

```

import threading, socket, time
class sock(threading.Thread):
    def __init__(self):
        self.sck=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        threading.Thread.__init__(self)
        self.flag=1
    def connect(self, addr, port, func):
        try:
            self.sck.connect((addr, port))
            self.handle=self.sck
            self.todo=2
            self.func=func
            self.start()
        except:
            print "Error: Could not connect"
    def listen(self, host, port, func):
        try:
            self.sck.bind((host, port))
            self.sck.listen(5)
            self.todo=1
            self.func=func
            self.start()
        except:
            print "Error: Could not bind"
    def run(self):
        while self.flag:
            if self.todo==1:
                x, ho=self.sck.accept()
                self.todo=2
                self.handle=x
                self.func=self.func
            else:
                dat=self.handle.recv(4096)
                self.data=dat
                self.func()
    def send(self, data):
        self.handle.send(data)
    def close(self):
        self.flag=0
        self.sck.close()

```

DAM/DAW

PROGRAMACIÓN

02 - Java

Datos y Operadores

Indice

- Otra aplicación: suma de enteros
- Variables
- Tipos de datos en Java
- Números enteros
- Números en punto flotante
- Caracteres
- Valores lógicos
- Más sobre los tipos numéricos
- Literales y Constantes
- Operadores
- Conversiones de tipos
- API de Java
- Salida por consola
- Entrada de datos

```
import threading, socket, time
class sock(threading.Thread):
    def __init__(self):
        self.sock=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
        threading.Thread.__init__(self)
        self.flag=1
    def connect(self,addr,port,func):
        try:
            self.sock.connect((addr,port))
            self.handle=self.sock
            self.todo=2
            self.func=func
            self.start()
        except:
            print "Error:Could not connect"
            self.listen(self,host,port,func):
            try:
                self.sock.bind((host,port))
                self.sock.listen(2)
                self.todo=1
                self.func=func
                self.start()
            except:
                print "Error:Could not bind"
            def run(self):
                while self.flag:
                    if self.todo==1:
                        x,ho=self.sock.accept()
                        self.todo=2
                        self.client=ho
                        self.handle=x
                    else:
                        dat=self.handle.recv(4096)
                        self.data=dat
                        self.func()
                def send(self, data):
                    self.handle.send(data)
                def close(self):
                    self.flag=0
                    self.sock.close()
```

Otra aplicación: suma de enteros (I)

- Nuestra siguiente aplicación lee dos números **enteros** introducidos por el usuario, calcula la suma de los mismos y muestra el resultado
- El objeto del mismo **sólo** es introducir nuevos conceptos (tipos de datos, variables, operadores, entrada/salida de datos,...) que iremos desarrollando posteriormente a lo largo de la unidad

```
/**
 * Suma.java
 * Suma dos enteros introducidos por el usuario
 */
import java.util.Scanner;

class Suma {
    public static void main(String[] args) {
        // declaramos las variables que almacenarán los datos introducidos y el resultado de la suma
        int num1;
        int num2;
        int suma;

        // creamos el objeto Scanner para obtener la entrada del usuario
        Scanner entrada = new Scanner(System.in);
```

Otra aplicación: suma de enteros (II)

```
// leemos el primer número
System.out.print("Escriba el primer número: ");
num1 = entrada.nextInt(); // lectura

// leemos el segundo número
System.out.print("Escriba el segundo número: ");
num2 = entrada.nextInt(); // lectura

// calculamos y almacenamos la suma
suma = num1 + num2;

// mostramos el resultado
System.out.println("La suma de " + num1 + " y " + num2 + " es " + suma);
}
}
```

- Compilamos y ejecutamos la nueva aplicación:

```
bowman@hal:~/work/src/java$ javac Suma.java
```

```
bowman@hal:~/work/src/java$ java Suma
Escriba el primer número: -5
Escriba el segundo número: 32
La suma de -5 y 32 es 27
```

Otra aplicación: suma de enteros (III)

❏ Analizando el código: declaraciones *Import*

- Java dispone de un extenso conjunto de clases predefinidas que están a disposición del programador. Estas clases, contenidas en paquetes, conforman lo que se denomina la **biblioteca de clases** de Java o *Java Application Programming Interface (Java API)*
- En nuestro caso, vamos a hacer uso de una de estas clases, la clase *Scanner*, del paquete *java.util*, para capturar la entrada del usuario

```
import java.util.Scanner;
```

- La declaración *import* facilita la localización de dicha clase por parte del compilador y nos evita tener que indicar el nombre del paquete al que pertenece cada vez que la usemos
- Las declaraciones *import* van siempre al principio del archivo

Otra aplicación: suma de enteros (IV)

❑ Analizando el código: declaración de variables

- Una **variable** se asocia a una ubicación en la memoria de la computadora, en donde se puede guardar un valor para utilizarlo posteriormente dentro del programa
- Todas las variables tienen que **declararse** con un **nombre** (identificador) y un **tipo** antes de poder usarse. Por ejemplo, la declaración:

```
int num1;
```

indica que la variable de **nombre** *num1* está asociada a una localización de memoria en la que se podrán almacenar números de **tipo** entero (*int*)

- Se **declaran** tres variables, todas de **tipo primitivo** *int*, y **nombres**:
 - *num1* y *num2* almacenarán los números introducidos por el usuario
 - *suma* almacenará el resultado de la suma de los números anteriores

Otra aplicación: suma de enteros (V)

❑ Analizando el código: creación de objetos (*Scanner*)

- Para *capturar* la entrada del usuario, se emplea la clase *Scanner*. Esta es una clase de utilidad proporcionada por el API de Java
- La sentencia:

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

declara la variable de nombre *entrada* del **tipo referenciado** *Scanner*. A continuación, se crea un **objeto** de dicha clase mediante el operador *new* y se asigna dicho objeto a la variable mediante el operador asignación (=)

- La sentencia anterior podría reescribirse de la siguiente manera para mostrar, más claramente, ambos pasos:

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

Otra aplicación: suma de enteros (VI)

❏ Analizando el código: entrada de datos

- Una vez creadas las variables que almacenarán los datos y el objeto *Scanner* que nos permite leer la entrada del usuario, se hace uso del método *nextInt* de dicho objeto para obtener un número entero del usuario mediante el teclado y almacenarlo
- La sentencia:

```
num1 = entrada.nextInt(); // lectura
```

provoca que se llame al método *nextInt* del objeto *entrada*. Notar el empleo del punto (.) para *acceder* a los métodos del objeto. Este método es *bloqueante*, es decir, el programa se detendrá a la espera de que el usuario introduzca un número entero y pulse la tecla *Intro*

- Una vez obtenido el número, se *almacenará* en la posición de memoria asociada a la variable *num1* mediante el uso del *operador asignación* (=)

Otra aplicación: suma de enteros (VII)

❏ Analizando el código: variables y cálculos

- La sentencia:

```
suma = num1 + num2;
```

es una instrucción de asignación que calcula la suma de los valores *contenidos* en las variables *num1* y *num2*, y asigna el resultado a la variable *suma*

- Las sentencias de asignación tienen el siguiente formato:
variable = *expresión* ;
- Por *expresión* se entiende cualquier *predicado* computacional, sintácticamente válido, que devuelva un valor: llamadas a métodos, cálculo, una variable, combinaciones de los anteriores,...
- Primero se evalúa **toda** la *expresión* a la **derecha** del operador asignación (=) y, el resultado obtenido, se asigna a la *variable*

Otra aplicación: suma de enteros (y VIII)

❑ Analizando el código: salida por la consola

- Al igual que hicimos en el ejemplo anterior, vamos a emplear algunos métodos de la clase *System.out* para mostrar datos en la consola.
- La sentencia:

```
System.out.print("Escriba el primer número: ");
```

provoca que el mensaje “*Escriba el primer número:*” se imprima en pantalla

- La siguiente sentencia mostrará el resultado del cálculo realizado:

```
System.out.println("La suma de " + num1 + " y " + num2 + " es " + suma);
```

Fíjate como el operador suma (+), en este caso, permite **concatenar** los literales “*La suma de*”, “*y*”, “*es*” con los valores numéricos almacenados por las variables *num1*, *num2* y *suma*. Es lo que se conoce como **sobrecarga** del operador (su comportamiento varía según el tipo de los operandos)

Variables (I)

- Ya vimos que las **variables** son elementos del lenguaje que posibilitan que nuestros programas almacenen valores para su uso posterior
- Para crear una nueva variable (y poder usarla posteriormente) debemos **declararla**. La declaración de variables sigue el siguiente formato:

tipo nombre ;

```
float tipoDeInteres;  
int numArticulos;
```

- En el ejemplo anterior, *float* o *int*, establecen el **tipo** de variable y, *tipoInteres* o *numArticulos*, son sus **identificadores** o nombres
- Dependiendo del tipo indicado, la JVM, reservará una cantidad de espacio en **memoria** para esa variable. El nombre de la misma quedará “asociado” a dicha posición de memoria



Variables (II)

- El tipo determina también la “naturaleza” de los datos que se pueden almacenar en una variable (en realidad, en las posiciones de memoria “asociadas” a la variable) (caracteres, números,...) y cómo se codifican
- Java permite declarar más de una variable en la misma sentencia, siempre y cuando sean del mismo tipo:

```
int num1, num2, suma;
```

- Los **nombres** de las variables deben seguir las **reglas** generales para cualquier **identificador** en Java:
 - Comienzan por una letra, subrayado (_) ó dólar (\$)
 - Los siguientes caracteres pueden ser letras, dígitos, subrayado (_) ó dólar (\$)
 - Se distinguen mayúsculas de minúsculas
 - No hay una longitud máxima para el identificador

Variables (III)

- Los identificadores no podrán ser ninguna de las **palabras reservadas**:

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

- Por convención, se emplea notación *CamelCase* para los identificadores:
 - UpperCamelCase*, para los nombres de clases
 - lowerCamelCase*, para métodos, atributos, variables locales,...

Ejemplo: Google Java Style Guide: <https://google.github.io/styleguide/javaguide.html>

Variables (IV)

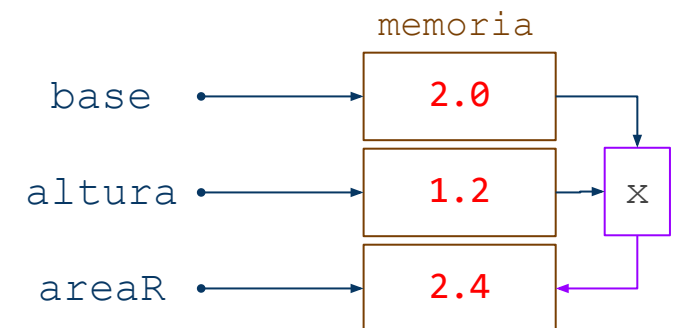
- Decimos que una variable está **inicializada** si ha sido declarada y contiene algún valor
- Los **atributos** de las clases se inicializan a un valor por defecto (dependiente del tipo) al declararlas
- Las **variables locales** no se inicializan al declararse. Por tanto, no las podremos “leer” hasta que se les haya asignado un valor inicial
- Una vez inicializada la variable, podremos usarla en posteriores sentencias de nuestro programa. A través de ella, accederemos al contenido de las posiciones de memoria que identifica, para leer o modificar su contenido.
- Para acceder al contenido de una variable (leer), no tenemos más que usar su identificador en cualquier expresión o sentencia

```
System.out.println("La suma de " + num1 + " y " + num2 + " es " + suma);
```

Variables (V)

- Mediante el operador de **asignación** (=) (no confundir con el operador de comparación ==) podremos **inicializar** con un valor o **modificar** el valor actual de cualquier variable
- De forma general, la asignación se realizará de la siguiente manera:
$$\text{variable} = \text{expresión};$$
 - se evalúa la expresión a la derecha del símbolo igual (=)
 - se guarda el valor resultante en la variable indicada a la izquierda
- Java permite inicializar una variable en el momento de su declaración asignando el valor de un literal (*estática*) o de una expresión (*dinámica*)

```
double base = 2.0, altura = 1.2; // estática
double areaR = base * altura;    // dinámica
```



Variables (VI)

- La lectura de variables no declaradas o no inicializadas, generará **errores de compilación**

- Uso de variable no declarada (no existente): al compilar, generará un error de *símbolo no encontrado*

```
int num1 = 5, suma;  
suma = num1 + num2;
```



```
... error: cannot find symbol  
      suma = num1 + num2;  
                      ^
```

- Uso de variable no inicializada (declarada pero sin un valor asignado): al compilar, generará un error de *variable no inicializada*

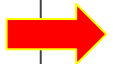
```
int num1 = 5, num2, suma;  
suma = num1 + num2;
```



```
... error: variable num2 might not have been initialized  
      suma = num1 + num2;  
                      ^
```


Variables (y VII)

- Se producirán también errores de compilación cuando intentemos asignar a una variable de un tipo, un valor de un tipo incompatible
 - En el siguiente ejemplo, se intenta asignar un valor *booleano* (*true/false*) a una variable de entera, produciéndose un error de *type mismatch*

<pre>int num1 = true;</pre>		Exception in ...: Unresolved compilation problem: Type mismatch: cannot convert from boolean to int
-----------------------------	---	--

- Por último, aunque volveremos sobre ello en posteriores unidades, es conveniente saber que las variables tienen un **alcance** que determina su **visibilidad** y su **vida**.
- Es decir, una variable declarada dentro de un bloque del lenguaje (por ejemplo, un método) sólo “existirá” y será “visible” dentro de ese bloque. Se creará en el momento de la declaración y se “destruirá” al abandonar el bloque donde se creó, siendo sólo accesible dentro del mismo

Tipos de datos en Java (I)

- A la hora de desarrollar nuestros propios programas, nos encontraremos con que dichas aplicaciones tendrán que trabajar con numerosos datos de diferente naturaleza: números (enteros, reales, ...), textos, caracteres individuales, valores lógicos sí/no,...
- Con objeto de facilitar el tratamiento de esos datos, **Java**, al igual que el resto de lenguajes de alto nivel, ofrece una serie de **tipos de datos** para categorizarlos.
- Estos tipos de datos definen **cómo** se almacenan internamente dichos datos y **qué** operaciones podemos realizar con ellos.
- Si bien internamente todos los datos son almacenados en formato binario, los esquemas de codificación empleados para su representación binaria difieren en función de su propia naturaleza (ó tipo)
- En el [*Anexo 02 \(Representación de la Información\)*](#) se describen estos esquemas de representación interna de los datos por el computador

Tipos de datos en Java (II)

- En Java distinguimos dos categorías de tipos de datos:
 - Tipos *primitivos*, representan un único *dato simple*. Establecen una codificación binaria y rango de valores específico. Son la base del almacenamiento en Java y de la definición de los atributos de las clases

<i>byte</i> , para números enteros (8 bits) <i>short</i> , para números enteros cortos (16 bits) <i>int</i> , para números enteros (32 bits) <i>long</i> , para números enteros grandes (64 bits)	<i>float</i> , para números reales (precisión simple) <i>double</i> , para números reales (precisión doble) <i>char</i> , para caracteres (UNICODE, 16 bits) <i>boolean</i> , para valores lógicos (<i>true/false</i>)
--	---

- Tipos *referenciados*, cuando el tipo de nuestra variable es una clase, *array*, interfaz... La variable correspondiente almacenará una “*dirección de memoria*” (*referencia*) a un *objeto* o *conjunto de valores*. En el ejemplo anterior, la variable *entrada* se emplea para almacenar una *referencia* a un objeto de tipo *Scanner*

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

Tipos de datos en Java (y III)

- Java, al igual que otros lenguajes como C++, C# o Haskell, es un lenguaje de **tipado estático**. Otros, como Python o PHP, son de **tipado dinámico**.
- Los lenguajes de *tipado estático*, normalmente compilados, nos obligan a establecer el tipo de una variable (**declaración**) antes de poder usarla. En los lenguajes de *tipado dinámico*, es en el proceso de asignación de un valor a una variable cuando se establece el tipo de datos de dicha variable (en función del valor recibido)
- Además, Java es de **tipado fuerte**. Esto implica que a una variable de un tipo dado, sólo se le pueden asignar valores de dicho tipo. En casos determinados, podremos realizar conversiones (implícitas o explícitas) entre diferentes tipos de datos
- Tanto el compilador de Java (en tiempo de compilación) como la JVM (en tiempo de ejecución), verificarán que la manipulación de los datos a través de las variables no violen las especificaciones del lenguaje

Números enteros

- Para almacenamiento de números enteros, Java define los tipos:

Tipo	Bits	Rango
byte	8	-128 a 127
short	16	-32.768 a 32.767
int	32	-2.147.483.648 a 2.147.483.647
long	64	-9.223.372.036.854.775.808 a 9.223.372.036.854.775.807

- Java no soporta enteros *unsigned* (sólo-positivos) como sí hacen lenguajes como C++
- El tipo más usado para almacenar enteros es *int*, aunque dependerá de cada uso particular
- Para facilitar la portabilidad, los tamaños de almacenamiento son fijos e independientes de la arquitectura (esto es aplicable a todos los tipos de datos primitivos)

Números en punto flotante (I)

- La representación de **punto flotante** es una forma de notación científica usada en los computadores con la cual se pueden representar **números reales** extremadamente grandes y pequeños de una manera muy eficiente y compacta
- El estándar actual para la representación binaria de números en coma flotante es el **IEEE 754**
- Java soporta dos tipos para números de punto flotante, *float* y *double*, que representan números de simple y doble precisión respectivamente

Tipo	Bits	Rango
float	32	aprox $\pm 3.40282347E+38$ (6-7 dígitos decimales significativos)
double	64	aprox $\pm 1.79769313486231570E+308$ (15 dígitos decimales significativos)

- *double* suele emplearse más a menudo. Por ejemplo, muchos de los métodos de la clase **Math** del API de Java devuelven un valor *double*

Números en punto flotante (y II)

❏ Ejemplo: cálculo de la hipotenusa de un triángulo rectángulo

```
//: Hipotenusa.java
/**
 * Calcula la hipotenusa de un triángulo rectángulo
 * @author bowman
 * @version 1.0
 */
class Hipotenusa {
    public static void main(String[] args) {
        double a, b, h;

        a = 3; // cateto a
        b = 4; // cateto b

        // Invocamos el método sqrt de la clase Math para obtener la raíz cuadrada de (a2 + b2)
        h = Math.sqrt(a*a + b*b);

        System.out.println("La hipotenusa del triángulo es " + h);
    }
}
/* Output:
La hipotenusa del triángulo es 5.0
*///:~
```

Caracteres (I)

- Java proporciona el tipo *char* para el almacenamiento de caracteres
- En Java, los caracteres son representados mediante la codificación UNICODE, de forma que se puede usar cualquiera de los caracteres de los diferentes lenguajes humanos
- A diferencia de otros lenguajes, como C o C++, en los que 8 bits son suficientes para almacenar las representaciones ASCII de los caracteres, el empleo de UNICODE supuso la redimensión de dicho tipo
- En Java, el tipo *char* es de **16 bits** y permite almacenar valores numéricos **enteros positivos** en el **rango de 0 a 65535**
- Resaltar que el tipo *char* es un tipo numérico, por lo que se pueden realizar sobre él operaciones aritméticas (suma, resta,...)
- Los literales en Java representando caracteres individuales irán encerrados entre **comillas simples** `' '`. Por ejemplo: `'a'`, `'4'`, `'\n'`,...

Caracteres (y II)

Ejemplo: aritmética con caracteres

```
//: CharAritDemo.java
/** Aritmética con caracteres
 * Las variables char pueden manejarse como enteros
 */
class CharAritDemo {
    public static void main(String[] args) {
        char ch;

        ch = 'X';    // Asignamos el caracter X
        System.out.println("ch contiene " + ch);

        ch += 1;     // Incrementamos en 1 su valor actual
        System.out.println("ch ahora contiene " + ch);

        ch = 90;     // Asignamos un valor numérico
        System.out.println("ch ahora contiene " + ch);
    }
}

/* Output:
ch contiene X
ch ahora contiene Y
ch ahora contiene Z
*///:~
```

Valores lógicos (I)

- El tipo *boolean* representa los valores lógicos verdadero/falso
- En Java, las palabras *true* y *false* son palabras reservadas que se emplean para establecer los dos *únicos valores posibles* de una variable *boolean*
- Usos típicos del tipo *boolean* son:
 - El resultado de cualquier operación que emplee operadores relacionales (>, <, >=, ==, ...) será de tipo *boolean*
 - Control de sentencias condicionales (*if*)
 - Creación de variables de dominio binario (*flags*): verdadero/falso, on/off, activo/inactivo, sí/no...
 - Por ejemplo: `usuarioRegistrado`, `motorEncendido`,...
- El siguiente ejemplo muestra algunas de estas situaciones...

Valores lógicos (y II)

```
//: BoolDemo.java

/**
 * Ejemplos de uso de boolean
 * @author bowman
 * @version 1.0
 */

class BoolDemo {
    public static void main(String[] args) {
        boolean b;

        b = false;
        System.out.println("b es " + b);
        b = true;
        System.out.println("b es " + b);

        // un boolean puede controlar un condicional if
        if(b)
            System.out.println("Esto se ejecuta");

        b = false;
        if(b)
            System.out.println("Esto no se ejecuta");
    }
}
```

```
/* El resultado de un operador relacional es un valor
   boolean */
System.out.println("10 > 9 es " + (10>9));

}

/* Output:
b es false
b es true
Esto se ejecuta
10 > 9 es true
*///:~
```

Más sobre los tipos numéricos (I)

❏ Una nota acerca de la **precisión**...

- Tenemos que tener en cuenta que, tanto el tipo *float* como el *double*, disponen de un número limitado de bits para representar la mantisa y exponente de nuestros números reales. Es decir, en la mayoría de los casos, serán aproximaciones a dichos números
- Supongamos que ejecutamos la siguiente sentencia:

```
System.out.println(0.3 + 0.6);
```

- Sorprendentemente, el resultado de ejecutar la sentencia anterior sería:

```
0.8999999999999999
```

y no **0,9** como cabría esperar. Esto es debido a que no existe una representación exacta de dicho valor en el formato IEEE754 (necesitaríamos un número infinito de bits)

Más sobre los tipos numéricos (II)

- De igual modo, si ejecutáramos:

```
double n1=0.3, n2=0.6, suma = n1 + n2;  
System.out.println("La suma de n1 y n2 es 0.9? " + (suma == 0.9));
```

El resultado sería:

```
La suma de n1 y n2 es 0.9? false
```

- Como vemos, esto puede dar lugar a ciertos comportamientos no deseados, especialmente en aplicaciones financieras, donde trabajamos con un número fijo de posiciones decimales
- Java proporciona la clase *BigDecimal* que nos permite definir número de **dígitos de precisión** decimal y **tipo de redondeo** al trabajar con reales.
Por ejemplo, podríamos redondear a 2 dígitos de las siguientes maneras:

```
// usando double: suma = 0.8999999999999999 --> suma1 = 0.9  
double suma1 = Math.round( suma*100)/100.0;  
// usando BigDecimal: suma = 0.8999999999999999 --> suma2 = 0.90  
BigDecimal suma2 = new BigDecimal( suma, new MathContext(2, RoundingMode.HALF_UP));
```

Más sobre los tipos numéricos (III)

❏ Una nota acerca de los **desbordamientos**...

- Hemos visto como las variables, dependiendo de su tipo, tienen un espacio máximo reservado en memoria y, por tanto, un rango de valores posibles. Pero, ¿qué pasa si nos salimos de dicho rango?
- Supongamos el siguiente caso en el que intentamos asignar a una variable de tipo *int* un valor fuera de su rango [-2.147.483.648 a 2.147.483.647]:

```
int n = 2147483650;
```

- Como era de esperar, el **compilador** nos informará de dicha situación y abortará la compilación:

```
.....: error: integer number too large: 2147483650
      int n = 2147483650;
              ^
1 error
```

Más sobre los tipos numéricos (y IV)

- Pero, ¿y si el desbordamiento se produce en tiempo de ejecución?
- En el siguiente caso, vamos a provocar una situación de desbordamiento por arriba (*overflow*) a una variable de tipo *byte*, de rango [-128, 127]:

```
byte b = 127;    // Asignamos el valor más alto del rango
b += 1;          // Incrementamos su valor en 1
System.out.println(b);
```

- Contrariamente a lo que podríamos esperar, no se produce un error de ejecución y obtenemos como resultado el número **-128!!!**
- Con objeto de evitar estos errores de desbordamiento que podrían provocar la corrupción de las zonas de memoria adyacentes, los rangos de valores de los tipos de datos en Java son **circulares**. Es decir, cuando sobrepasamos el valor máximo (*overflow*) se continúa por el extremo inferior del rango. Y viceversa en una situación de *underflow*
- Es **importante** tenerlo en cuenta al operar con valores numéricos

Literales y Constantes (I)

❏ Literales

- En general, el término *literal* se refiere a valores fijos representados de forma que puedan ser leídos por nosotros. Por ejemplo, el número 100, o el texto “esto es un ejemplo de literal”
- Empleamos los literales para hacer asignaciones a variables y para definir **constantes** (variables cuyo valor no se puede modificar)
- Podemos crear literales de cualquiera de los tipos primitivos. La forma de representarlos dependerá de su tipo:
 - **Caracteres**, entre comillas simples: 'a', '3', '%',...
 - **Enteros**, en forma numérica, con o sin signo: 100, -43,...
 - **De punto flotante**, con el punto decimal: 3.5, -128.2,...También se puede usar notación científica: 2.3e4, 1.25e-2,...

Literales y Constantes (II)

- Desde JDK7, podemos insertar uno o más *underscores* (_) entre los dígitos de un número entero o en punto flotante. Esto puede hacer más fácil la lectura de números con múltiples dígitos como referencias, cuentas bancarias,... Por ejemplo: 2910_2116_54_7166211402
- Por defecto, los literales enteros son de tipo *int*. Pueden ser asignados directamente a variables de tipo *char*, *byte* o *short* siempre y cuando su valor no exceda el rango de la variable. Un literal entero siempre se puede asignar a una variable de tipo *long*
- Podemos crear un literal de tipo *long* añadiendo una *l* (ó *L*) al número: 125L, -32l, ...
- Por defecto, los literales de punto flotante son de tipo *double*. Podemos crear un literal de tipo *float* añadiendo una *f* (ó *F*) al número: 12.5F, -1.33E5f, ...

Literales y Constantes (III)

❑ Literales en Hexadecimal, Octal y Binario

- En ocasiones, es más aconsejable expresar un valor numérico en base 8 ó 16 en lugar de base 10. Por ejemplo: direcciones IPv6, direcciones de memoria, componentes RGB de colores,...
- Java nos permite expresar los literales de tipo entero empleando notación octal o hexadecimal en lugar de decimal:
 - **Octal**, anteponiendo un **0** al valor en octal: **011** (9 en decimal)
 - **Hexadecimal**, anteponiendo **0x** (ó **0X**) al valor en hexadecimal: **0xFF** (255 en decimal)
- Desde JDK7, es posible expresar un literal entero usando notación binaria. Para ello, precederemos el número binario con **0b** (ó **0B**). Por ejemplo: **0b1100** (12 en decimal)

Literales y Constantes (IV)

❑ Strings

- Java soporta otro tipo de literal: el *string*. Es una cadena o secuencia de caracteres encerrada entre comillas dobles ("): "esto es una prueba"
- Además de caracteres alfanuméricos, el *string* puede incluir caracteres especiales denominados **caracteres de escape**, que van precedidos de un *backslash* (\) y tienen un significado particular. Por ejemplo:

secuencia de escape	significado	secuencia de escape	significado
\n	nueva línea	\b	retroceso
\r	retorno de carro	\"	comilla doble
\f	avance línea	\'	comilla simple
\t	tabulación	\\	barra invertida

- Las secuencias de escape, como caracteres que son, pueden asignarse individualmente a variables de tipo *char*

Literales y Constantes (V)

❏ Ejemplo: uso de secuencias de escape

```
//: EscapeDemo.java
/**
 * Ejemplos de uso de secuencias de escape
 */
class EscapeDemo {
    public static void main(String[] args) {
        char c = '\t'; // secuencia de tabulación

        System.out.println("\"Este texto\nse va a imprimir\nen varias líneas\"");

        System.out.println("Estas\tpalabras\testán\ttabuladas");

        System.out.println("Y" + c + "estas" + c + "también");
    }
}

/* Output:
Este texto
se va a imprimir
en varias líneas"
Estas   palabras       están   tabuladas
Y       estas   también
*///:~
```

Literales y Constantes (y VI)

❏ Constantes

- Las **constantes** son un tipo especial de variables que, una vez asignado su primer valor, ya no se podrá modificar y permanecerá inmutables.
- La declaración de constantes es similar a la de cualquier variable. Simplemente añadiremos el modificador *final* al comienzo de la misma:

final tipo nombre ;

```
final float TIPO_IVA = 0.21f;  
final double PI = 3.141592;
```

- El modificador *final* imposibilita que las variables puedan ser modificadas tras inicializarse (error de compilación). Veremos que dicho modificador puede ser aplicado también a clases y métodos
- Por convención, los nombres de las constantes van en **mayúsculas**. En nombres compuestos, se empleará el signo (_) para separar las palabras

Operadores (I)

- Los *operadores* son símbolos que empleamos para indicar la realización de algún tipo de manipulación aritmética o lógica sobre los datos
- Java define las siguientes clases generales de operadores:
 - *aritméticos* (+, -, ...)
 - *de bit* (>>, <<, ...)
 - *relacionales* (>, >=, <, ...)
 - *lógicos* (&&, ||, ...)
- Adicionalmente, Java define los siguientes operadores:
 - de *asignación* (=, +=, -=, ...)
 - condicional (*?:*) (*unidad 4*)
 - tipo de instancia (*instanceOf*) (*unidad 3*)

Operadores (II)

Operadores aritméticos

- Los *operadores aritméticos* se emplean en expresiones matemáticas del mismo modo que se emplean en álgebra

operador	descripción
+/-	signo (positivo/negativo)
+	suma
-	resta
*	multiplicación
/	división
%	módulo (resto de la división entera)
++	incremento (+1)
--	decremento (-1)

Operadores (III)

- El operador **-** unario nos permite modificar el signo de una variable:

```
int n = 1, m;  
m = -n;           //--> m vale -1
```

- El operador módulo **%** devuelve el resto de una división entera:

```
int n = 2, m = 5, r;  
r = m % n;        //--> r vale 1
```

- Los operadores **++** (*incremento*) y **--** (*decremento*) nos permiten aumentar o disminuir el valor actual de una variable en una unidad, algo muy frecuente, por ejemplo, en el control de bucles:

```
int n = 1;         //--> n vale 1  
n++;              //--> n vale 2 --> equivalente a:  n = n + 1
```

La sentencia anterior **n++** sería equivalente a realizar la asignación:

```
n = n + 1;
```


Operadores (IV)

- Ambos operadores *incremento* (++) y *decremento* (--) pueden utilizarse tanto de forma *prefija* (++n), como *postfija* (n++), aunque con resultados diferentes.
- En el caso de la forma *prefija*, primero se actualizará el valor de la variable y, posteriormente, se usará el valor actualizado en la expresión:

```
int n = 10, m;  
m = ++n;           //--> n vale 11 y m vale 11  
m = ++n * 2;       //--> n vale 12 y m vale 24
```

- En el caso de la forma *postfija*, primero se evaluará la expresión con el valor actual de la variable y, posteriormente, se actualizará el valor de la misma:

```
int n = 10, m;  
m = n++;           //--> n vale 11 y m vale 10  
m = n++ * 2;       //--> n vale 12 y m vale 22
```

Operadores (V)

❏ Operadores relacionales

- Los *operadores relacionales* nos permiten construir expresiones para *comparar* valores.

operador	descripción
==	igual que
!=	distinto que
>	mayor que
>=	mayor o igual que
<	menor que
<=	menor o igual que

- El resultado de la expresión siempre será un valor *boolean* (*true/false*)

Operadores (VI)

Operadores lógicos

- Los *operadores lógicos* nos permiten crear expresiones lógicas complejas, normalmente formadas por expresiones que incluyen operadores relacionales

operador	descripción
&	AND (Y lógico)
	OR (O lógico)
^	XOR (O exclusivo)
	OR (<i>short-circuit</i>)
&&	AND (<i>short-circuit</i>)
!	NOT (negación)

- Tanto los operandos, como el resultado de la expresión lógica, serán valores de tipo *boolean* (*true/false*)

Operadores (VII)

- Tablas de verdad de los operadores lógicos

& (AND)		
Operandos		Resultado
True	True	True
True	False	False
False	True	False
False	False	False

(OR)		
Operandos		Resultado
True	True	True
True	False	True
False	True	True
False	False	False

^ (XOR)		
Operandos		Resultado
True	True	False
True	False	True
False	True	True
False	False	False

! (NOT)	
Operando	Resultado
True	False
False	True

```
boolean b;  
int a = 2, b = 4;  
  
b = (3<=5) & !(a == b);           //--> b = true  
b = (3>5) ^ (a != b);             //--> b = true  
b = ((a>5) | (2*a == b)) & false;  //--> b = false  
b = (a>5) & (2*a == b) & !false;  //--> b = false  
b = (a>5) | (2*a != b) | !false;  //--> b = true
```

Operadores (VIII)

- **Operadores lógicos *Short-Circuit* `&&` y `||`**
- Java proporciona versiones *short-circuit* de los operadores lógicos *AND* (`&`) y *OR* (`|`) que pueden ser usados para generar código más eficiente.
- Mientras que los operadores *AND* (`&`) y *OR* (`|`) estándar evalúan **siempre** cada uno de los operandos de una expresión lógica, su versiones *short-circuit*, evaluarán el segundo operando **sólo** si es necesario
 - El operador *AND short-circuit* es `&&`
Si el primer operando es **false**, ya no se evalúa el segundo, pues el resultado de la expresión siempre será *false*
 - El operador *OR short-circuit* es `||`
Si el primer operando es **true**, ya no se evalúa el segundo, pues el resultado de la expresión siempre será *true*

Operadores (IX)

Operadores *Bitwise*

- Los *operadores bitwise*, que se pueden aplicar sobre cualquier valor numérico entero (*byte*, *char*, *short*, *int*, *long*), permiten realizar operaciones lógicas a nivel de **bit**

operador	descripción
&	AND (Y binario)
	OR (O binario)
^	XOR (O exclusivo binario)
<<	LS (Desplazamiento Izq)
>>	ARS (Desplazamiento Dcha Aritmético) (preserva signo)
>>>	LRS (Desplazamiento Dcha Lógico) (relleno con 0's)
~	COMP (complementario)

En decimal		En binario	
Expresión	Resultado	Expresión	Resultado
5 & 12	4	00000101 & 00001100	00000100
5 12	13	00000101 00001100	00001101
5 ^ 12	9	00000101 ^ 00001100	00001001
5 << 1	10	00000101 << 00000001	00001010
5 << 2	20	00000101 << 00000010	00010100
5 << 3	40	00000101 << 00000011	00101000
5 >> 1	2	00000101 >> 00000001	00000010

Operadores (X)

❏ Operadores de asignación *compuestos*

- Ya conocemos el operador de asignación de Java (=). Veamos otras características de este operador
- En Java, este operador permite la creación de “*cadena de asignación*”, de forma que podemos realizar una asignación múltiple a un grupo de vbles:

```
int a, b, c;  
a = b = c = 10;           //--> a, b y c valen 10
```

- Java dispone de un conjunto de *operadores de asignación compuestos*. Mediante un **único** operador, combinaremos una operación aritmética y una asignación, mejorando la legibilidad del código

```
int a = 10;      //--> a vale 10  
a += 4;          //--> a vale 14  (es equivalente: a = a + 4; )  
a /= 2;          //--> a vale 7   (es equivalente: a = a / 2; )
```

Operadores (XI)

❑ Operadores de *asignación compuestos*

operador	descripción
<code>+=</code>	suma
<code>-=</code>	resta
<code>*=</code>	multiplicación
<code>/=</code>	división
<code>%=</code>	módulo
<code><<=</code>	left-shift
<code>>>=</code>	right-shift
<code>>>>=</code>	logical-right-shift
<code>&=</code>	bitwise AND
<code> =</code>	bitwise OR
<code>^=</code>	bitwise XOR

Antes de realizar la operación con el *valor actual* de la variable, se “resolverá” la expresión a su derecha

La expresión:

```
a *= b + c
```

es equivalente a:

```
a = a * (b + c)
```

y **no** a:

```
a = a * b + c
```


Operadores (XII)

❏ Precedencia de operadores

- La **asociatividad** y la **precedencia** de los operadores establecen la manera en la que se evalúan las expresiones, especialmente cuando intervienen diversos operandos y operadores

$$3 + 4 - 2 = 5$$

suma y resta tienen la misma precedencia y se asocian por la izquierda, así que el orden en que se resolvería sería: primero $3+4$ y, al resultado, restarle 2

$$3 + 4 * 2 = 11$$

el producto tiene mayor precedencia que la suma y se asocia por la izquierda, así que el orden en que se resolvería sería: primero $4*2$ y, al resultado, sumarle 3

$$(3 + 4) * 2 = 14$$

$$(3 + 4) * (4 - (2 + 1)) = 7$$

el uso de **paréntesis** nos permite modificar el orden en que se evalúa la expresión, desde los paréntesis más internos a los más externos

Operadores (XIII)

❑ Tabla de orden de precedencia (de mayor a menor)

categoría	operador	asociatividad
postix	<i>expr</i> [++ --]	izq -> drch
unario	[++ -- + - ~ !] <i>expr</i>	drch -> izq
Multiplicativo	* / %	izq -> drch
Aditivo	+ -	izq -> drch
Relacional	< > <= >= instanceof	izq -> drch
Igualdad	== !=	izq -> drch
<i>bitwise</i> AND	&	izq -> drch
<i>bitwise</i> XOR	^	izq -> drch
<i>bitwise</i> OR		izq -> drch
AND	&&	izq -> drch
OR		izq -> drch
Condicional	?:	drch -> izq
Asignación	= += -= *= /= %= ^= = <<= >>= >>>=	drch -> izq

Operadores (y XIV)

❑ Paréntesis para agrupar subexpresiones

- Los paréntesis se utilizan para agrupar términos en las expresiones en Java, de la misma manera que en las expresiones algebraicas
- Por ejemplo, para multiplicar a por $b + c$, escribimos:

```
a * (b + c)
```

De no hacerlo así, debido a la mayor precedencia de la multiplicación respecto a la suma, se haría primero $a * b$ y, al resultado, se le suma c

- Si una expresión contiene **paréntesis anidados**, como:

```
a / ((b + c) * d)
```

se evalúa **primero** la expresión en el conjunto más interno:

```
a / ((b + c) * d)
```


$$\frac{a}{(b + c) \times d}$$

Conversiones de tipos (I)

- En programación, es habitual asignar el valor de una variable de un tipo a otra variable de un tipo diferente. Por ejemplo:

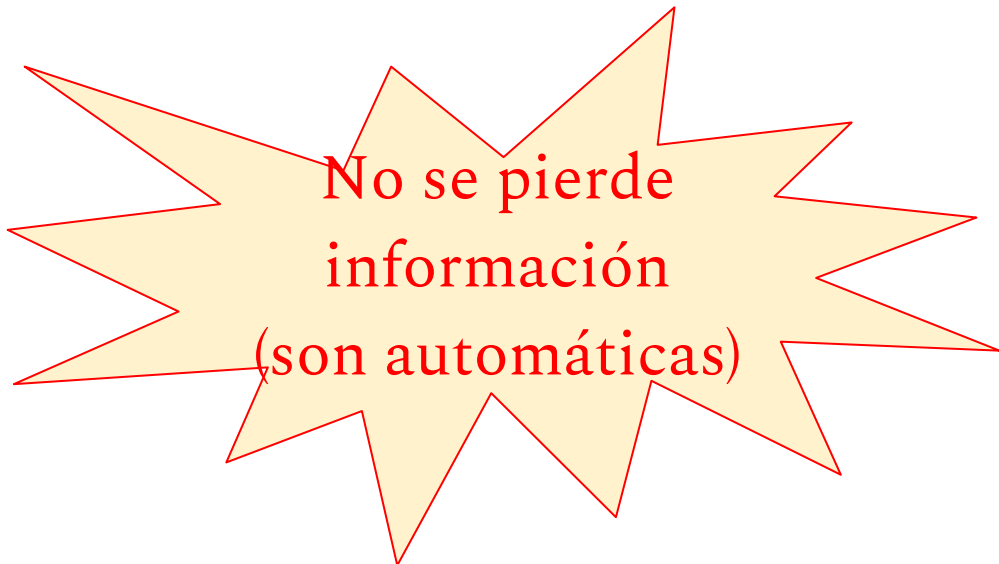
```
int i = 10;  
float f;  
f = i           // asignación de un int a un float --> f vale 10.0
```

- Cuando usamos tipos *compatibles*, el valor de la derecha se convierte automáticamente al tipo de la izquierda. En el ejemplo anterior, el valor de *i* se convierte a *float* y, a continuación, se asigna a *f*
- Sin embargo, debido a la naturaleza *fuertemente tipada* de Java, estas *conversiones implícitas* entre tipos no siempre son posibles. Por ejemplo, los tipos *boolean* e *int* no son compatibles
- Se permitirá una *conversión automática de tipos* si:
 - Los tipos implicados son compatibles
 - El tipo del destino es más grande que el tipo del origen

Conversiones de tipos (II)

❏ Conversión de ensanchamiento (*widening-conversion*)

- *byte --> short, int, long, float, double*
- *short --> int, long, float, double*
- *char --> int, long, float, double*
- *int --> long, float, double*
- *long --> float, double*
- *float --> double*



No se pierde
información
(son automáticas)

- En **operaciones aritméticas** donde intervengan datos de diferentes tipos, se **promueven** los tipos al del operando del tipo más grande:

```
int i = 2;  
float f;  
f = 11.0/i;           // f vale 2.5
```

- En el ejemplo anterior, el divisor (*int* 2) se promueve a *float* (2.0), que es el tipo del dividendo (11.0), antes de realizar la operación

Conversiones de tipos (III)

❏ Conversión de tipos incompatibles (*casting*)

- Aunque las conversiones automáticas son de ayuda, habrá situaciones en las que no se cumplan las condiciones de ensachamiento. Por ejemplo:

```
int i;  
float f = 10.3F;  
i = f;                                //--> error de compilación
```

generará un error de compilación al tratar de asignar el valor en punto flotante a la variable entera

```
...: error: incompatible types: possible lossy conversion from float to int  
      i = f;  
      ^
```

- En esos casos, tendremos que “*forzar*” el compilador para que realice la conversión solicitada. Para ello, invocaremos una operación de *cast*

Conversiones de tipos (IV)

- Para realizar un *cast* (o moldeado) de tipos, usaremos la sintaxis:

(tipo-requerido) expresión

donde *tipo-requerido*, encerrado entre paréntesis *()*, será el tipo de datos al que queremos convertir el resultado de evaluar la *expresión*

```
int i;  
float f = 10.3F;  
i = (int) f;           // cast a int del float f --> i vale 10 (truncamiento)
```

- Tenemos que tener presente que, estas conversiones por *estrechamiento*, pueden suponer la *pérdida de información*. Por ejemplo, la conversión de un valor en punto flotante a un tipo entero, *siempre* va a suponer el *truncamiento* de la parte decimal (en el ejemplo perderíamos el *0.3*)
- Podemos hacer *cast* sobre cualquier expresión que devuelva un valor:

```
(int) (x / y)
```

Fíjate en los paréntesis que rodean la *expresión x/y*. De no usarlos, el *cast* sólo se aplicaría sobre la variable *x*

Conversiones de tipos (V)

```
//: CastDemo.java

/**
 * Ejemplos de uso de <i>cast</i>
 * @author bowman
 * @version 1.0
 */

class CastDemo {
    public static void main(String[] args) {
        double x, y;
        byte b;
        int i;
        char ch;

        x = 10.0;
        y = 3.0;

        i = (int) (x / y); // truncamiento del resultado
        System.out.println("La división x/y es " + i);

        i = 100;
        b = (byte) i; // No hay pérdida (100<127)
        System.out.println("El valor de b es " + b);
```

```
        i = 257;
        b = (byte) i; // Un byte no puede almacenar 257!!!
        System.out.println("El valor de b es " + b);

        i = 88; // ASCII de 'X'
        ch = (char) i; // No hay pérdida (88<65535)
        System.out.println("El valor de ch es " + ch);
    }
}

/* Output:
La división x/y es 3
El valor de b es 100
El valor de b es 1
El valor de ch es X
*///:~
```


Conversiones de tipos (y VI)

❑ *Casting* y reglas de promoción de tipos

- Cada vez que se evalúa una expresión, se aplican las reglas de **promoción** de tipos de Java:
 1. Los valores *char*, *byte* y *short* se promueven a *int*.
 2. Si alguno de los operandos es *long*, la expresión se promueve a *long*
 3. Si alguno de los operandos es *float*, la expresión se promueve a *float*
 4. Si alguno de los operandos es *double*, se promueve a *double*
- Según lo anterior, habrá situaciones en las que nos veamos obligados a realizar un *cast* aún siendo los operandos y el destino del mismo tipo:

```
int i;  
byte b = 10;  
  
i = b * b;           // no se necesita hacer un cast  
b = (byte) (b * b); // se necesita hacer un cast!!!
```

API de Java (I)

- Cualquier lenguaje de alto nivel moderno, pone a disposición del programador una amplia biblioteca de utilidades y herramientas para facilitar su trabajo.
- Una parte esencial de estas herramientas son las **API** (*Application Programming Interface*). Estas son un conjunto de subrutinas, funciones y procedimientos (clases y métodos, en la programación orientada a objetos) que se pone a disposición de los programadores para la realización de tareas comunes y habituales. Por ejemplo, entre otros:
 - **Entrada/Salida** de datos desde consola, ficheros, red,...
 - **Cálculos matemáticos** complejos
 - **Gestión de errores** en tiempo de ejecución
 - Manejo de **estructuras avanzadas de datos**: listas, árboles,...
 - **Construcción de GUI** (*Graphical User Interface*)

API de Java (II)

- El número de clases que conforman la biblioteca de Java es enorme. Por ello, están organizadas en diferentes paquetes (y subpaquetes) por temática
- La **documentación oficial** del Java API está disponible tanto *online* como *offline*, y es el **primer y fundamental recurso** de un programador Java:
 - <https://docs.oracle.com/javase/8/docs/api/> (Java SE 8 API docs)otros:
 - <https://www.oracle.com/technetwork/java/api-141528.html> (Java API all-versions)
 - <https://www.oracle.com/technetwork/java/javase/documentation/jdk8-doc-downloads-2133158.html>
- Hacer uso de estas clases (y sus métodos) en Java es sencillo:
 - 1) **Importar** la clase, salvo las del paquete ***java.lang***, que se importan automáticamente. Por ejemplo: *java.lang.Math*, *java.lang.System*,...
 - 2) **Invocar** el método de clase si es *estático* (por ej., *Math.sqrt*). Si no, crearemos un nuevo objeto para llamar al método (p.e., *nextInt* de *Scanner*)

API de Java (y III)

❏ Ejemplo: uso de clases del API

```
//: DemoAPI.java
/**
 * Ejemplo de llamadas a métodos estáticos de clases del API de Java
 */

import javax.swing.JOptionPane; // Importamos la clase JOptionPane del paquete javax.swing
import java.time.LocalDateTime; // Importamos la clase LocalDateTime del paquete java.time

class DemoAPI {
    public static void main(String[] args) {
        /*
         * Llamada al método estático showMessageDialog() de la clase JOptionPane para
         * que muestre la hora actual, obtenida mediante el método estático now() de la
         * clase LocalDateTime
         */
        JOptionPane.showMessageDialog(null, "La hora actual es: " + LocalDateTime.now());
    }
}
```

En una única sentencia, gracias a las clases del API, hemos construido una ventana en la que se muestra la hora del sistema!!

Salida por consola (I)

- Una de las operaciones más básicas que proporciona el API es la que nos permite mostrar mensajes de texto en la consola del sistema
- Para ello, disponemos de la clase **System** que proporciona acceso a diferentes recursos del sistema. Uno de sus atributos, **out**, está “conectado” a la **salida estándar** del sistema (generalmente la pantalla o consola) y nos proporciona una serie de métodos para *escribir* en ella:
 - `System.out.print(mensaje)`, mostrará la cadena de texto *mensaje*.
 - `System.out.println(mensaje)`, mostrará la cadena de texto *mensaje* y añadirá un salto de línea al final
 - `System.out.printf(formato, lista_valores)`, permite realizar salidas formateadas. *formato* será una cadena de texto que incluirá **campos de reemplazo** que serán sustituidos por los valores de la *lista_valores*

Salida por consola (II)

- La consola emplea un **cursor** que determina la primera posición a partir de la cual se va a imprimir un nuevo mensaje
- A medida que se van imprimiendo nuevos caracteres, este cursor se va desplazando hasta quedar a continuación del último carácter impreso
- Vimos como en Java podemos hacer uso de una serie de **secuencias de escape**, que pueden ir incluídas en nuestros textos, y que alteran la forma en que se imprimen esos mensaje al modificar la posición del cursor
- Una de estas secuencias de escape, *nueva línea* (**\n**), se suele incluir habitualmente al final de los mensajes, para que la nueva impresión comience en la primera posición de la siguiente línea
- Por esta razón, la clase *System* de Java incluye ambos métodos, **print()** y **println()**, el segundo de los cuales incluye el salto de línea final

Salida por consola (III)

❑ Componiendo mensajes...

- El argumento *mensaje* de cualquiera de los métodos de impresión será la cadena de caracteres a imprimir. Java tratará dicha cadena como un objeto de tipo *String*.
- Este objeto *String* podrá crearse a partir de literales de tipo cadena de caracteres y variables, concatenados mediante el operador (+). Con independencia de su tipo, Java obtendrá una representación “*textual*” de los valores de la variables para poder imprimirlas. Por ejemplo:

```
int edad = 9;  
System.out.println("Tengo " + edad + " años");
```

Java interpreta las cadenas "Tengo " y " años", encerradas **entre comillas**, como valores **literales**. Sin embargo, *edad* (**sin comillas**), se interpreta como una **variable**. Java convierte su valor en un texto y los concatena

Salida por consola (III)

❑ Mostrando texto con *printf*...

- El método `System.out.printf` (f significa *formato*) permite mostrar datos con un formato específico (número de decimales, caracteres de relleno,...)
- Veamos un ejemplo:

```
System.out.printf("Hola %s!%n", "Mundo");
```

La salida de la sentencia anterior sería:

```
Hola Mundo!
```

- El primer argumento del método *printf*, en nuestro caso "Hola %s!%n", es un *String* que puede incluir texto y *secuencias de formato* (o *campos de reemplazo*). Cada una de estas secuencias, será reemplazada en el texto impreso por un valor o comportamiento. Por ejemplo, *%s* se sustituye por la cadena "Mundo", mientras que *%n* provoca un salto de línea

Salida por consola (IV)

- Las *secuencias de formato* tienen la siguiente sintaxis:

% [flags] [ancho] [.precisión] carácter_de_conversión ;

carácter conversión	imprime
c	caracter individual
C	caracter individual (mayúsculas)
b	booleano
B	booleano (mayúsculas)
d	número entero
f	número en punto flotante
e	número en punto flotante (notación científica)
s	cadena de caracteres
S	cadena de caracteres (mayúsculas)
n	salto de línea
tM	fecha y hora (M modificador de formato)

- flags*, establece ciertos modificadores de salida, especialmente con números
- ancho*, espacio de caracteres mínimo que ocupará el campo
- .precisión*, número de dígitos decimales en números de punto flotante

Salida por consola (V)

❑ Algunos ejemplos...

```
boolean b = true;
char c = 'x';
String s = "Texto de ejemplo";    // Creación de un objeto String

System.out.printf("Valores boolean: %b, char: %c y string: %s%n", b, c, s);
System.out.printf("En mayúsculas: %B, %C y %S", b, c, s);
```

```
Valores boolean: true, char: x y String: Texto de ejemplo
En mayúsculas: TRUE, X y TEXTO DE EJEMPLO
```

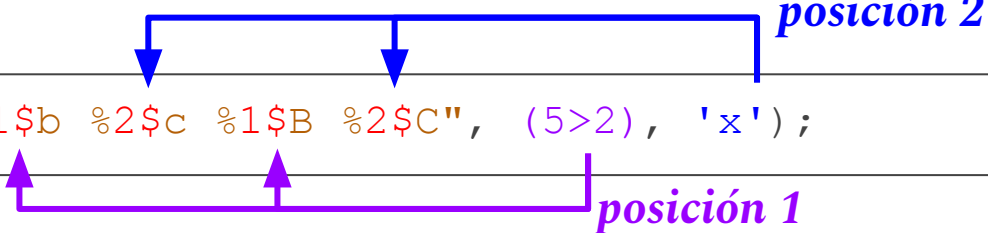
Fíjate como, por cada *campo de formato* (menos `%n` -> salto línea), se especifica un valor en la lista y se van emparejando en orden. Dichos valores pueden ser literales, variables o expresiones más complejas

```
System.out.printf("Valores boolean: %b, char: %c y string: %s%n", b, c, s);
```



Salida por consola (VI)

- Podemos también establecer directamente el **emparejamiento** entre un campo de formato y un valor de la lista. Para ello, indicaremos en el campo de formato la posición que ocupa en la lista el valor que nos interesa (empezando en 1). Por ejemplo:



```
System.out.printf("%1$b %2$c %1$B %2$C", (5>2), 'x');
```

La salida de la sentencia anterior será:

```
true x TRUE X
```

- También podemos especificar el ancho del campo en la salida, así como su alineación izquierda/derecha (derecha *por defecto*):

```
System.out.printf("'%1$10s' '%1$-10S'%n", "hola"); // campos de 10 caracteres
```

```
'      hola' 'hola      '
```

Salida por consola (VII)

- Emplearemos la secuencia de formato **%d** para la impresión de valores enteros (*byte, char, short, int, long*).
- Para números en punto flotante usaremos los códigos **%f** y **%e** (*notación científica*). Además, podremos especificar a cuántos dígitos decimales queremos **redondear** el valor
- En ambos casos, podremos indicar mediante el *flag* **,** si queremos que se muestre el separador de millares

```
System.out.printf("%d %f %n", 123, 5.128);      // un entero y un punto flotante
System.out.printf("%1$f %1$.2f %n", 123.456);  // sin y con redondeo a 2 decimales
System.out.printf("%,.2f %n", 12300.456);      // con redondeo y separador de millares
System.out.printf("%.2e %n", 123.456);         // con redondeo y notación científica
```

```
123 5,128000
123,456000 123,46
12.300,46
1,23e+02
```

Salida por consola (VIII)

- Por defecto, como separadores de decimales y millares, se emplearán los signos establecidos por la configuración regional del sistema
- Java proporciona una variante de *printf* en la que podemos pasarle como argumento la configuración regional que deseamos usar. Para ello, crearemos un objeto de la clase *java.util.Locale* con la configuración local deseada, o usaremos alguna de las que ya tiene predefinidas

```
import java.util.Locale;
. . .
System.out.printf("df: %,f %n", 12300.456); // config. reg. del sistema (defecto)
System.out.printf(Locale.US, "US: %,f %n", 12300.456); // aplica config. reg. US
// predefinida en Locale
Locale locES = new Locale("es", "ES"); // creamos config. regional para es_ES
System.out.printf(locES, "ES: %,f %n", 12300.456); // aplicamos config. es_ES
```

```
df: 12.300,456000
US: 12,300.456000
ES: 12.300,456000
```

Salida por consola (IX)

- Por último, también disponemos de secuencias de formato para valores de fechas y horas, permitiéndonos extraer campos individuales

carácter conversión	imprime
tH	hora
tM	minutos
tS	segundos
tp	am/pm
tz	zona horaria
tA	día de la semana (texto)
td	día del mes (numérico 2-dig)
tB	mes (texto)
tm	mes (numérico 2-dig)
tY	año (4-dig)
ty	año (2-dig)

Salida por consola (y XII)

❑ Ejemplo impresión de fecha/hora

```
//: PrintDateDemo.java
/**
 * Salida formateada de fecha/hora actual
 */

import java.time.ZonedDateTime;

class PrintDateDemo {
    public static void main(String[] args) {
        // Obtenemos fecha/hora actual con información de zona
        ZonedDateTime date = ZonedDateTime.now();

        System.out.printf("Hoy es %1$tA, %1$td de %1$tB de %1$tY %n", date);
        System.out.printf("Son las %1$tH:%1$tM:%1$tS [%1$tp] %n", date);
        System.out.printf("En la zona horaria %s [%tz] %n", date.getZone(), date);
    }
}

/* Output:
Hoy es lunes, 12 de agosto de 2019
Son las 00:23:01 [am] +0200
En la zona horaria Europe/Madrid [+0200]
*///:~
```

Entrada de datos (I)

- Aunque disponemos en Java de diferentes alternativas para leer la entrada de datos estándar, una de las formas más simples es a través de la clase *java.util.Scanner*, pues nos permite obtener directamente valores de tipos primitivos sin necesidad de realizar ninguna conversión
- Para poder utilizar cualquiera de los métodos de *Scanner*, deberemos primeramente crear un objeto de la misma. Algunos de sus métodos son:
 - *nextBoolean()*, lee un valor *boolean* introducido por el usuario
 - *nextByte()*, lee un valor de tipo *byte*
 - *nextDouble()*, lee un valor de tipo *double*
 - *nextFloat()*, lee un valor de tipo *float*
 - *nextInt()*, lee un valor de tipo *int*
 - *nextLine()*, lee toda la entrada como un *String*
 - *nextLong()*, lee un valor de tipo *long*
 - *nextShort()*, lee un valor de tipo *short*

Entrada de datos (II)

- La llamada a cualquiera de los métodos anteriores de *Scanner* es bloqueante, es decir, el programa se detendrá hasta que pulsemos la tecla *[Return]*, momento en el cual se devolverá el control al programa y se le “pasará” lo que haya introducido el usuario
- Si al llamar a alguno de esos métodos, el programa recibe un dato no convertible al tipo esperado, se producirá un error y el programa finalizará. Más adelante veremos cómo gestionar estos errores
- Al introducir valores en punto flotante, Java espera que lo hagamos utilizando la **configuración regional** del sistema. En nuestro caso, deberíamos emplear la **coma** en los datos de entrada para separ decimales (los literales en el programa usan el *punto* como separador decimal). Podemos modificar este comportamiento mediante el método *useLocale()* y establecer la configuración regional para la entrada que nos interese

Entrada de datos (y III)

❑ Ejemplo de uso de *java.util.Scanner*

```
//: ScannerDemo.java
import java.util.Scanner;
import java.util.Locale;

class ScannerDemo {
    public static void main(String[] args) {
        // Creamos el objeto Scanner
        Scanner entrada = new Scanner(System.in);

        // Cambiamos la configuración regional para usar '.' decimal
        // en lugar de ',' decimal (opcional)
        entrada.useLocale(Locale.US);

        System.out.println("Introduce tu nombre, edad y estatura (metros):");

        String nombre = entrada.nextLine();
        int edad = entrada.nextInt();
        double estatura = entrada.nextDouble(); // usamos separador decimal según config. regional

        System.out.println("Nombre: " + nombre);
        System.out.println("Edad: " + edad);
        System.out.println("Estatura (m): " + estatura);
    }
}
```