catch`(`NumberFormatException ex`), que escribe un mensaje indicando que el `String` no representa un número entero. `ex` es un objeto de tipo `NumberFormatException` que contiene información acerca de la excepción que se ha producido. En este programa en particular no interesa utilizarla. La salida del anterior programa es la siguiente.

```
dos no es un número entero.  
La suma es: 4
```