

TEMA4. INTRODUCCIÓN A JAVA

Programación
CFGS DAW

Profesores:

Carlos Cacho

Raquel Torres

carlos.cacho@ceedcv.es

raquel.torres@ceedcv.es

Licencia



Reconocimiento - NoComercial - CompartirIgual (by-nc-sa): No se permite un uso comercial de la obra original ni de las posibles obras derivadas, la distribución de las cuales se debe hacer con una licencia igual a la que regula la obra original.

Nomenclatura

A lo largo de este tema se utilizarán distintos símbolos para distinguir elementos importantes dentro del contenido. Estos símbolos son:



Importante



Atención



Interesante

Revisiones

-

ÍNDICE DE CONTENIDO

1.Introducción.....	4
2.Primer ejemplo.....	5
3.Elementos básicos.....	6
3.1 Comentarios.....	6
3.2 Identificadores.....	7
4.Tipos de datos.....	8
4.1 Tipos de datos simples.....	9
5.Declaración de variables.....	10
5.1 Ámbito de una variable.....	11
5.2 Variables locales.....	12
6.Salida y entrada estándar.....	13
6.1 Salida estándar.....	13
6.2 Entrada estándar.....	14
6.2.1 Entrada de datos numéricos.....	16
6.3 La clase Scanner.....	18
7.Operadores.....	20
7.1 Aritméticos.....	21
7.2 Relacionales.....	22
7.3 Lógicos.....	23
7.4 De asignación.....	25
7.5 La clase Math.....	25
8.Literales.....	26
8.1 Literales lógicos.....	26
8.2 Literales enteros.....	26
8.3 Literales reales.....	27
8.4 Literales carácter.....	27
8.5 Literales cadenas.....	28
9.Estructuras alternativas.....	29
9.1 Alternativa simple y doble.....	30
9.2 Alternativa múltiple.....	33
10.ejemplos.....	35
10.1 Ejemplo1.....	35
10.2 Ejemplo 2.....	36
11.Agradecimientos.....	38

UD04. INTRODUCCIÓN A JAVA

1. INTRODUCCIÓN

Java es un lenguaje de programación de propósito general, concurrente, orientado a objetos que fue diseñado específicamente para tener tan pocas dependencias de implementación como fuera posible. Su objetivo es permitir que los desarrolladores de aplicaciones escriban el programa una vez y lo ejecuten en cualquier dispositivo (conocido en inglés como *WORA*, o "*write once, run anywhere*"), lo que quiere decir que el código que es ejecutado en una plataforma no tiene que ser recompilado para correr en otra.

Las características de Java son:

- Simple: Es un lenguaje sencillo de aprender.
- Orientado a Objetos: Posiblemente sea el lenguaje más orientado a objetos de todos los existentes; en Java todo, a excepción de los tipos fundamentales de variables (int, char, long...) es un objeto.
- Distribuido: Java está muy orientado al trabajo en red, soportando protocolos como TCP/IP, UDP, HTTP y FTP. Por otro lado el uso de estos protocolos es bastante sencillo comparandolo con otros lenguajes que los soportan.
- Robusto: El compilador Java detecta muchos errores que otros compiladores solo detectarían en tiempo de ejecución o incluso nunca.
- Seguro: Sobre todo un tipo de desarrollo: los Applet. Estos son programas diseñados para ser ejecutados en una página web.
- Portable: En Java no hay aspectos dependientes de la implementación, todas las implementaciones de Java siguen los mismos estándares en cuanto a tamaño y almacenamiento de los datos.
- Arquitectura Neutral: El código generado por el compilador Java es independiente de la arquitectura: podría ejecutarse en un entorno UNIX, Mac o Windows.
- Rendimiento medio: Actualmente la velocidad de procesamiento del código Java es semejante a las de otros lenguajes orientados a objetos.
- Multithread: Soporta de modo nativo los threads (hilos de ejecución), sin necesidad del uso de librerías específicas.

2. PRIMER EJEMPLO

La aplicación más pequeña posible es la que simplemente imprime un mensaje en la pantalla. Tradicionalmente, el mensaje suele ser "Hola Mundo!". Esto es justamente lo que hace el siguiente fragmento de código:

```
12  public class HolaMundo {  
13  
14      public static void main(String[] args) {  
15          // TODO code application logic here  
16          System.out.println("Hola Mundo!");  
17      }  
18  
19  }
```

Hay que ver en detalle la aplicación anterior, línea a línea. Esas líneas de código contienen los componentes mínimos para imprimir *Hola Mundo!* en la pantalla. Es un ejemplo muy simple, que no instancia objetos de ninguna otra clase; sin embargo, accede a otra clase incluida en el JDK.

public class HolaMundo

Esta línea **declara la clase `HolaMundo`**. El nombre de la clase especificado en el fichero fuente **se utiliza para crear un fichero `nombredeclase.class`** en el directorio en el que se compila la aplicación. En este caso, el compilador creará un fichero llamado `HolaMundo.class`.

public static void main(String args[])

Esta línea **especifica un método** que el intérprete Java busca para ejecutar en primer lugar. Igual que en otros lenguajes, Java utiliza una **palabra clave `main`** para **especificar la primera función a ejecutar**. En este ejemplo tan simple no se pasan argumentos.

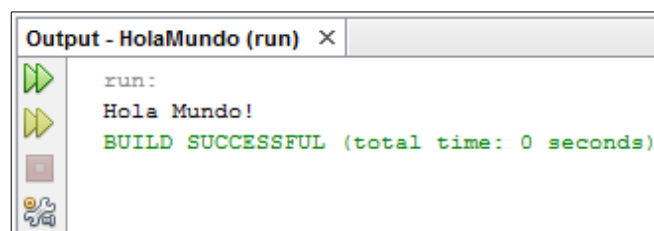
- **public** significa que el método `main()` puede ser llamado por cualquiera, incluyendo el intérprete Java.
- **static** es una palabra clave que le dice al compilador que `main` se refiere a la propia clase `HolaMundo` y no a ninguna instancia de la clase. De esta forma, si alguien intenta hacer otra instancia de la clase, el método `main()` no se instanciaría.
- **void** indica que `main()` no devuelve nada. Esto es importante ya que Java realiza una estricta comprobación de tipos, incluyendo los tipos que se ha declarado que devuelven los métodos.
- **args[]** es la declaración de un array de Strings. Estos son los argumentos escritos tras el nombre de la clase en la línea de comandos: `java HolaMundo arg1 arg2 ...`

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

Esta es la funcionalidad de la aplicación. Esta línea muestra el uso de un nombre de clase y método. Se usa el **método `println()`** de la **clase `out`** que está en el **paquete `System`**.

El método `println()` toma una cadena como argumento y la escribe en el stream de salida estándar; en este caso, la ventana donde se lanza la aplicación. La clase `PrintStream` tiene un método instanciable llamado `println()`, que lo que hace es presentar en la salida estándar del Sistema el argumento que se le pase. En este caso, se utiliza la variable o instancia de `out` para acceder al método.

El resultado sería el siguiente:



⚡ Todas las instrucciones (creación de variables, llamadas a métodos, asignaciones) se deben **finalizar** con un **punto y coma**.

3. ELEMENTOS BÁSICOS

3.1 Comentarios

En Java hay tres tipos de comentarios:

```
// comentarios para una sola línea
```

```
/*  
comentarios de una o más líneas  
*/
```

```
/** comentario de documentación, de una o más líneas  
*/
```

Los dos primeros tipos de comentarios son los que todo programador conoce y se utilizan del mismo modo.

Los **comentarios de documentación**, colocados inmediatamente antes de una declaración (de variable o función), indican que ese comentario ha de ser colocado en la documentación que se genera automáticamente cuando se utiliza la herramienta de Java, **javadoc**, no disponible en otros lenguajes de programación. Este tipo de comentario lo veremos más adelante.

3.2 Identificadores

Los **identificadores nombran variables, funciones, clases y objetos**; cualquier cosa que el programador necesite identificar o usar.

Reglas para la creación de identificadores:

- **Java hace distinción entre mayúsculas y minúsculas**, por lo tanto, nombres o identificadores como var1, Var1 y VAR1 son distintos.
- Pueden estar formados por cualquiera de los caracteres del código Unicode, por lo tanto, se pueden declarar variables con el nombre: añoDeCreación, raím, etc., aunque eso sí, el **primer carácter no** puede ser un **dígito numérico** y **no** pueden utilizarse **espacios en blanco** ni símbolos coincidentes con operadores.
- La **longitud** máxima de los identificadores es prácticamente **ilimitada**.
- **No** puede ser una **palabra reservada del lenguaje** ni los valores lógicos true o false.
- **No** pueden ser **iguales a otro identificador** declarado en el mismo ámbito.
- **IMPORTANTE:** Por convenio:
 - Los **nombres** de las **variables** y los **métodos** deberían **empezar** por una **letra minúscula** y los de las **clases** **por mayúscula**.
 - Si el identificador está formado por **varias palabras**, la **primera** se escribe en **minúsculas** (excepto para las clases) y el **resto** de palabras se hace **empezar por mayúscula** (por ejemplo: añoDeCreación).
 - Estas **reglas** no son obligatorias, pero son **convenientes** ya que ayudan al proceso de codificación de un programa, así como a su legibilidad. Es más sencillo distinguir entre clases y métodos o variables.

Serían identificadores válidos:

```
identificador
nombre_Usuario
Nombre_Usuario
_variable_del_sistema
$transaccion
```

y su uso sería, por ejemplo:

```
int contador_principal;
char _lista_de_ficheros;
float $cantidad;
```

4. TIPOS DE DATOS

En Java existen dos tipos principales de datos:

1. Tipos de datos simples.
2. Referencias a objetos.



Los tipos de **datos simples** son aquellos que pueden utilizarse directamente en un programa.

Estos tipos son:

- byte
- short
- int
- long
- float
- double
- char
- boolean

El **segundo tipo está formado por todos los demás**. Se les llama referencias porque en realidad lo que se almacena en los mismos son punteros a zonas de memoria donde se encuentran almacenadas las estructuras de datos que los soportan. Dentro de este grupo se encuentran las clases (objetos) y también se incluyen las interfaces, los vectores y los Strings. Estos tipos de datos los veremos en la unidad 8.

4.1 Tipos de datos simples

Los tipos de datos simples soportados por Java son los siguientes:

Tipo	Descripción	Longitud	Rango
byte	byte	1 byte	-128 ... 127
short	entero corto	2 bytes	-32768 ... 32767
int	entero	4 bytes	-2147483648 ... 2147483647
long	entero largo	8 bytes	-9223372036854775808 ... 9223372036854775807
float	real en coma flotante de simple precisión	32 bits	$\pm 3,4 \cdot 10^{-38}$... $\pm 3,4 \cdot 10^{38}$
double	real en coma flotante de doble precisión	64 bits	$\pm 1,7 \cdot 10^{-308}$... $\pm 1,7 \cdot 10^{308}$
char	carácter	2 bytes	
boolean	lógico	1 bit	true/false

⚡ **Java no realiza una comprobación de los rangos.**

Por ejemplo: si a una variable de tipo short con el valor 32.767 se le suma 1, el resultado será -32.768 (es decir, no desborda, sino que su comportamiento es cíclico, vuelve al valor inicial del rango) y no se producirá ningún error de ejecución.


A diferencia de otros lenguajes de programación:

⚡ Los **Strings** en Java no son un tipo simple de datos sino un **objeto**.

Los valores de tipo String van entre comillas dobles ("Hola"), mientras que los de tipo char van entre comillas simples ('K').

5. DECLARACIÓN DE VARIABLES

La forma básica de una declaración de variable, por ejemplo, sería:


 tipo identificador [= valor][,identificador [= valor] ...];

La declaración de una variable siempre contiene:

1. el nombre (identificador de la variable)
2. el tipo de dato al que pertenece.

El **ámbito** de la variable depende de la localización en el programa donde es declarada.

Ejemplo: int x;

 Las variables pueden ser **inicializadas** en el momento de su declaración, siempre que el valor que se les asigne coincida con el tipo de dato de la variable.

Ejemplo: int x = 0;

Esto es equivalente a: int x;
x = 0;

Palabras clave

Las siguientes son las palabras clave que están definidas en Java y que no se pueden utilizar como identificadores:


abstract	continue	for	new	switch
boolean	default	goto	null	synchronized
break	do	if	package	this
byte	double	implements	private	threadsafe
byvalue	else	import	protected	throw
case	extends	instanceof	public	transient
catch	false	int	return	true
char	final	interface	short	try
class	finally	long	static	void
const	float	native	super	while

Palabras reservadas

Además, el lenguaje se reserva unas cuantas palabras más, pero que hasta ahora no tienen un cometido específico. Son:


cast	uture	generic	inner
operator	outer	rest	var

5.1 Ámbito de una variable

 El **ámbito** de una variable es la porción de programa donde dicha variable es visible para el código del programa.

El ámbito de una variable depende del lugar del programa donde es declarada, pudiendo pertenecer a cuatro categorías distintas.

- Variable local.
- Atributo.
- Parámetro de un método.
- Parámetro de un tratador de excepciones.

 Si una variable no ha sido inicializada, tiene un valor asignado por defecto.

Este valor es:

- Para las variables de tipo referencial (**objetos**), el valor **null**.
- Para las variables de tipo **numérico**, el valor por defecto es cero (**0**),.
- Las variables de tipo **char**, el valor '**\u0000**'.
- Las variables de tipo **boolean**, el valor **false**.

Es una buena práctica inicializarlas siempre.

5.2 Variables locales

Una **variable local** se declara dentro del cuerpo de un método de una clase y es **visible únicamente dentro** de dicho **método**.

Se puede declarar en cualquier lugar del cuerpo, incluso después de instrucciones ejecutables, aunque es una **buena costumbre declararlas justo al principio**.

También pueden declararse variables dentro de un bloque parentizado por llaves {...}. En ese caso, sólo serán “visibles” dentro de dicho bloque.

Por ejemplo (No es necesario entender lo que hace el programa) :

```
14 public static void main(String[] args) {  
15  
16     int i;  
17  
18     for (i=0;i<10;i++)  
19         System.out.println(i);  
20 }
```

En este ejemplo existe una variable local: **int i**; únicamente es visible dentro del método main.

Cuando se pretende **declarar múltiples variables** del mismo tipo pueden declararse, en forma de lista, **separadas por comas**.

Ejemplo:

```
int x,y,z;
```

Declara tres variables x, y, z de tipo entero.

Podrían haberse inicializado en su declaración de la forma:

```
int x=0,y=0,z=3;
```

Las variables locales pueden ser antecedidas por la palabra reservada *final*. En

ese caso, sólo permiten que se les asigne un valor una única vez.

Ejemplo:

```
final int x=0;
```

No permitirá que a x se le asigne ningún otro valor. Siempre contendrá 0.

No es necesario que el valor se le asigne en el momento de la declaración, podría haberse inicializado en cualquier otro lugar, pero una sola vez.

//Ejemplo:

```
final int x;
```

```
...
```

```
x=y+2;
```

Después de la asignación `x=y+2`, no se permitirá asignar ningún otro valor a x.

⚡ Por lo tanto una variable precedida de la palabra **final** se convierte en una **constante**. O lo que es lo mismo, para definir una constante en Java deberemos preceder su declaración de la palabra reservada **final**.

6. SALIDA Y ENTRADA ESTÁNDAR

6.1 Salida estándar

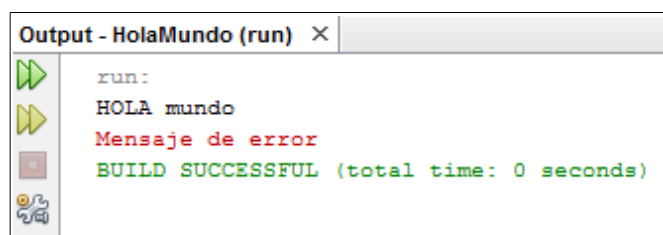
Ya hemos visto el uso de `System.out` para enviar información a la pantalla con los métodos `print` y `println` (el segundo introduce un salto de línea después de imprimir la cadena).

La utilización de `System.err` sería totalmente análoga para enviar los mensajes producidos por errores en la ejecución (es el canal que usa también el compilador para notificar los errores encontrados).

Por ejemplo, para presentar el mensaje de saludo habitual por pantalla, y después un mensaje de error, tendríamos la siguiente clase (aunque en realidad toda la información va a la consola de comandos donde estamos ejecutando el programa):

```
14 public static void main(String[] args) {  
15  
16     System.out.print("HOLA ");  
17     System.out.println("mundo");  
18     System.err.println("Mensaje de error");  
19 }
```

Y la salida sería la siguiente:



```
Output - HolaMundo (run) ×  
run:  
HOLA mundo  
Mensaje de error  
BUILD SUCCESSFUL (total time: 0 seconds)
```

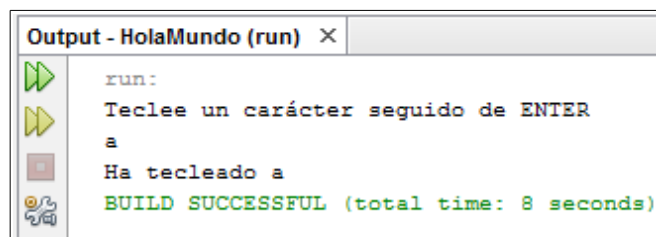
6.2 Entrada estándar

En cuanto a la entrada estándar en Java, es bastante más compleja. El procedimiento más simple y de más bajo nivel para leer de la entrada estándar es invocar directamente al método `read()` del flujo `System.in`. Este método nos devolverá un byte del flujo de entrada en forma de entero, o un -1 si no hay más entrada disponible.

El siguiente ejemplo lo ilustra:

```
12 import java.io.*;  
13  
14 public class Leer {  
15  
16     public static void main(String[] args) throws IOException{  
17  
18         char c;  
19  
20         System.out.println("Teclee un carácter seguido de ENTER");  
21  
22         c = (char) System.in.read();  
23  
24         System.out.println("Ha tecleado "+c);  
25     }  
26 }
```


El resultado sería algo como lo siguiente:



```
Output - HolaMundo (run) X
run:
Teclee un carácter seguido de ENTER
a
Ha tecleado a
BUILD SUCCESSFUL (total time: 8 seconds)
```

Como el método *read* almacena el resultado como un entero, es necesario convertirlo después a un carácter con una operación de *cast* (molde) .

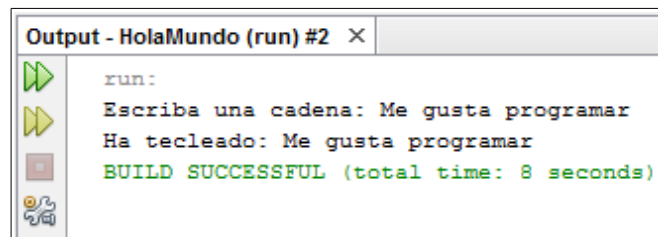
El uso directo del método *read* para leer bytes es muy poco práctico y, por lo tanto poco usado. No hay más que pensar en el esfuerzo que supondría leer datos de esta manera y hacer las conversiones apropiadas para cada caso. Lo que se hace es utilizar varias clases *java.io*. El método que utilizaremos es *readLine()* que pertenece a la clase denominada *BufferedReader*.

 *readLine()* permite leer una secuencia de caracteres de texto, en formato Unicode, para devolver una cadena de tipo *String*.

Sin embargo, para crear el objeto de esta clase necesitamos utilizar otra clase intermedia, *InputStreamReader*, que nos permite cambiar un objeto de clase *InputStream* (que lee bytes) en un objeto de tipo *Reader*, necesario para crear un objeto de tipo *BufferedReader*.

```
12 import java.io.*;
13
14 public class Leer {
15
16     public static void main(String[] args) throws IOException{
17
18         BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in));
19         String cadena;
20
21         System.out.print("Escriba una cadena: ");
22
23         cadena = stdin.readLine();
24
25         System.out.println("Ha tecleado: " + cadena);
26     }
27 }
```

El resultado sería algo como lo siguiente:



Como podemos ver, una vez constituido el objeto *stdin*, simplemente le pedimos cadenas de texto (objetos *String*), ya que el proceso de convertir bytes en caracteres y agruparlos en cadenas es totalmente transparente para el programador.

Por ahora no hace falta entender cómo funciona, simplemente saber que lo podemos utilizar para leer cadenas de texto.

⚡ Resumiendo, con el método `readLine()`, lo único que podemos leer desde el teclado son cadenas de caracteres (tipo `String`), estas cadenas son objetos.

Para poder hacer esto debemos:

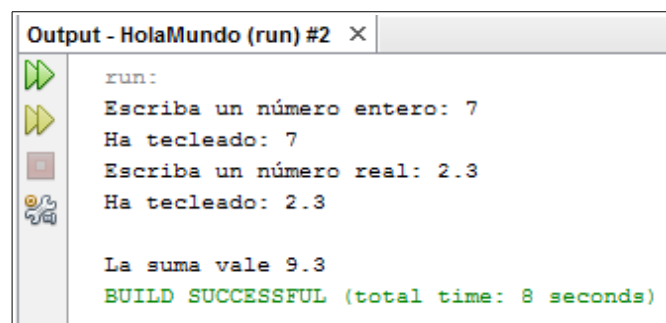
- Importar la librería que me permite utilizar estos objetos: **`import java.io.*;`**
- Preparar mi clase principal (programa) para posibles errores, esto es obligatorio. (Esto lo veremos más adelante): **`throws IOException;`**
- Declarar un objeto que se asocia al teclado del ordenador, este objeto se llamará *stdin* (podéis darle el nombre que queráis): **`BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in));`**
- Declarar objeto para poder guardar lo que leamos desde el teclado: **`String cadena;`**
- Realizar la lectura desde el teclado y guardar el valor introducido en objeto de tipo `String`. RECUERDA que siempre será una cadena alfanumérica: **`cadena = stdin.readLine();`**

6.2.1 Entrada de datos numéricos

Finalmente, una vez que tenemos los datos como cadenas a través de `readLine()`, si necesitamos tener tipos de datos básicos podemos recurrir a las *clases envoltorio*, con las funcionalidades *parser* para extraer datos, como se indica a continuación:


```
16 public static void main(String[] args) throws IOException{
17
18     BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in));
19
20     int entero;
21     float real;
22
23     System.out.print("Escriba un número entero: ");
24
25     // Lectura de un número entero, int, short, byte o long
26     // Short.parseInt() Byte.parseByte() Long.parseLong()
27     entero = Integer.parseInt(stdin.readLine());
28
29     System.out.println("Ha tecleado: " + entero);
30
31     System.out.print("Escriba un número real: ");
32
33     // Lectura de un número real: float o double
34     // Float.parseFloat() Double.parseDouble()
35     real = Float.parseFloat(stdin.readLine());
36
37     System.out.println("Ha tecleado: " + real);
38     System.out.println();
39     System.out.println("La suma vale " + (entero+real));
```

La salida sería algo como lo siguiente:



```
run:
Escriba un número entero: 7
Ha tecleado: 7
Escriba un número real: 2.3
Ha tecleado: 2.3

La suma vale 9.3
BUILD SUCCESSFUL (total time: 8 seconds)
```


En realidad lo que hacemos es convertir la cadena de caracteres que leemos desde el teclado a un numérico específico (byte, short, int, long, float o double).

En Java, a excepción de los tipos básicos descritos antes, todos los demás elementos son clases y objetos, lo que hace que en algunas circunstancias tengamos que convertir estos tipos básicos en objetos.

Para facilitar esta conversión utilizamos los envoltorios, que son clases especiales que recubren el tipo básico con una clase, y a partir de este momento el tipo básico envuelto se convierte en un objeto.

Existen **nueve envoltorios** para tipos básicos de Java. Cada uno de ellos envuelve un tipo básico (añadiendo el tipo vacío, *void*) y **nos permiten trabajar con objetos en lugar de con tipos básicos**.

En concreto, son los siguientes:

 Envoltorios para tipos básicos de Java: **Integer, Long, Float, Double, Short, Byte, Character, Boolean y Void.**

La función principal de estas clases es la de crear objetos cuya información sea la de un tipo básico asociado.

Además, existen algunos métodos que nos permiten convertir cadenas de caracteres en tipos básicos. Así, podemos convertir la cadena "123" en el número entero 123 utilizando el método `parseInt()` de la clase `Integer`.

```
int num = Integer.parseInt("123");
```

6.3 La clase `Scanner`

Desde Java 1.5 disponemos de la clase `Scanner`.

La utilización de la **clase `Scanner`** es muy sencilla. Lo primero que tenemos que hacer es declarar un **objeto `Scanner`** instanciándolo contra la consola, es decir, contra el **objeto `System.in`**


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

Ahora, para leer lo que el usuario está introduciendo por la consola deberemos de utilizar el **método `.next`**. Este nos devolverá los caracteres que encuentre en la consola hasta encontrarse un retorno de carro y salto de línea. El valor se lo asignaremos a una **variable `String`**.

```
String sTexto = reader.next();
```

Una vez que se declara e inicializa la variable se pueden invocar **métodos de la clase `Scanner`** tales como:

- `nextByte`: obtiene un número entero tipo byte.
- `nextShort`: obtiene un número entero tipo short.
- `nextInt`: obtiene un número entero tipo int.
- `nextLong`: obtiene un número entero tipo long.
- `nextFloat`: obtiene un número real float.
- `nextDouble`: obtiene un número real double.
- `nextLine`: obtiene una cadena de caracteres.

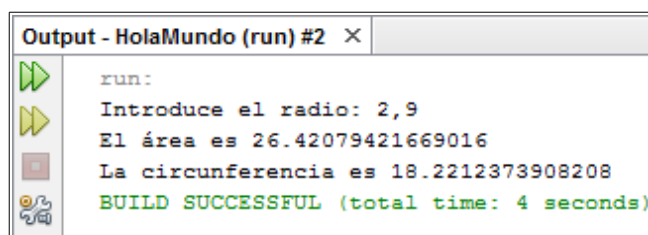
 No existen métodos de la clase `Scanner` para obtener directamente booleanos ni para obtener un solo carácter.

- `nextLine().charAt(0)` puede resolver el segundo problema.

Un ejemplo de lectura de un double sería el siguiente:

```
12  import java.util.Scanner;
13
14  public class EjemploScanner {
15
16      public static void main(String[] args){
17
18          double radio, area, circunferencia;
19
20          Scanner entrada = new Scanner(System.in);
21
22          System.out.print("Introduce el radio: ");
23
24          radio = entrada.nextDouble();
25
26          // Se hace uso de la librería Math para usar PI y la potencia(pow)
27          area = Math.PI * Math.pow(radio, 2);
28
29          circunferencia = 2 * Math.PI * radio;
30
31          System.out.println("El área es " + area);
32
33          System.out.println("La circunferencia es " + circunferencia);
34      }
35  }
```

Y su salida:

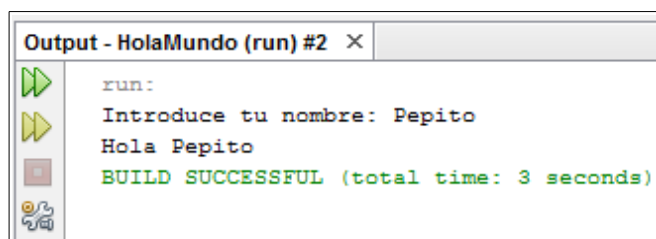


```
Output - HolaMundo (run) #2 ×
run:
Introduce el radio: 2,9
El área es 26.42079421669016
La circunferencia es 18.2212373908208
BUILD SUCCESSFUL (total time: 4 seconds)
```

Otro ejemplo, en este caso vamos a leer una cadena de texto:

```
12 import java.util.Scanner;
13
14 public class EjemploScanner {
15
16     public static void main(String[] args){
17
18         String nombre;
19
20         Scanner entrada = new Scanner(System.in);
21
22         System.out.print("Introduce tu nombre: ");
23
24         nombre = entrada.nextLine();
25
26         System.out.println("Hola " + nombre);
27
28     }
29 }
```

Y su salida:



```
Output - HolaMundo (run) #2 x
run:
Introduce tu nombre: Pepito
Hola Pepito
BUILD SUCCESSFUL (total time: 3 seconds)
```

7. OPERADORES

Los operadores son partes indispensables en la construcción de expresiones. Existen muchas definiciones técnicas para el término expresión.

Una **expresión** es una combinación de operandos ligados mediante operadores.

Los operandos pueden ser variables, constantes, funciones, literales, etc. y los operadores pueden ser:

- Aritméticos.
- Relacionales.
- Lógicos.
- Bits (prácticamente no los utilizaremos en este curso).
- Operador asignación.

Precedencia de operadores en Java:

- Operadores postfijos: `[] .` (*paréntesis*)
- Operadores unarios: `++expr, --expr, -expr, ~ !`
- Creación o conversión de tipo: `new (tipo)expr`
- Multiplicación y división: `*, /, %`
- Suma y resta: `+, -`
- Desplazamiento de bits: `<<, >>, >>>`
- Relacionales: `< >, <=, >=`
- Igualdad y desigualdad: `==, !=`
- AND a nivel de bits: `&`
- AND lógico: `&&`
- XOR a nivel de bits: `^`
- OR a nivel de bits: `|`
- OR lógico: `||`
- Condicional al estilo C: `? :`
- Asignación: `=, +=, -=, *=, /=, %=, ^=, &=, |=, >>=, <<=`

7.1 Aritméticos

Operador	Formato	Descripción
+	op1 +op2	Suma aritmética de dos operandos.
-	op1 - op2 -op1	Resta aritmética de dos operandos. Cambio de signo.
*	op1 * op2	Multiplicación de dos operandos
/	op1 / op2	División entera de dos operandos
%	op1 %op2	Resto de la división entera (o módulo)
++	++op1 op1++	Incremento unitario
--	--op1 op1--	Decremento unitario

El

operador `-` puede utilizarse en su versión unaria (`- op1`) y la operación que realiza es la de invertir el signo del operando.

Los operadores unarios `++` y `--` realizan un incremento y un decremento respectivamente. Estos operadores admiten notación prefija y postfija.

- `++op1`: En primer lugar realiza un incremento (en una unidad) de `op1` y después ejecuta la instrucción en la cual está inmerso.
- `op1++`: En primer lugar ejecuta la instrucción en la cual está inmerso y después realiza un incremento (en una unidad) de `op1`.
- `--op1`: En primer lugar realiza un decremento (en una unidad) de `op1` y después ejecuta la instrucción en la cual está inmerso.

- `op1--`: En primer lugar ejecuta la instrucción en la cual está inmerso y después realiza un decremento (en una unidad) de `op1`.

La diferencia entre la notación prefija y la postfija no tiene importancia en expresiones en las que únicamente existe dicha operación:

`++contador`; es equivalente a: `contador++`;

`--contador`; es equivalente a: `contador--`;

⚡ Es importante no emplear estos operadores en expresiones que contengan más de una referencia a la variable incrementada, puesto que esta práctica puede generar resultados erróneos fácilmente.

La diferencia es apreciable en instrucciones en las cuáles están incluidas otras operaciones. Por ejemplo:

```
cont = 1;
do {
...}
while ( cont++ < 3 );
```

```
cont = 1;
do {
...}
while ( ++cont < 3 );
```

En el primer caso, el bucle se ejecutará 3 veces, mientras que en el segundo se ejecutará 2 veces!!

7.2 Relacionales

Operador	Formato	Descripción
>	<code>op1 > op2</code>	Devuelve true (cierto) si <code>op1</code> es mayor que <code>op2</code>
<	<code>op1 < op2</code>	Devuelve true (cierto) si <code>op1</code> es menor que <code>op2</code>
>=	<code>op1 >= op2</code>	Devuelve true (cierto) si <code>op1</code> es mayor o igual que <code>op2</code>
<=	<code>op1 <= op2</code>	Devuelve true (cierto) si <code>op1</code> es menor o igual que <code>op2</code>
==	<code>op1 == op2</code>	Devuelve true (cierto) si <code>op1</code> es igual a <code>op2</code>
!=	<code>op1 != op2</code>	Devuelve true (cierto) si <code>op1</code> es distinto de <code>op2</code>

Los operadores relacionales actúan sobre valores enteros, reales y caracteres (char); y devuelven un valor del tipo boolean (true o false).

Ejemplo:

```

15 public static void main(String[] args){
16
17     double op1,op2;
18     char op3,op4;
19
20     op1=1.34;
21     op2=1.35;
22     op3='a';
23     op4='b';
24
25     System.out.println("op1=" + op1 + " op2=" + op2);
26     System.out.println("op1>op2 = " + (op1 > op2));
27     System.out.println("op1<op2 = " + (op1 < op2));
28     System.out.println("op1==op2 = " + (op1 == op2));
29     System.out.println("op1!=op2 = " + (op1 != op2));
30     System.out.println("'a'>'b' = " + (op3 > op4));
31
32 }
```

Salida:

```

run:
op1=1.34 op2=1.35
op1>op2 = false
op1<op2 = true
op1==op2 = false
op1!=op2 = true
'a'>'b' = false
BUILD SUCCESSFUL (total time: 0 seconds)
```

⚡ Los operadores == y != actúan también sobre valores lógicos (boolean).

7.3 Lógicos

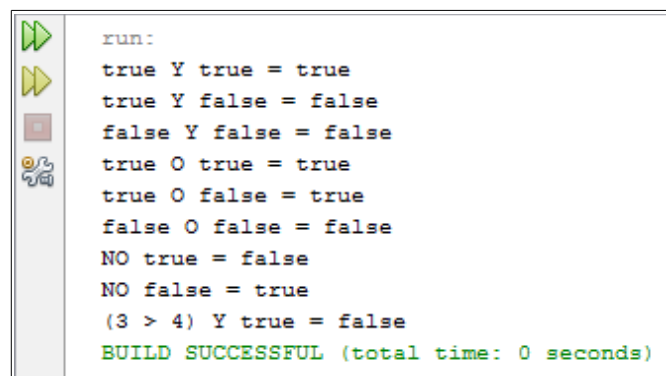
Operador	Formato	Descripción
&&	op1 && op2	Y lógico. Devuelve true (cierto) si son ciertos op1 y op2
	op1 op2	O lógico. Devuelve true (cierto) si son ciertos op1 o op2
!	! op1	Negación lógica. Devuelve true (cierto) si es false op1.

Estos operadores actúan sobre operadores o expresiones lógicas, es decir, aquellos que se evalúan a cierto o falso (true / false).

Ejemplo:

```
15 public static void main(String[] args){
16
17     boolean a, b, c, d;
18
19     a=true;
20     b=true;
21     c=false;
22     d=false;
23
24     System.out.println("true Y true = " + (a && b) );
25     System.out.println("true Y false = " + (a && c) );
26     System.out.println("false Y false = " + (c && d) );
27     System.out.println("true O true = " + (a || b) );
28     System.out.println("true O false = " + (a || c) );
29     System.out.println("false O false = " + (c || d) );
30     System.out.println("NO true = " + !a);
31     System.out.println("NO false = " + !c);
32     System.out.println("(3 > 4) Y true = " + ((3 >4) && a) );
33
34 }
35 }
```

Salida:



```

run:
true Y true = true
true Y false = false
false Y false = false
true O true = true
true O false = true
false O false = false
NO true = false
NO false = true
(3 > 4) Y true = false
BUILD SUCCESSFUL (total time: 0 seconds)

```

7.4 De asignación

El operador de asignación es el símbolo igual (=).

op1 = Expresión

Asigna el resultado de evaluar la expresión de la derecha a op1.

Además del operador de asignación existen unas abreviaturas cuando el operando que aparece a la izquierda del símbolo de asignación también aparece a la derecha del mismo:

Operador	Formato	Equivalencia
<code>+=</code>	<code>op1 +=op2</code>	<code>op1 =op1 +op2</code>
<code>-=</code>	<code>op1 -=op2</code>	<code>op1 =op1 - op2</code>
<code>*=</code>	<code>op1 *=op2</code>	<code>op1 =op1 * op2</code>
<code>/=</code>	<code>op1 /=op2</code>	<code>op1 =op1 / op2</code>
<code>%=</code>	<code>op1 %=op2</code>	<code>op1 =op1 %op2</code>
<code>&=</code>	<code>op1 &=op2</code>	<code>op1 =op1 & op2</code>
<code> =</code>	<code>op1 =op2</code>	<code>op1 =op1 op2</code>
<code>^=</code>	<code>op1 ^=op2</code>	<code>op1 =op1 ^op2</code>
<code>>>=</code>	<code>op1 >>=op2</code>	<code>op1 =op1 >>op2</code>
<code><<=</code>	<code>op1 <<=op2</code>	<code>op1 =op1 <<op2</code>
<code>>>>=</code>	<code>op1 >>>=op2</code>	<code>op1 =op1 >>>op2</code>

7.5 La clase Math

Se echan de menos operadores matemáticos más potentes en Java. Por ello se ha incluido una clase especial llamada **Math** dentro del paquete `java.lang`.

Esta clase posee métodos muy interesantes para realizar cálculos matemáticos complejos. Por ejemplo:

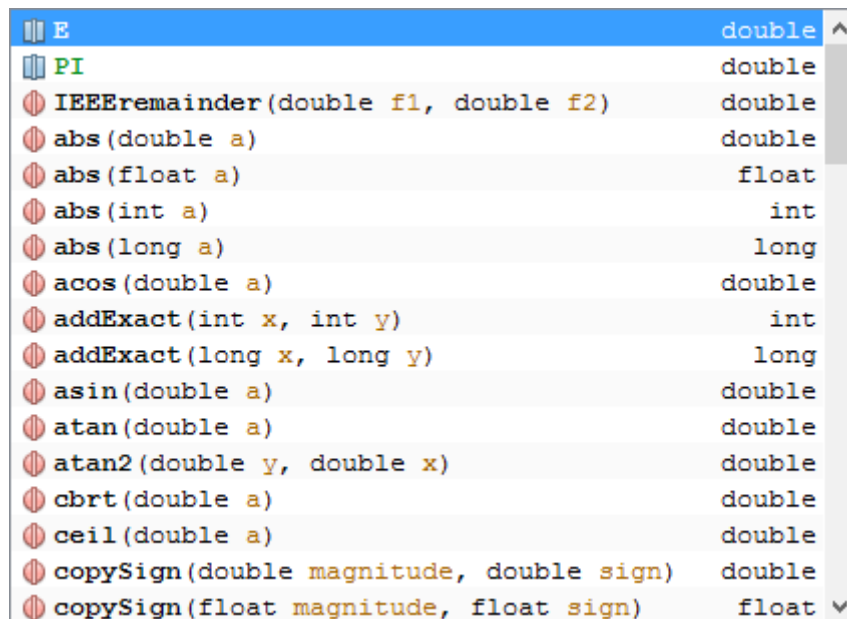
```
double x= Math.pow(3,3);
```

Además posee dos constantes, que son:

double E -> El número e (2, 7182818245...)

double PI -> El número Π (3,14159265...)

Por otro lado posee numerosos métodos como:



E	double
PI	double
IEEEremainder(double f1, double f2)	double
abs(double a)	double
abs(float a)	float
abs(int a)	int
abs(long a)	long
acos(double a)	double
addExact(int x, int y)	int
addExact(long x, long y)	long
asin(double a)	double
atan(double a)	double
atan2(double y, double x)	double
cbrt(double a)	double
ceil(double a)	double
copySign(double magnitude, double sign)	double
copySign(float magnitude, float sign)	float

8. LITERALES

A la hora de tratar con valores de los tipos de datos simples (y Strings) se utiliza lo que se denomina “literales”. Los literales son elementos que sirven para representar un valor en el código fuente del programa.

En Java existen literales para los siguientes tipos de datos:

- Lógicos (boolean).
- Carácter (char).
- Enteros (byte, short, int y long).
- Reales (double y float).
- Cadenas de caracteres (String).

8.1 Literales lógicos

Son únicamente dos, las palabras reservadas *true* y *false*.

Ejemplo: boolean activado = false;

8.2 Literales enteros

Los literales de tipo entero: *byte*, *short*, *int* y *long* pueden expresarse en decimal (base 10), octal (base 8) o hexadecimal (base 16). Además, puede añadirse al final del mismo la letra *L* para indicar que el entero es considerado

como `long` (64bits).

En Java, el compilador identifica un entero decimal (base 10) al encontrar un número cuyo primer dígito es cualquier símbolo decimal excepto el cero (del 1 al 9). A continuación pueden aparecer dígitos del 0 al 9.

La letra *L* al final de un literal de tipo entero puede aplicarse a cualquier sistema de numeración e indica que el número decimal sea tratado como un entero largo (de 64 bits). Esta letra *L* puede ser mayúscula o minúscula, aunque es aconsejable utilizar la mayúscula ya que de lo contrario puede confundirse con el dígito uno (1) en los listados.

Ejemplo:

```
long max1 = 9223372036854775807L; //valor máximo para un entero largo (64 bits)
```

8.3 Literales reales

Los literales de tipo real sirven para indicar valores *float* o *double*. A diferencia de los literales de tipo entero, **no pueden expresarse en octal o hexadecimal**.

Existen dos formatos de representación: mediante su parte entera, el punto decimal (.) y la parte fraccionaria; o mediante notación exponencial o científica:

Ejemplos equivalentes:

```
3.1415
0.31415e1
.31415e1
0.031415E+2
.031415e2
314.15e-2
31415E-4
```

Al igual que los literales que representan enteros, se puede poner una letra como sufijo. Esta letra puede ser una *F* o una *D* (mayúscula o minúscula indistintamente).

F --> Trata el literal como de tipo *float*.

D --> Trata el literal como de tipo *double*.

Ejemplo:

```
3.1415F
.031415d
```

8.4 Literales carácter

Los literales de tipo carácter se representan siempre entre comillas simples.

Entre las comillas simples puede aparecer:

- Un **símbolo** (letra) siempre que el carácter esté asociado a un código Unicode.
Ejemplos: 'a', 'B', '{', 'ñ', 'á'.
- Una **“secuencia de escape”**. Las secuencias de escape son combinaciones del símbolo contrabarra \ seguido de una letra, y sirven para representar caracteres que no tienen una equivalencia en forma de símbolo.

Las posibles secuencias de escape son:

\n -----> Nueva Línea.
\t -----> Tabulador.
\r -----> Retroceso de Carro.
\f -----> Comienzo de Pagina.
\b -----> Borrado a la Izquierda.
\ \ -----> El carácter barra inversa (\).
\' -----> El carácter prima simple (').
\" -----> El carácter prima doble o bi-prima (").

Por ejemplo:

Para imprimir una diagonal inversa se utiliza: \

Para imprimir comillas dobles en un String se utiliza: \"

8.5 Literales cadenas

Los **Strings** o **cadenas de caracteres** no forman parte de los tipos de datos elementales en Java, sino que son instanciados a partir de la clase `java.lang.String`, pero aceptan su inicialización a partir de literales de este tipo, por lo que se tratan en este punto.

✎ Un literal de tipo string va encerrado entre comillas dobles (") y debe estar incluido completamente en una sola línea del programa fuente (no puede dividirse en varias líneas).

Entre las comillas dobles puede incluirse cualquier carácter del código Unicode (o su código precedido del carácter \) además de las secuencias de escape vistas anteriormente en los literales de tipo carácter.

Así, por ejemplo, para incluir un cambio de línea dentro de un literal de tipo string deberá hacerse mediante la secuencia de escape \n :

Ejemplo:

```
System.out.println("Primera línea\nSegunda línea del string\n");  
System.out.println("Hola");
```

La visualización del *string* anterior mediante *println()* produciría la siguiente salida por pantalla:

```
Primera línea  
Segunda línea del string  
Hola
```

La forma de incluir los caracteres: comillas dobles (") y contrabarra (\) es mediante las secuencias de escape \" y \\ respectivamente (o mediante su código Unicode precedido de \).

🔊 Si el string es demasiado largo y debe dividirse en varias líneas en el fichero fuente, puede utilizarse el operador de concatenación de strings (+) de la siguiente forma:

```
"Este String es demasiado largo para estar en una línea del " +  
"fichero fuente y se ha dividido en dos."
```

9. ESTRUCTURAS ALTERNATIVAS

Como ya vimos, las estructuras alternativas son construcciones que permiten alterar el flujo secuencial de un programa, de forma que en función de una condición o el valor de una expresión, el mismo pueda ser desviado en una u otra alternativa de código.

Las estructuras alternativas disponibles en Java son:

- Alternativa if-else (simple y doble)
- Alternativa switch (compuesta)

9.1 Alternativa simple y doble

La alternativa simple se codifica de la siguiente forma:

Código	Ordinograma
<pre>if (expresión) { Acciones; }</pre> <p>El bloque de Acciones se ejecuta si, y sólo si, la expresión (que debe ser lógica) se evalúa a true, es decir, se cumple una determinada condición.</p> <pre>if (cont == 0) System.out.println("he llegado a cero");</pre>	<pre> graph TD Start(()) --> Condicion{Condicion} Condicion -- SI --> Acciones[Acciones] Condicion -- NO --> Exit(()) Acciones --> Exit Exit --> End(()) </pre>

✦ Si dentro del if o del else solo hay una instrucción no es necesario poner las llaves { }

La alternativa doble se codifica de la siguiente forma:

Código	Ordinograma
<pre>if (expresión) { Acciones_SI; } else { Acciones_NO; }</pre> <p>El bloque de Acciones_SI se ejecuta si, y sólo si, la expresión se evalúa a true. Y en caso contrario, si la expresión se evalúa a false, se ejecuta el bloque de</p>	<pre> graph TD Start(()) --> Condicion{Condicion} Condicion -- SI --> Acciones_SI[Acciones] Condicion -- NO --> Acciones_NO[Acciones] Acciones_SI --> Exit(()) Acciones_NO --> Exit Exit --> End(()) </pre>

Acciones_NO.

```
if (cont == 0)
{
    System.out.println("he llegado a
cero");
}
else
{
    System.out.println("no he llegado a
cero");
}
```

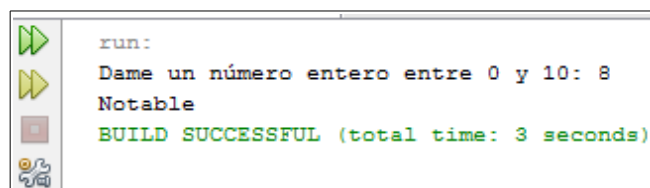
⚡ Recordad que el operador relacional para comprobar si son iguales es ==, no un solo = que corresponde con el operador de asignación. Este error no lo detecta el compilador y es difícil de averiguar.

En muchas ocasiones, se anidan estructuras alternativas if-else, de forma que se pregunte por una condición si anteriormente no se ha cumplido otra sucesivamente.

Por ejemplo: supongamos que realizamos un programa que muestra la nota de un alumno en la forma (insuficiente, suficiente, bien, notable o sobresaliente) en función de su nota numérica. Podría codificarse de la siguiente forma:

```
15 public class Nota {
16
17     public static void main(String[] args) throws IOException{
18
19         BufferedReader stdin;
20         stdin=new BufferedReader(new InputStreamReader(System.in));
21
22         int nota;
23
24         // Suponemos que el usuario introduce el número correctamente.
25         // No hacemos comprobación.
26         System.out.print("Dame un número entero entre 0 y 10: ");
27
28         nota = Integer.parseInt(stdin.readLine());
29
30         if (nota<5)
31             System.out.println("Insuficiente");
32         else
33             if (nota<6)
34                 System.out.println("Suficiente");
35             else
36                 if (nota<7)
37                     System.out.println("Bien");
38                 else
39                     if (nota<9)
40                         System.out.println("Notable");
41                     else
42                         System.out.println("Sobresaliente");
43     }
44 }
```

Siendo la salida:



```
run:
Dame un número entero entre 0 y 10: 8
Notable
BUILD SUCCESSFUL (total time: 3 seconds)
```


🔊 Es muy recomendable usar el tabulador en las instrucciones de cada bloque. Como se puede ver en el ejemplo, cada **else** está alineado con su **if** asociado, de esta forma es más fácil leer el código.

9.2 Alternativa múltiple

La alternativa múltiple se codifica de la siguiente forma:

Código	Ordinograma
<pre> switch (expresión) { case valor1: Acciones1; case valor2: Acciones2; ... case valorN: AccionesN; default: AccionesPorDefecto; } </pre>	

En primer lugar se evalúa la expresión cuyo resultado debe ser un **valor**. El programa comprueba el primer valor (valor1). En el caso de que el valor resultado de la expresión coincida con valor1, se ejecutarán las Acciones1.

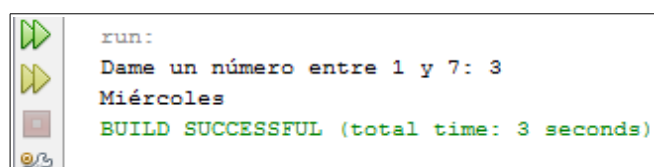
Pero ¡ojo! También se ejecutarían las instrucciones Acciones2 .. AccionesN hasta encontrarse con la palabra reservada **break**. Si el resultado de la expresión no coincide con valor1, evidentemente no se ejecutarían Acciones1, se comprobaría la coincidencia con valor2 y así sucesivamente hasta encontrar un valor que coincida o llegar al final de la construcción *switch*.

Si no coincide ningún valor, se ejecutarán las *AccionesPorDefecto*.

Un ejemplo sería el siguiente:

```
17 public static void main(String[] args) throws IOException{
18
19     BufferedReader stdin;
20     stdin=new BufferedReader(new InputStreamReader(System.in));
21
22     int dia;
23
24     System.out.print("Dame un número entre 1 y 7: ");
25
26     dia =Integer.parseInt(stdin.readLine());
27
28     switch (dia)
29     {
30         case 1:
31             System.out.println("Lunes");
32             break;
33         case 2:
34             System.out.println("Martes");
35             break;
36         case 3:
37             System.out.println("Miércoles");
38             break;
39         case 4:
40             System.out.println("Jueves");
41             break;
42         case 5:
43             System.out.println("Viernes");
44             break;
45         case 6:
46             System.out.println("Sábado");
47             break;
48         case 7:
49             System.out.println("Domingo");
50             break;
51         default:
52             System.out.println("Error el número debe estar entre 0 y 7 :");
53     }
54 }
```

Y la salida:



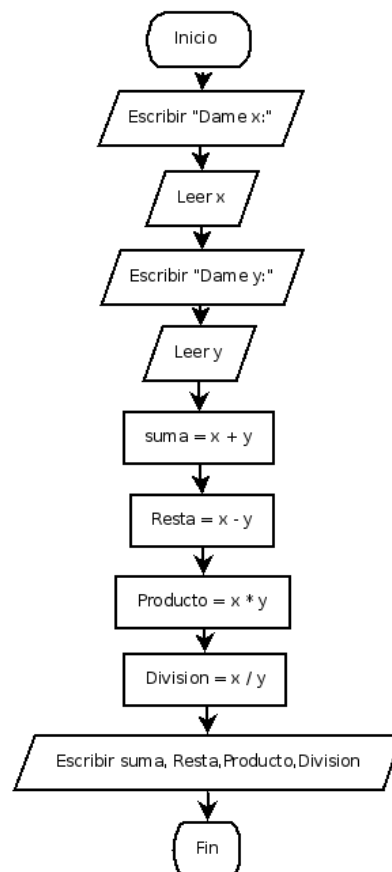
```
run:
Dame un número entre 1 y 7: 3
Miércoles
BUILD SUCCESSFUL (total time: 3 seconds)
```

10. EJEMPLOS

10.1 Ejemplo1

Programa que lea dos números, calcule y muestre el valor de sus suma, resta, producto y división. (ejercicio 4 de la UD3).

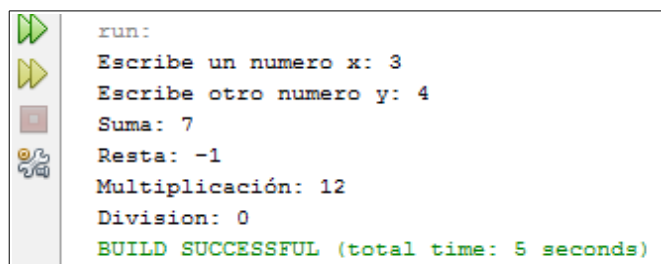
Ordinograma:



Código:

```
16 import java.io.*;
17
18 public class Opera {
19     public static void main (String [] args) throws IOException {
20         BufferedReader stdin = new BufferedReader (new InputStreamReader(System.in));
21         String teclado;
22
23         int x, y, suma, resta, mult, divi;
24
25         System.out.print("Escribe un numero x: ");
26         teclado = stdin.readLine();
27         x = Integer.parseInt(teclado);
28
29         System.out.print("Escribe otro numero y: ");
30         teclado = stdin.readLine();
31         y = Integer.parseInt(teclado);
32
33         suma = x + y;
34         resta = x - y;
35         mult = x * y;
36         // en este ejercicio no hacemos la comprobación de la división entre 0.
37         divi = x / y;
38
39         System.out.println("Suma: " + suma);
40         System.out.println("Resta: " + resta);
41         System.out.println("Multiplicación: " + mult);
42         System.out.println("Division: "+ divi);
43     }
44 }
```

Salida:

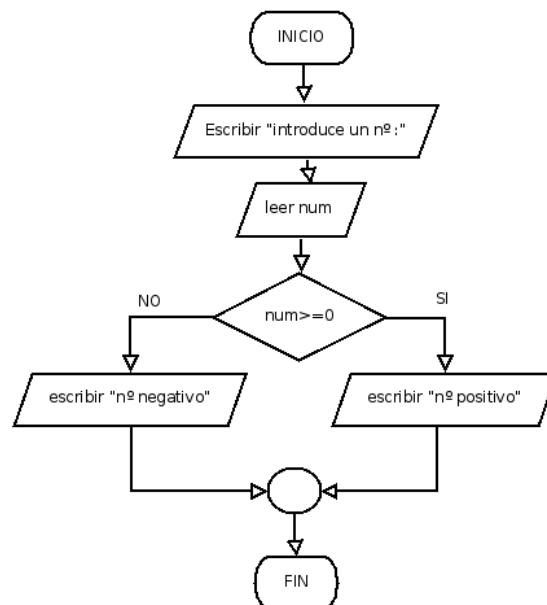


```
run:
Escribe un numero x: 3
Escribe otro numero y: 4
Suma: 7
Resta: -1
Multiplicación: 12
Division: 0
BUILD SUCCESSFUL (total time: 5 seconds)
```

10.2 Ejemplo 2

Programa de un programa que lee un número y me dice si es positivo o negativo, consideraremos el cero como positivo (ejercicio 12 de la UD3).

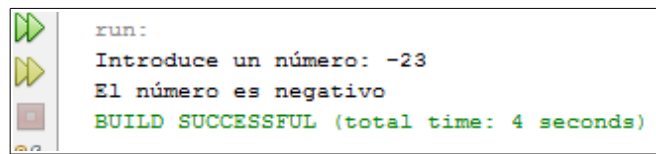
Ordinograma:



Código:

```
14 import java.io.*;
15
16 public class PosNegNul{
17     public static void main(String args[]) throws IOException{
18
19         BufferedReader stdin;
20         stdin = new BufferedReader(new InputStreamReader(System.in));
21         int num;
22
23         System.out.print("Introduce un número: ");
24         num = Integer.parseInt(stdin.readLine());
25
26         if(num >= 0)
27             System.out.println("El número es positivo");
28         else
29             System.out.println("El número es negativo");
30     }
31 }
```

Salida:



```
run:
Introduce un número: -23
El número es negativo
BUILD SUCCESSFUL (total time: 4 seconds)
```

11. AGRADECIMIENTOS

Apuntes actualizados y adaptados al CEEDCV a partir de la siguiente documentación:

- [1] Apuntes Programación de José Antonio Díaz-Alejo. IES Camp de Morvedre.