2.- Fundamentos de programación 2

Contenidos

1. Tipos de clases

Atendiendo a para qué se utilizan, podemos decir que hay tres tipos de clases.

Clases de programa:

Son clases que, por sí mismas, constituyen un programa que se puede ejecutar. Contienen al método main y, como veremos al estudiar la programación modular, puede contener otros métodos.

Tanto el método main como el resto de métodos que contiene son *métodos de clase* (métodos static).

Clases de librería.

Son clases que contienen una serie de métodos relacionados con alguna temática común (funciones matemáticas, graficos 2D, gráficos3D, etc).

La clase no tiene método main y por tanto no es un programa. Los métodos que contiene son *métodos de clase* (static)

Un ejemplo de este tipo es la clase **Math**, que estudiaremos más adelante en el tema.

Clases de datos.

Permiten representar alguna entidad que aparece en el problema.

Por ejemplo, en un programa para gestionar las compras y ventas de una empresa, podríamos encontrar clases como Cliente, Proveedor, Factura, Albarán, etc. Así, los objetos de la clase Clientes permitirán representar dentro del programa a los clientes de la empresa, con sus datos y comportamiento característicos (nombre, apellidos, tipo de cliente, posibilidad de crédito, etc).

Las clases de programa contienen ***atributos*** (que representan las características de los objetos de la clase) y ***métodos*** (para manipular los atributos). Los métodos pueden ser *de objeto* (no static) o de clase (static).

La clase String, que estudiaremos más adelante, es un ejemplo de clase de datos.

La clase Scanner también es una clase de datos. Representa a la entrada estándar. El objeto de la clase Scanner (tec) almacena una serie de datos (atributos) y dispone de métodos (nextInt( ), nextDouble( ), ...)

2. Tipos de datos

Tipos numéricos: Tamaño y rango de valores.

Recordemos cuales son los tipos primitivos para almacenar datos numéricos en Java

Tipo	Nº bytes	Rango de valores	
		Desde	Hasta
byte	1	-128	127
short	2	-32.768	32.767
int	4	-2.147.483.648	2.147.483.647
long	8	-9.223.372.036.854.775.808	9.223.372.036.854.775.807
float	4	$+/-3,4*10^{-38}$	$+/-3,4*10^{38}$
double	8	$+/- 1,7 *10^{-308}$	$+/- 1,7 *10^{308}$

Tipos numéricos: Literales

Llamamos ***literales*** a los valores constantes que utilizamos

```
int a = 3;  double b = 2.52;
```

Los ***literales de tipo entero*** (byte, short, int y long) pueden expresarse en decimal (base 10), octal (base 8) o hexadecimal (base 16).

```
int a = 15;    // decimal
int a = 017    // octal, empiezan por cero
int a = 0xF    // hexadecimal, empiezan por 0x
```

Por defecto, Java interpreta los literales enteros como de tipo int pero se puede poner la letra L al final para indicar que debe interpretarse como un entero largo (long)

```
long milimetros = 92233720368547758O7L ;
```

Los ***literales de tipo real*** (float o double) solo pueden expresarse en decimal. Existen dos formatos de representación:

- Usando el punto decimal:

```
double a = - 0.000025 ;
```

- Usando notación científica:

```
double a = -2.5E-5;   // -2.5 · 10^-5
```

Por defecto Java interpreta los literales reales como de tipo double, pero puede ponerse la letra F al final para que los interprete como float

```
float d = 1.7E30f ;
```

Los tipos float y double disponen de tres valores especiales, que permiten representar desbordamientos y errores

- Infinity (+infinito)
- –Infinity (-infinito)
- NaN (Not a Number).

Compatibilidad y conversión de tipos

En la asignación (variable = expresion), la variable y la expresión deben ser del **mismo tipo** o de **tipos compatibles**.

Cambio de tipo implícito:

Una expresión se puede asignar a una variable siempre que sea de un tipo de tamaño "menor" que el tipo de la variable:

short -> int -> long -> float -> double

Cuando se realizan este tipo de asignaciones entre variables y expresiones compatibles, Java realiza un ***cambio de tipo implícito*** de la expresión para acomodarlo al tipo de la variable. .

Ejemplos:

```
int a = 10 ;            int i = 2.5;
float f;                byte b = i;
long l = a ;            float f = 34E45;
f = l ;                 (incorrectos)
```

Cambio de tipo explícito:

Java también permite hacer la conversión de tipo de manera explícita, mediante la operación de cambio de tipo llamada ***casting***:

((tipo) expresión)

Por ejemplo

```
// Calcular la media de edad de un grupo de personas
int sumaEdades; // suma de las edades de las personas
int numeroPersonas; //numero de personas que hay
. . .
double media = sumaEdades / (double) numeroPersonas;
```

El tipo char

El tipo char permite representar valores consistentes en un único carácter. Entre los caracteres simples hay:

- letras del alfabeto
- dígitos numéricos
- caracteres especiales

En el siguiente ejemplo se declara una variable de tipo char y se inicializa con el carácter J

char inicialNombre = 'J';

Tablas Unicode i ASCII

Java utiliza la codificación Unicode de caracteres (juego de caracteres internacionales estándar de 16 bits). Al utilizar **dos bytes para almacenar cada**

carácter, existen 65536 posibilidades, suficiente para representar todos los caracteres de todos los lenguajes del planeta.

El estándar ***ASCII/ANSI*** (American Standard Code for Information Interchange / American National Standards Institute) es un subconjunto de Unicode y ocupa las 256 primeras posiciones de la tabla de códigos.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENO	ACK	BEL	BS	TAB	LF	VT	FF	CR	SO	SI
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL
	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127

Conversión char <---> int

Se puede mostrar el código Unicode de un carácter sin más que convertirlo a int. Igualmente, dado un código Unicode se puede obtener el carácter que representa convirtiéndolo a char.

Ejemplos

```
int n;
n = (int)'A'; // n = el código Unicode de 'A'

char c;
c = (char)65; //c = el carácter cuyo Unicode es 65
```

El hecho de que cada carácter lleve asociado un código numérico permite realizar con ellos ciertas ***operaciones aritméticas*** que, en principio,

asociaríamos a valores numéricos. Estas se realizan entre los códigos de los caracteres:

Ejemplos:

¿Cuál es el carácter que sigue a otro?

```
// siguienteLetra será el carácter que sigue a letra
// en la tabla de caracteres Unicode
char letra= 'M';
char siguienteLetra = letra + 1;
```

¿Cuántos caracteres hay entre la 'a' y la 'z'?

int carsEntreAyZ = 'z'-'a'+1

Literales

Los **literales** de tipo char se representan entre comillas simples. Puede ser

- Un carácter: 'a', '@', 'ñ'
- Una "secuencia de escape". Las secuencias de escape sirven para representar caracteres que no tienen una equivalencia en forma de símbolo o que tienen un uso reservado en el lenguaje Java

Las posibles secuencias de escape son:

Secuencia de escape	Significado
'\''	Comilla simple.
'\"'	Comillas dobles.
'\\'	Contrabarra.
'\b'	Backspace .
'\n'	Cambio de línea.
'\r'	Retorno de carro.
'\t'	Tabulador.

Ejemplos:

```
System.out.println( "Este texto se \n muestra en \n tres lineas");
System.out.println ("El acompañante gritó \" frenaaaa \" ")
```

3. La clase Math

Java incluye clases de librería con métodos que realizan tareas concretas relacionadas con alguna temática. La existencia de clases de librería nos permite poder resolver problemas complejos sin tener que hacer todo el trabajo desde cero. Una de esas ***clases de librería*** es Math, que contiene funciones relacionadas con el ámbito de las matemáticas:

- Operaciones matemáticas (ab, ex, …)
- Funciones trigonométricas (seno, coseno, …)
- Redondeo y aproximación.
- etc.

Las clases de librería tienen métodos de clase (métodos static). La manera de llamar (utilizar) una método de clase es anteponer el nombre de la clase y un punto, al nombre del método. Así por ejemplo, para llamar al método sqrt de la clase Math, se utiliza la expresión:

```
Math.sqrt(4) //Calcula la raíz cuadrada de 4
```

Funcionalidad de la clase Math.

Constantes E y PI.

Math tiene definidas dos constantes que representan respectivamente a los números PI y E

- Math.E=2.7182818284590452354
- Math.PI=3.14159265358979323846.

Funciones de redondeo (con x de tipo double) :

ceil(x) : devuelve un double que es el número entero más pequeño que es mayor o igual a x.

```
double x = 879.327;
double ceilx = Math.ceil(x); // ceilx = 880
```

floor(x) : devuelve un double que es el número entero más grande que es menor o igual a x.

```
double x = 879.327;
double floorx = Math.floor(x); // floorx = 879
```

round(x) : devuelve el número entero más próximo a x.

El redondeo de un número double da como resultado un long.

```
long redondeado1 = Math.round(6243.45) // 6243;
```

El redondeo de un número float da como resultado un int

```
int redondeado2 = Math.round(6243.65F); // 6244
```

Funciones trigonométricas:

sin(x) : calcula el seno del ángulo (en radianes) x

cos(x) : calcula el coseno del ángulo (en radianes) x

asin(x) : calcula el arco seno del ángulo x (x entre -1 y 1)

acos(x) : calcula el arco coseno del ángulo x (x entre -1 y 1)

atan(x) : calcula el arco tangente del ángulo x

Otras funciones:

abs(x): *calcula el valor absoluto de x (entero o real)*

exp(x): *calcula* E^x *(El número E elevado a x).*

log(x): *calcula el logaritmo natural de x (x tiene que ser real y no negativo)*

max(x,y): *compara los números x e y (enteros o reales) y devuelve el mayor*

min(x,y): *compara los números x e y (enteros o reales) y devuelve el menor*

pow(x,y) : *calcula* x^y *. No está definida si x <= 0 e y no es entero, ni tampoco si x = 0 e y <= 0*

random(): *genera un número (pseudo)aleatorio entre 0.0 y 1.0 (sin llegar a 1)*

sqrt(x): *calcula la raíz cuadrada positiva de x (x no puede ser negativo)*

Generación de números aleatorios

Math.random() devuelve un double que es un número seudo-aleatorio en el rango [0, 1[

Realizando algunas operaciones, podemos obtener valores aleatorios en otros rangos de valores.

Valor aleatorio en el rango [0, a[

- Math.random ( ) * a

Ejemplos:

  - Math.random() * 50 → [0,50[

Valor aleatorio en el rango [a, b[

- a+ Math.random ( ) * (b-a)

Ejemplos:

  - 10 + Math.random ( ) * 90 → [10, 100[
  - -10 + Math.random ( ) * 10 → [-10, 10[

Valor aleatorio entero en el rango [a, b]

- (int) (a+ Math.random ( ) * (b-a+1))

Ejemplos:

  - (int) (10 + Math.random ( ) * 91) → [10, 100]
  - (int) (Math.random() * 51) → [0,50]

4. La clase String

Se utiliza para representar secuencias de cero o más caracteres. Hasta ahora hemos utilizado literales de tipo String, que sabemos que se ponen entre comillas dobles:

```
System.out.println( "Hola mundo")
//"Hola mundo" es una cadena de caracteres.
```

Pero también es posible almacenar cadenas de caracteres en memoria. Para almacenar cadenas de caracteres en variables se utilizan objetos de la clase String. La clase String se encuentra definida en el paquete java.lang. Las clases de este paquete se pueden utilizar sin necesidad de poner la sentencia import correspondiente.

La forma de definir Strings es la siguiente:

String variable = new String(" texto");

Ejemplo

```
String nombre = new String("Javier");
System.out.println("Mi nombre es " + nombre);
```

Sin embargo, debido a que es una clase que se utiliza ampliamente en los programas, Java permite una forma abreviada de crear objetos String:

String nombreVariable = "texto";

Ejemplo

```
String nombre = "Javier";
System.out.println("Mi nombre es " + nombre);
```

Leer cadenas desde teclado.

Para leer cadenas de caracteres desde teclado podemos utilizar la clase Scanner. Ésta dispone de dos métodos para leer cadenas:

- next(): Lee desde la entrada estándar (teclado) una secuencia de caracteres hasta encontrar un delimitador (un espacio, tabulador, salto de línea, ...). Devuelve un String.
- nextLine(): Lee desde la entrada estándar (teclado) una secuencia de caracteres hasta encontrar un salto de línea. Devuelve un String.

Ejemplo:

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

```
//De lo que introduce el usuario, lee la 1° palabra.
String nombre = tec.next();
//Lee lo que introduce el usuario hasta que pulsa intro.
String nombreCompleto = tex.nextLine();
```

Cuando utilicemos nextLine() hay que tener en cuenta que es el único método next…() de Scanner que es capaz de leer una línea vacía. Esto puede causar algún problema cuando se ejecuta algún método next…( ) y a continuación un nextLine(), puesto que nextLine() se alimentará del salto de línea que deja en el buffer de teclado el método next…( )

Comparación de Strings

Los operadores relacionales (>, >=, <, <=, ==, !=) solo se deben utilizar para comparar variables y expresiones de tipo primitivo (byte, short, int, long, float, double, char y boolean).

Si utilizamos operadores relacionales para comparar dos objetos (por ejemplo, dos Strings) el resultado no será el esperado. En lugar de compararse los objetos en sí, se estarán comparando las direcciones de memoria en que se encuentran almacenados dichos objetos.

Por tanto, expresiones como las siguientes no hacen lo que se podría esperar de ellas:

String s1 = "hola", s2 = "adiós";

if (s1 == s2) …

if (s1 > s2) …

if (s1 <= s2) …

Para comparar Strings se utilizan dos métodos de la clase String:

- **equals**: para comprobar si dos cadenas son o no iguales.
- **compareTo**: para comprobar si una cadena es >, <, >=, <=, == o != a otra.

equals

Sean s1 y s2 dos variables o expresiones de tipo String

s1.equals(s2)

devuelve true si s1 y s2 contienen el mismo texto y false en caso contrario.

Ejemplo:

```
String n1="Miguel", n2="MIGUEL", n3="Ana", n4="Ana";
n1.equals(n2)  ☾  Se evalua a false
n3.equals(n4)  ☾  Se evalua a true
```

compareTo

Sean s1 y s2 dos variables o expresiones de tipo String

s1.compareTo(s2)

devuelve un entero que será:

- 0 si s1 y s2 son iguales.
- >0, si s1 es mayor que s2
- <0, si s1 es menor que s2

Ejemplo:

```
String n1="Miguel", n2="MIGUEL", n3="Ana", n4="Ana";
n2.compareTo(n3) -> Devuelve un número >0
n3.compareTo(n4) -> Devuelve 0
```

Métodos de la clase String.

La clase String es una clase de las que al comienzo del tema definíamos como "de datos". Los objetos de esta clase permiten representar secuencias de caracteres y, además, manipular dichas secuencias utilizando los métodos de la clase.

Muchos de los métodos de la clase String (no todos) son "***métodos de objeto***" (métodos no estáticos). La clase tiene también algunos "métodos de clase" (métodos estáticos).

Los ***métodos de objeto*** son métodos que se ejecutan sobre algún elemento de la clase, es decir, manipulan, extraen información o realizan algún cálculo sobre algún objeto en concreto. Para usar un método de objeto se utiliza la siguiente sintaxis:

objeto.método ()

o

objeto.método (parámetros)

Observa el siguiente ejemplo: Se definen dos variables de tipo String. Una se inicializa con el nombre de una persona y la otra con los apellidos. Al mostrar por pantalla, se ejecuta el método toUpperCase() sobre el String nombre. Éste método devuelve otro String, con el mismo contenido que nombre pero convertido a mayúsculas:

```
String nombre = "Miguel";
String apellido = "López Rubio";

System.out.println(nombre.toUpperCase()+ " " + apellido);
// Se muestra MIGUEL Lopez Rubio por pantalla
```

Se dice que toUpperCase es un método de objeto porque actua sobre un objeto en concreto (nombre en este caso). Los métodos de clase, por el contrario, no actúan sobre un objeto en concreto. La clase es un mero contenedor del método.

Accede a la documentación en línea de Java y estudia los siguientes métodos de la clase. Con cada método, explica qué hacen e ilustra su funcionamiento poniendo un ejemplo:

- toUpperCase
- toLowerCase()
- charAt
- indexOf
- substring
- trim

Ten en cuenta que de algunos métodos, como indexOf o substring, hay varias versiones en la documentación. Explica varias de las versiones.

5. Estructuras de control: La sentencia while

Como se indicó en el tema de introducción, la programación estructurada se base en el uso de 3 estructuras de programación para resolver cualquier problema:

- La secuencia.
- La estructura condicional o alternativa.
- La repetición o iteración.

En el tema anterior introdujimos el uso de las estructuras alternativas con la sentencia if.

Java tiene varias sentencias para implementar estructuras repetitivas, también llamadas ***bucles***. Una de ellas es la sentencia ***while***

Su sintaxis es la siguiente:

while (condición ***) {***

....

Instrucciones a ejecutar si se cumple la condición

...

}

Las llaves se pueden omitir si lo que se repite es una sola instrucción o sentencia.

Ejemplo: Mostrar los múltiplos de 5 que son menores que 1000

```
int multiplo = 5;
while (multiplo < 1000) {
    System.out.format("%d es múltiplo de 5%n",
multiplo);
    multiplo = multiplo + 5;
}
```

Repetir algo un número de veces

Para repetir una acción un número determinado de veces utilizaremos un ***contador***. Un contador, es una variable entera que utilizamos para saber en todo momento cuántas veces hemos realizado una acción.

Los contadores se inicializan a cero y se incrementan en uno cada vez que se ejecuta el bucle, es decir, cada vez que se realiza la acción que estamos contando.

Ejemplo: Pedir al usuario 10 números enteros.

```
int pedidos = 0; //contador
while (pedidos < 10) {
    System.out.println("Introduce número entero: ");
    int numero = tec.nextInt();
    …
    //Hacer algo con el número introducido
    …
    //Incrementar el contador
    pedidos = pedidos + 1;
}
```

Repetir algo mientras se cumpla una condición

En otras ocasiones tenemos que repetir alguna acción pero no sabemos a priori cuantas veces se tiene que repetir, sino que se quiere hacer mientras que se de cierta condición.

En estos casos, simplemente expresamos la condición de repetición en el bucle, y no habrá contador.

Ejemplo: Pedir al usuario y leer un número positivo

```
…
System.out.println("Introduce número positivo: ");
num = tec.nextInt();
while( num < 0 ) {
    System.out.println("Tiene que ser positivo")
    System.out.pringln(Vuelve a intentarlo");
    num = tec.nextInt();
}
…
//Hacer algo con el número introducido
…
```

6. Programación modular (Ampliación de la estructura de programa)

La programación modular consiste en dividir el programa en partes independientes o ***módulos*** que realizan una tarea determinada. La programación modular resulta útil para solucionar problemas de complejidad. Cuando un programa se divide en partes independientes, fáciles de comprender, su complejidad se puede reducir.

La ***programación modular*** se define como la organización de un programa en pequeñas unidades independientes y a su vez, relacionados entre sí, de forma que unos módulos pueden utilizar o invocar a otros para realizar su tarea.

Un módulo debe realizar una función o tarea determinada. Consta de un conjunto de instrucciones que se procesan de una sola vez y que se agrupan bajo un nombre por el que posteriormente serán llamados o invocados desde diferentes puntos del programa.

Cada módulo puede ser llamado o invocado por uno o varios módulos, quedando suspendida la ejecución del módulo que llama y entrando en ejecución el módulo llamado hasta que este termina, devolviendo a continuación el control del programa al módulo que lo llamó en el mismo punto donde se efectuó la llamada.

Ventajas de la programación modular

Las principales ventajas de la programación modular son las que se enumeran a continuación:

- Facilita la comprensión del problema y su resolución escalonada.
- Aumenta la claridad y legibilidad de los programas.
- Permite la resolución del problema por varios programadores a la vez.
- Reduce el tiempo de desarrollo aprovechando módulos desarrollados previamente.
- Facilita la depuración de los programas, ya que se puede depurar los módulos de forma más sencilla e independiente.
- Posibilita un mejor y más rápido mantenimiento de la aplicación, al poder realizar las modificaciones o implementaciones de una forma más sencilla tomando como base los módulos ya desarrollados.

- Posibilita que distintos módulos puedan implementarse con lenguajes de programación distintos, que pueden ser más adecuados para realizar determinadas tareas.

Programación modular en Java: métodos

En el lenguaje Java, los módulos de los que hablamos se materializan en ***métodos*****.**

Hasta ahora hemos utilizado ciertos ***métodos que se encuentran ya definidos*** en clases del propio lenguaje, como Math o String, o en otras clases diseñadas para tareas específicas, como Scanner. En concreto, hemos utilizado los métodos sqrt de la clase Math, los métodos nextInt() o nextDouble() de Scanner, etc.

Observa el siguiente fragmento de programa, que tiene cuatro instrucciones y en las cuatro se utilizan o invocan métodos:

```
System.out.println("Introduce un número: ")
int x = tec.nextInt();
double rx = Math.sqrt(x);
System.out.format("La raíz de %d es %f",x,rx);
```

(Insistimos en que, podemos encontrar métodos de clase (static) y métodos de objeto (no static). Dependiendo de si se trata de un tipo de método u otro, habrá que poner el nombre de la clase o el nombre de un objeto a la izquierda del punto para llamarlos.)

Si se observa el ejemplo, se pueden extraer algunas características comunes referentes al uso de métodos:

- Todos los métodos tienen un **identificador**: sqrt, println, format, nextInt. Para dar nombre a los métodos se aplican las mismas reglas (estudiadas en el Tema 1), que para dar nombres a las variables
- Después del identificador, y entre paréntesis, figuran los parámetros del método. Muchos de los problemas que resolvemos están parametrizados, es decir, realizan una tarea o un cálculo a partir de uno o varios datos que se le proporcionan. De poco serviría, por ejemplo, un método que solo fuera capaz de calcular la raíz cuadrada de 4. La potencia del método sqrt radica en poder calcular la raíz cuadrada de cualquier número real positivo. rqrt requiere un parámetro para su ejecución.
- En cualquier caso, los métodos también pueden no tener parámetros, como el método nextInt(), y también podemos encontrar métodos en los

que los parámetros sean opcionales. Aún cuando llamamos a un método que no recibe parámetros, tenemos que poner paréntesis.

- Algunos métodos devuelven un resultado, por ejemplo pow o sqrt devuelven un double. Otros, sin embargo, realizan alguna tarea pero no devuelven ningún resultado, como es el caso de println, que imprime por pantalla un texto.

Definición de métodos

La posibilidad de ***definir nuevos métodos*** es una herramienta de gran importancia en la programación ya que es lo va a hacer posible descomponer nuestros programas en piezas más manejables.

La definición de un método tiene dos partes:

- ***la cabecera:*** En ella se indican:
  - **el tipo de dato** que devuelve (el tipo de dato de su resultado). Si se trata de un procedimiento (no devuelve nada) se indica con la palabra reservada ***void***
  - **el nombre del método,** que ha de ser un identificador válido
  - **los parámetros** o argumentos que recibe, que se especifican entre paréntesis

  NOTA: Además, en la cabecera aparecen también una serie de modificadores como public, private, static, syncronized,etc., que no estudiaremos todavía.

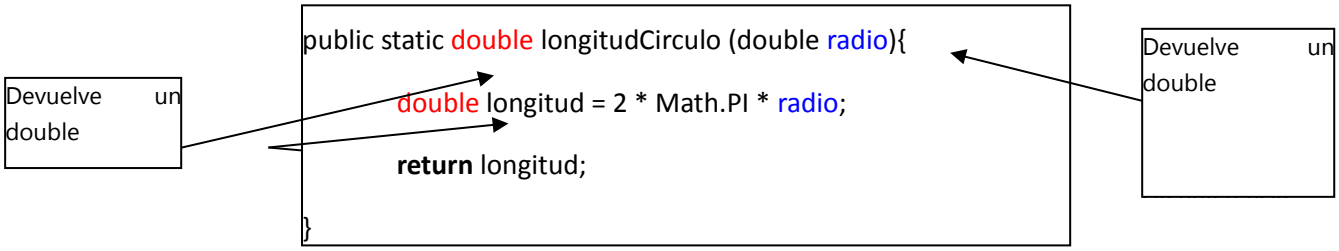
- ***el cuerpo:*** Contiene las instrucciones destinadas a realizar la tarea para la que se ha diseñado el método. El resultado se devuelve utilizando la instrucción **return** *expresión;* . La expresión devuelta tiene que ser del mismo tipo o de un tipo compatible con el que se ha declarado en la cabecera. Si el método no devuelve nada, se puede omitir la sentencia return o poner simplemente return; (sin expresión)

```
tipoResultado nombreMetodo (tipo param1, tipo param2,..) {
        // instrucciones
        ...
        return expresión;
}
```

Los parámetros que aparecen en la definición del método (param1, param2, etc) reciben

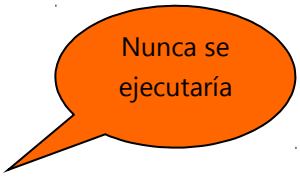
el nombre de ***parámetros formales***. Vemos que la declaración de cada parámetro lleva su tipo y su nombre y que van separados por comas, como si de declaraciones de variables se tratara. Podemos definir un método con tantos parámetros como queramos, aunque lo habitual es que no sean muchos. También es posible que el método no tenga ningún parámetro. Aunque no haya parámetros se ponen los paréntesis.

Ejemplo



El resultado del método es el que se indica con la sentencia **return** y, como vemos en el ejemplo, su tipo ha de coincidir con el tipo declarado en la cabecera.

Hay que tener en cuenta que la sentencia return, además de proporcionar el resultado, provoca la terminación de la ejecución del método. Por ejemplo, en esta otra versión de longitudCirculo, la instrucción System.out.println nunca se ejecutaría, ya que está precedida de una sentencia return que finaliza la ejecución del método.



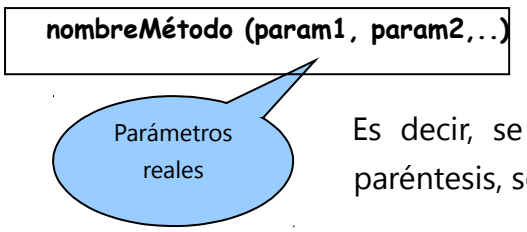

```
public static double longitudCirculo (double radio){
    double longitud = 2 * Math.PI * radio;
    return longitud;
    System.out.println("La longitud es": + longitud);
}
```

Aun así, en un método pueden aparecer varias sentencias return, aunque, claro está, solo una de ellas llegará a ejecutarse, como por ejemplo en el siguiente ejemplo:

```
//Devuelve el máximo de dos enteros
public static int maximo (int a, int b){
    if (a>b){
        return a;
    } else {
        return b;
    }
}
```

Llamada o invocación a un método.

Llamamos o invocamos a un método cada vez que queremos calcular o hacer aquello que el método es capaz de hacer. Por ejemplo, llamaremos a Math.sqrt cada vez que queramos calcular una raíz cuadrada, o a longitudCirculo cada vez que queramos calcular la longitud de un círculo o a imprimirCabecera cada vez que queremos imprimir la cabecera de una factura.



Para llamar o invocar a un método se utiliza la siguiente sintaxis:

Es decir, se escribe el nombre del método y, entre paréntesis, se pasan los parámetros que requiere.

Para que la llamada sea correcta se tienen que cumplir ciertas reglas:

- El número y el tipo de los parámetros reales que se pasan en la llamada tiene que coincidir con el número y el tipo de los parámetros formales que se han declarado en la definición del método (cabecera).
- La llamada se ha de realizar en el contexto adecuado:

- Así, por ejemplo, si es una función que devuelve un int la llamada a la función podría aparecer en la asignación a una variable de tipo int …

```
int x,y;
...
int mayor = maximo (x , y);
```

… o en una expresión aritmética

```
int a = (7 * maximo (x , y)  + 1)
```

… o, en definitiva, en cualquier parte en la que se espere una expresión cuyo resultado sea int.

- De la misma forma, si un método devuelve un boolean, su resultado podría asignarse a una variable boolean …

```
public static boolean fechaEsValida (int dia, int mes, int año) {
    …
    …
}
```

```
boolean fechaOk = fechaEsValida(d, m, a);
```

… o utilizarse como parte de una condición…

```
if ( fechaEsValida(d,m,a) ) { … }
```

- Si el método no devuelve nada, la llamada al método se realizará de forma aislada, es decir, sin ser parte de una expresión más compleja (asignación, condición, etc):

```
System.out.println("Hola");
```

Efecto de la llamada a un método.

Cuando se llama a un método se realizan los siguientes pasos:

- Se crean las variables y los parámetros formales del método al que se ha llamado. Cada método tiene su propio espacio de memoria, lo que permite que distintos métodos tengan variables o parámetros con el mismo nombre sin que se interfieran unas con otras.
- Cada parámetro formal toma el valor de su correspondiente parámetro real.
- La ejecución del método que ha realizado la llamada se detiene y comienza la ejecución de las instrucciones del método al que se ha llamado. El método finalizará cuando se ejecute su instrucción return o cuando se ejecute su última instrucción (si el método no devuelve nada).
- El espacio de memoria del método se libera, por lo que sus variables y parámetros dejan de existir.
- El método que ha llamado continúa en el punto en que se quedó detenido. Si el método llamado devolvía un resultado, el resultado que ha devuelto sustituye a todos los efectos a la llamada.

7. Anexos

Entrada y salida formateada

https://dzone.com/articles/java-string-format-examples

Tipos de datos elementales

En Java existen ***8 tipos de datos elementales*** (o básicos, o primitivos):

El tipo de una variable determina qué información puede contener y que operaciones se pueden realizar con ella.

Numéricos	Enteros	**byte**
		short
		int
		long
	Reales	**float**

		double
Carácter		**char**
Lógicos		**boolean**