

# Ficha 1

## Fundamentos de Java y POO

---

### 1.] Breve referencia histórica sobre el lenguaje Java.

A comienzos de la década de 1990, el lenguaje C++ era el preferido por los desarrolladores para la creación de aplicaciones. Este lenguaje (cuya especificación fue definida por B. Stroustrup) reunía la potencia del C estándar con la programación orientada a objetos, y aunque no fue el primer lenguaje de objetos, fue el primero en ser usado para desarrollos profesionales.

Hacia 1991 un equipo de ingenieros de la firma *Sun Microsystems*, liderado por *James Gosling* y *Patrick Naughton*, comenzó a trabajar en el diseño de un lenguaje de programación que pudiera usarse para programar dispositivos electrodomésticos. Ese lenguaje debía tener la capacidad de adaptarse a distintos tipos de dispositivos y procesadores, siendo capaz de generar programas que pudieran correr en cualquier tipo de dispositivo que soportara al lenguaje. Originalmente ese lenguaje se llamó *\*7* (léase: *star seven*), pero ese nombre no fue muy bien recibido por la comunidad de programadores y se propuso cambiarlo al nombre *Oak*. Sin embargo, ya existía un lenguaje con ese nombre por lo que debió ser cambiado otra vez. Se dice que algunos miembros del equipo de trabajo de Gosling y Naughton estaban tomando un café discutiendo sobre el nuevo nombre del lenguaje, y alguien notó que estaban tomando "café de Java". Sin más, se propuso el nombre *Java* para el nuevo lenguaje, y eso explica también por qué el logotipo del lenguaje es una *taza de café humeante*.

Por los motivos ya indicados, el lenguaje *Java* se diseñó para poder generar programas que sean *portables* de una plataforma a otra. Vale decir: un programa desarrollado en un tipo de computadora con cierto tipo de sistema operativo, debería poder llevarse a otro tipo de computadora con un sistema operativo diferente, sin tener que cambiar el programa y sin tener que volver a compilar. La idea era buena y el lenguaje tenía elementos que lo hacían muy flexible y poderoso. Entre sus características, se decidió que sea *orientado a objetos*, y que su núcleo de instrucciones fuera idéntico al de *C++*, para facilitar la migración de programadores de ese lenguaje hacia *Java*. Sin embargo, el mercado no estaba listo para un lenguaje con esas características: aún no era masiva la cantidad de electrodomésticos programables disponibles, ni la necesidad de contar con un lenguaje avanzado para programarlos.

También en 1991, la *Internet* fue liberada para su uso comercial y eso provocó un cambio profundo en las estrategias de comercialización, transmisión de datos, comunicaciones y (por supuesto) en el diseño y desarrollo de sistemas informáticos. Hasta ese momento *Internet* era usada sólo en ámbitos académicos y gubernamentales de los Estados Unidos, y era controlada por la Fundación Nacional de Ciencia (*NSF: National Science Foundation*). La *Internet* ofrece numerosos servicios, entre los cuales se encuentra la *World Wide Web* (o simplemente, la *web*) a través de la cual todas las computadoras enlazadas a la red pueden

acceder a información gratuita sobre prácticamente cualquier tema que haya sido publicado, y también realizar operaciones publicitarias y comerciales a nivel mundial.

Con el dramático incremento en el uso de la Internet, pronto se vio que Java podría ser muy útil para programación de páginas web más atractivas que las que hasta ese momento se producían. El lenguaje HTML de las páginas web no es un lenguaje de programación sino un lenguaje de formateo de texto, y no se podía hacer con HTML que una página incluyera gráficas o animaciones sofisticadas, ni procesamiento de datos. Pero con Java podían crearse pequeños programas llamados applets, los cuales podían incluirse dentro de una página web y descargarse en la computadora del usuario. Como Java es portable, no importaba si ese applet se desarrollaba en un contexto Windows y luego se descargaba en una computadora con Linux (o viceversa): el applet se ejecutaría sin problemas si el navegador web usado soporta Java.

Mediante applets, un programador web adquiría mucha más flexibilidad en la programación y muchos más elementos gráficos que con HTML, y Java se relanzó como lenguaje adquiriendo de inmediato un éxito rotundo. La primera versión de Java fue canalizada mediante el navegador Netscape en 1996, pero desde entonces en adelante todos los navegadores web del mercado soportan Java.

Con el tiempo, surgieron otros lenguajes capaces de introducir programación compleja en una página web. Los lenguajes de script (como JavaScript o Visual Basic Script) así como Flash, son un claro ejemplo de ello. Pero Java no era (ni es) un simple lenguaje para programación de applets, aunque les deba su popularidad inicial a los applets. Java es un lenguaje multipropósito: permite desarrollar sistemas de amplia gama y hoy en día es muy utilizado para programación de aplicaciones de negocios (soluciones informáticas integrales para la gestión de las actividades de una empresa de cualquier magnitud).

## **2.] Versiones y distribuciones de Java.**

Desde su lanzamiento en 1995, Java evolucionó a través de muchas versiones. Cada una de ellas incorporaba soluciones a problemas de la anterior, mejoras, y también librerías o paquetes de clases que no necesariamente eran desarrolladas desde Sun Microsystems sino también por terceras organizaciones específicamente dedicadas a producir esas librerías (también llamadas APIs por Application Programming Interface o Interfaz de Programación de Aplicaciones) para Java. Cada versión de Java se identifica históricamente con números separados por puntos. Así, la primera versión de Java de 1996 para Netscape Navigator fue la versión Java 1.0 y hacia 1997 se lanzó la versión Java 1.1 que incluía una API (llamada Abstract Windowing Toolkit o AWT) para el desarrollo de interfaces de usuario basadas en ventanas, que no fue muy bien recibida por los desarrolladores debido a su débil estrategia para el control de eventos (entre otros problemas).

Sun Microsystems reaccionó rápido y en 1998 lanzó la versión Java 1.2 que redefinía la estrategia de control de eventos del Java 1.1, agregaba una nueva y muy potente API (conocida como Swing, que era el nombre código del equipo de desarrollo que la diseñó) para el desarrollo de interfaces de usuario y muchas otras características que instalaron a Java definitivamente como uno de los lenguajes líderes para el desarrollo de aplicaciones de negocio. A partir de esta versión comenzó a usarse el nombre general Java 2 para referirse a Java 1.2 y a las subsiguientes versiones que aparecerían después: Java 1.3 en el año 2000 y Java 1.4 en el año 2002. De hecho, a partir de entonces el conjunto de herramientas para

programación Java disponible desde la versión Java 2, se conoce también como J2SE (abreviatura de Java 2 Standard Edition).

En el año 2004 apareció Java 5.0 (note que la versión comercial ya no se designa como Java 1.5 sino directamente como Java 5.0, aunque internamente el nombre Java 1.5 se usa todavía. Quedó en la anécdota del marketing mundial el hecho que no haya existido una versión oficialmente llamada Java 3 ni una llamada Java 4...) Esta versión Java 5.0 es realmente una nueva versión del lenguaje y no una corrección o conjunto de mejoras a la versión anterior. La versión Java 5.0 incorpora elementos que cambian el núcleo del lenguaje, tales como clases genéricas, enumeraciones y una variante mejorada del ciclo for.

En diciembre de 2006 apareció Java 6 (sí, Java 6 y NO Java 6.0) imponiendo aún nuevos cambios y mejoras. A partir de esta versión, se elimina la designación J2SE y se cambia directamente por Java SE (lo cual era una necesidad obvia desde la versión Java 5.0). Entre los años 2009 y 2010, Sun fue adquirida por la empresa Oracle, y en 2011 apareció la versión Java SE 7.

La plataforma Java está disponible para ser descargada en forma gratuita desde el sitio específico para Java de Oracle (<http://www.java.com>). Dependiendo de las necesidades del programador y del tipo de aplicaciones que desee desarrollar, Java se presenta en tres modalidades o distribuciones, que según se dijo, a partir de 2006 se denominan así:

- **Java SE (antes J2SE):** Por Java Standard Edition, incluye las librerías esenciales de Java para el desarrollo de aplicaciones multipropósito, abarcando applets, conectividad, interfaces de usuario basadas en ventanas y almacenamiento en archivos externos entre otras características
- **Java EE (antes J2EE):** Por Java Enterprise Edition, incluye al JSE y aporta un servidor de aplicaciones y librerías de clases para el desarrollo de sistemas integrados para la web, abarcando el acceso y manejo de bases de datos.
- Se lanzaron otras versiones, tales como **Java FX** para la construcción de interfaces declarativas y **Java Card** para pequeñas aplicaciones para tarjetas inteligentes.

### 3.] El entorno de desarrollo de Java.

Existen distintos programas comerciales que permiten desarrollar código Java. Oracle, empresa que compró la empresa Sun (la creadora de Java), distribuye gratuitamente el Java Development Kit (JDK). Se trata de un conjunto de programas y librerías que permiten desarrollar, compilar y ejecutar programas en Java.

El JDK Incorpora además la posibilidad de ejecutar parcialmente el programa, deteniendo la ejecución en el punto deseado y estudiando en cada momento el valor de cada una de las variables (es el denominado **debugger**). Es tarea muy común de un programador realizar validaciones y pruebas del código que construye. El debugger permite realizar una prueba de escritorio automatizada del código, lo cual ayuda a la detección y corrección de errores. En el momento de escribir este trabajo las herramientas de desarrollo JDK, van por la versión 1.8. Estas herramientas se pueden descargar gratuitamente desde el sitio de Oracle: <http://www.oracle.com/technetwork/java>.

Los IDEs (Integrated Development Environment), tal y como su nombre indica, son entornos de desarrollo integrados. En un mismo programa es posible escribir el código Java, compilarlo y ejecutarlo sin tener que cambiar de aplicación. Algunos incluyen una

herramienta para realizar debug gráficamente, frente a la versión que incorpora el JDK basada en la utilización de una consola bastante difícil y pesada de utilizar. Estos entornos integrados permiten desarrollar las aplicaciones de forma mucho más rápida, incorporando en muchos casos librerías con componentes ya desarrollados, los cuales se incorporan al proyecto o programa. Estas herramientas brindan una interfaz gráfica para facilitar y agilizar el proceso de escritura de los programas. El entorno elegido para este módulo, por su relevancia en el mercado y su licencia gratuita se denomina Netbeans. El mismo se encuentra disponible para descargar de forma gratuita en [www.netbeans.org](http://www.netbeans.org).

El **compilador** de Java es otra de las herramientas de desarrollo incluidas en el JDK. Realiza un análisis de sintaxis del código escrito en los ficheros fuente de Java (con extensión \*.java). Si no encuentra errores en el código genera los ficheros compilados (con extensión \*.class). En otro caso muestra la línea o líneas erróneas. En el JDK de Sun/Oracle dicho compilador se llama *javac.exe* si es para el sistema operativo Windows.

Otra herramienta muy importante incluida en el JDK es la **Java Virtual Machine** (o Máquina Virtual Java) que es el programa que permite ejecutar a su vez un programa Java. Como se ha comentado anteriormente, la existencia de distintos tipos de procesadores y ordenadores llevó a los ingenieros de Sun a la conclusión de que era muy importante conseguir un software que no dependiera del tipo de procesador utilizado. Se planteó la necesidad de producir código capaz de ejecutarse en cualquier tipo de máquina: una vez compilado no debería ser necesaria ninguna modificación por el hecho de cambiar de procesador o de ejecutarlo en otra máquina. La clave consistió en desarrollar un código “neutro” el cual estuviera preparado para ser ejecutado sobre una “máquina hipotética o virtual”, denominada justamente **Java Virtual Machine (JVM)**. Es esta **JVM** (en la plataforma Windows, el programa *java.exe* incluido en el JDK) quien interpreta este código neutro convirtiéndolo a código particular de la CPU o chip utilizada. Se evita tener que realizar un programa diferente para cada CPU o plataforma.

La **JVM** es el intérprete de Java. Ejecuta los “bytecodes” (ficheros compilados con extensión \*.class) creados por el compilador de Java (*javac.exe*). Tiene numerosas opciones entre las que destaca la posibilidad de utilizar el denominado **JIT** (Just-In-Time Compiler), que puede mejorar entre 10 y 20 veces la velocidad de ejecución de un programa.

#### 4.] El lenguaje de programación Java.

Sea cual fuere la plataforma de Java en la que nos dispongamos a construir un producto de software, hay un elemento que es común a todas ellas y que se mantiene invariable a través de distintos tipo de productos de software y distintos mecanismos de implementación, publicación y consumo. Ese elemento es el lenguaje de programación Java. El lenguaje Java, que en el primer apartado de la presente Ficha se presentó como un derivado de C++ en sus orígenes, es hoy un lenguaje de programación muy maduro y con una amplia gama de herramientas para simplificar la escritura de código. Podemos resumir la características principales de Java diciendo que Java es un lenguaje:

- ✓ **De Alto Nivel:** es un lenguaje que está más cerca de ser legible por el ser humano que de las instrucciones del procesador y por otro lado contiene una amplia gama de librerías y código ya desarrollado que permite que el programador se pueda centrar en las funcionalidades a programar e independizarse de las cuestiones de infraestructura física de las computadoras.

- ✓ **Orientado a objetos:** Al contrario de otros lenguajes como C++, Java no es un lenguaje modificado para poder trabajar con objetos, sino que es un lenguaje creado originalmente para trabajar con objetos. De hecho, todo lo que hay en Java son objetos.
- ✓ **Independiente de la plataforma:** Debido a que existen máquinas virtuales para diversas plataformas de hardware, el mismo código Java puede funcionar prácticamente en cualquier dispositivo para el que exista una JVM.
- ✓ **Compilado e Interpretado (la JVM es más que un intérprete, pero mantiene el mismo concepto):** La principal característica de Java es la de ser un lenguaje compilado e interpretado. De este modo se consigue la independencia de la máquina: el código compilado se ejecuta en máquinas virtuales que sí son dependientes de la plataforma. Para cada sistema operativo distintos, ya sea Microsoft Windows, Linux, OS X, existe una máquina virtual específica que permite que el mismo programa Java pueda funcionar sin necesidad de ser recompilado.
- ✓ **Robusto:** Su diseño contempla el manejo de errores a través del mecanismo de Excepciones y fuerza al desarrollador a considerar casos de mal funcionamiento para reaccionar ante las fallas.
- ✓ **Gestiona la memoria automáticamente (Memory Safe):** La máquina virtual de Java gestiona la memoria dinámicamente como veremos más adelante. Existe un recolector de basura que se encarga de liberar la memoria ocupada por los objetos que ya no están siendo utilizados.
- ✓ **Multihilos (multithreading):** Soporta la creación de partes de código que podrán ser ejecutadas de forma paralela y comunicarse entre sí.
- ✓ **Cliente-servidor:** Java permite la creación de aplicaciones que pueden funcionar tanto como clientes como servidores. Además, provee bibliotecas que permiten la comunicación, el consumo y el envío de datos entre los clientes y servidores.
- ✓ **Con mecanismos de seguridad incorporados:** Java posee mecanismos para garantizar la seguridad durante la ejecución comprobando, antes de ejecutar código, que este no viola ninguna restricción de seguridad del sistema donde se va a ejecutar. Además, posee un gestor de seguridad con el que puede restringir el acceso a los recursos del sistema.
- ✓ **Con herramientas de documentación incorporadas:** Como veremos más adelante, Java contempla la creación automática de documentación asociada al código mediante la herramienta Javadoc.

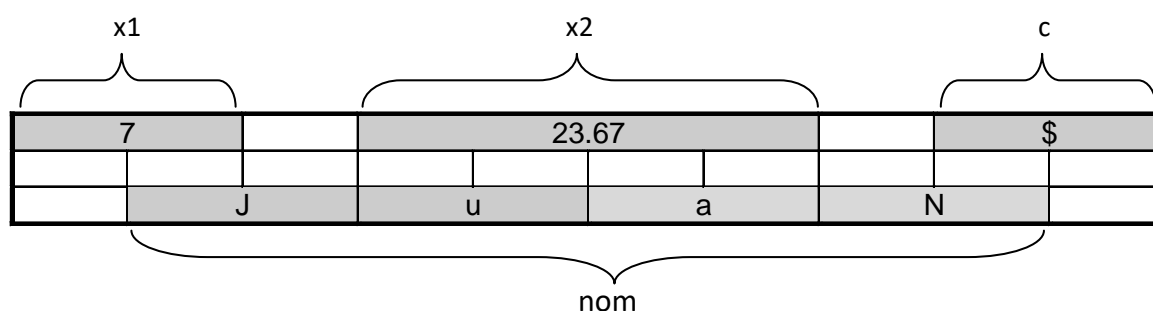
## 5.] Elementos básicos de programación en Java: tipos, variables y asignaciones.

Una computadora acciona o responde si posee un programa cargado al que pueda interpretar y ejecutar instrucción por instrucción. Esto implica que la computadora de alguna forma es capaz de representar y retener información dentro de sí. En ese sentido, se llama **memoria** al dispositivo interno del computador en el cual se almacena esa información. Básicamente, la memoria puede considerarse como una gran tabla compuesta por celdas individuales llamadas **bytes**, de forma que cada byte puede representar información usando el *sistema de numeración binario* basado en estados eléctricos.

Cuando se desea ejecutar un programa, el mismo debe cargarse en la memoria del computador. Un programa, entonces, ocupará normalmente varios cientos o miles de bytes. Sin embargo, con sólo cargar el programa no basta: deben cargarse también en la memoria los *datos* que el programa necesitará procesar. A partir de aquí, el programa puede ser ejecutado e irá generando ciertos *resultados* que de igual modo serán almacenados en la memoria. Estos valores (los datos y los resultados) ocuparán en memoria un cierto número de bytes, que depende del **tipo de valor** del que se trate (por ejemplo, en general en distintos lenguajes de programación un valor de *tipo entero* ocupa entre uno y ocho bytes en memoria, un valor de *tipo real* o de *coma flotante*, con punto y parte decimal, ocupa entre cuatro y ocho bytes, y un *caracter* ocupa uno o dos bytes dependiendo del estándar de representación empleado).

En todo lenguaje de programación, el acceso a esos datos y resultados ubicados en memoria se logra a través de ciertos elementos llamados **variables**. Una variable es un grupo de bytes asociado a un nombre o identificador, pero de tal forma que a través de dicho nombre se puede usar o modificar el contenido de los bytes asociados a esa variable. En el siguiente gráfico, se muestra un esquema conceptual referido a cómo podría verse la memoria de un computador, con cuatro variables (*x1*, *x2*, *c*, *nom*) alojadas en ella:

Figura 1: Memoria de un computador (esquema)



En el gráfico (que es sólo una representación libre de la idea de memoria como tabla formada por casilleros), se supone que la variable *x1* contiene el valor 7, y del mismo modo el resto de las variables del cuadro contiene algún valor. El nombre de cada variable es elegido por el programador, de acuerdo a ciertas reglas que se verán más adelante.

Para cambiar el valor de una variable se usa la **instrucción de asignación**, que consiste en escribir el nombre de la variable, seguido del signo igual y luego el valor que se quiere asignar. El signo igual (=) se designa como **operador de asignación**. Notar que en el lenguaje Java toda instrucción finaliza con punto y coma (;). Ejemplo:

```
x1 = 7;
```

Es importante destacar que en la memoria de un computador pueden representarse valores no numéricos. Es perfectamente posible usar variables para almacenar valores de caracteres. En el gráfico anterior, la variable *c* contiene el caracter simple '\$', y la variable *nom* contiene la cadena de caracteres "Juan" (una cadena de caracteres es una sucesión de caracteres simples). La clase de valores que una variable puede contener, se llama *tipo de dato* de esa variable (en general, una variable sólo puede contener valores de un tipo único). Los lenguajes de programación proveen varios tipos de datos estándar para manejar variables. Así, en el lenguaje Java existen ocho tipos de datos básicos, llamados *tipos simples*

o *tipos primitivos*, que permiten manejar números (enteros o con parte decimal), caracteres simples y valores lógicos (*true* y *false*). El siguiente cuadro muestra los nombres y algunas otras características de esos tipos:

Figura2: Tabla de Tipos de Datos Primitivos en Java

Tipo primitivo	Descripción	Bytes por cada variable	Rango
<b>boolean</b>	valores lógicos	1	false, true
<b>char</b>	caracteres simples	2	caracteres Unicode
<b>byte</b>	números enteros	1	[ -128, 127 ]
<b>short</b>	números enteros	2	[ -32768, 32767 ]
<b>int</b>	números enteros	4	[ -2,147,483,648, 2,147,483,647 ]
<b>long</b>	números enteros	8	[ -9,223,372,036,854,775,808, 9,223,372,036,854,775,807 ]
<b>float</b>	números decimales	4	hasta 6 o 7 decimales
<b>double</b>	números decimales	8	hasta 14 o 15 decimales

Notar, por otra parte, que una variable ocupará más o menos bytes en memoria de acuerdo al tipo de dato al que pertenece. Las variables *short* ocupan dos bytes cada una en memoria, mientras que las *float* ocupan cuatro cada una. Una variable *char* de caracteres simple, ocupa dos bytes en Java. En el gráfico de celdas de memoria mostrado en la *¡Error! No se encuentra el origen de la referencia.*, se supuso que la variable *x1* es de tipo *short*, y que *x2* es *float*. También se supuso que *c* es de tipo *char*, y que *nom* es una *cadena de caracteres*. En general, una cadena de caracteres es una secuencia de caracteres simples que ocupa tantos bytes como caracteres tenga la cadena. Por eso en el gráfico de la figura 1 la variable *nom* se dibujó ocupando ocho casillas. En Java, las cadenas de caracteres no son valores simples o primitivos, y por ello no figuran en el cuadro de tipos dado en la página anterior. Como veremos, las cadenas de caracteres son objetos que pertenecen a una clase particular llamada *String*, sobre la cual volveremos oportunamente.

En el lenguaje Java no se puede utilizar una variable en un programa si la misma no fue convenientemente **declarada** en ese programa. Declarar una variable significa:

- ✓ Indicar el *nombre* o *identificador* de la misma.
- ✓ Indicar a qué *tipo de dato* pertenece la variable.

En la declaración, se escribe primero el tipo de la variable y luego el nombre o identificador de la misma. Si hay varias variables del mismo tipo, se pueden declarar juntas separando sus nombres con comas (,) La declaración termina con el consabido punto y coma. Ejemplos:

```
int    x1;
float  x2, x3;
```

```
char c;  
String nom;
```

En el esquema anterior la variable *c* fue declarada como variable de caracter simple (es decir, admite que le sea asignado un único caracter), y la variable *nom* se declaró como cadena de caracteres (en el caso del ejemplo, admite cadenas de tantos caracteres como el programador necesite).

Una vez declaradas, las variables pueden comenzar a usarse para hacer asignaciones u otras operaciones. Solo debe observarse que si se quiere asignar un caracter a una variable de caracteres simple, el caracter a asignar debe encerrarse entre comillas simples (' '). Por ejemplo, la asignación del caracter '\$' a la variable *c*, se haría así:

```
c = '$';
```

Si se quiere asignar una cadena a una variable de tipo cadena de caracteres, entonces la cadena a asignar debe encerrarse entre comillas dobles (" "):

```
nom = "Juan";
```

Los siguientes ejemplos son válidos para ilustrar el uso de las declaraciones de variables y las posibilidades de la instrucción de asignación (los números a la izquierda de cada línea, se usan para poder hacer referencia a cada ejemplo, y no forman parte de un programa en Java):

```
1. char c, a;  
2. String nom1, nom2;  
3. c = '+';  
4. a = c;  
5. a = 'c';  
6. n1 = "José";  
7. n2 = n1;  
8. n2 = "n1";
```

En la línea 3.), se asigna el caracter '+' a la variable *c*, y luego en la línea 4.), se asigna la variable *c* en la variable *a*. Esto significa que la variable *a* queda valiendo lo mismo que la variable *c* (es decir, ambas contienen ahora el signo '+'). Observar la diferencia con lo hecho en la línea 5.), en donde se asigna a la variable *a* el caracter 'c' (y no el contenido de la variable *c*). En la línea 6.) se asigna la cadena "José" a la variable *n1*, y en la línea 7.), se asigna la variable *n1* en la variable *n2* (de nuevo, ambas quedan valiendo lo mismo: la cadena "José"). En la línea 8.), se está asignando literalmente la cadena "n1" a la variable *n2*, por lo que las líneas 7.) y 8.), no son equivalentes.

Un hecho importante derivado de la instrucción de asignación, es que cuando se asigna un valor a una variable, esta asume el nuevo valor, y *pierde cualquier valor anterior* que hubiera contenido. Por ejemplo, considérese el siguiente segmento de programa:

```
int a;  
a = 2;  
a = 4;
```

En este caso, se comienza asignando el valor 2 a la variable *a*, pero inmediatamente se asigna el 4 en la misma variable. El valor final de *a* luego de estas tres instrucciones, es 4, y el 2 originalmente asignado se pierde. Lo mismo ocurre con cualquier otra variable, sea cual sea el tipo de la misma.



Para resumir, recordemos que el nombre de una variable es elegido por el programador, pero para ello deben seguirse ciertas reglas, que indicamos a continuación:

- El nombre o identificador de una variable en Java, solo puede contener letras (mayúsculas y/o minúsculas, o también dígitos (0 al 9), o también el guión bajo ( `_` ) (también llamado guión de subrayado).
- El nombre de una variable *no debe comenzar* con un dígito.
- El nombre de una variable no puede ser una palabra reservada del lenguaje Java.
- Java es *case sensitive*, es decir, hace diferencia entre minúsculas y mayúsculas, por lo que toma como diferentes a dos nombres de variables que no sean exactamente iguales. El identificador *suel*do no es igual al identificador *Suel*do y Java tomará a ambos como dos variables *diferentes*.

Por ejemplo, los siguientes identificadores de variables son válidos:

```
n1
nombre_2
sueldo_anterior
x123
xint
año
número
```

Pero los siguientes son incorrectos, por los motivos que se indican:

3xy	Comienza con un número.
dir ant	Hay un blanco en el medio (no es un caracter válido).
nombre-2	Usa el guión alto y no el bajo.
int	Es palabra reservada.

## 6.] Operadores aritméticos en Java.

Hemos visto que se puede asignar en una variable un valor, también se puede asignar el *resultado de una expresión*. Una *expresión* es una fórmula en la cual se usan *operadores* (como suma o resta) sobre diversas variables y constantes (que reciben el nombre de *operandos* de la expresión). Si el resultado de la expresión es un número, entonces la expresión se dice *expresión aritmética*. Los siguientes es un ejemplo de una *expresión aritmética* en Java:

```
int suma, num1 = 10, num2 = 7;
suma = num1 + num2;
```

En la última línea se está asignando en la variable *suma* el resultado de la expresión *num1 + num2*; y obviamente la variable *suma* quedará valiendo 17. Note que en una asignación *primero* se evalúa cualquier expresión que se encuentre a la derecha del signo `=`, y *luego* se asigna el resultado obtenido en la variable que esté a la izquierda del signo `=`. La siguiente tabla muestra los *principales operadores aritméticos* del lenguaje Java (volveremos más adelante con un estudio más detallado sobre la aplicación de estos operadores):

Operador	Significado	Ejemplo de uso
+	suma	$a = b + c;$
-	resta	$a = b - c;$
*	producto	$a = b * c;$
/	división	$a = b / c;$
%	resto de una división	$a = b \% c;$

En Java los distintos operadores aritméticos *actúan de acuerdo al tipo de las variables o constantes sobre las que operan*. Así, si se usa el operador suma (+) para sumar dos variables *int*, el resultado será un valor *int*, y lo mismo ocurrirá con los demás.

Aunque lo anterior es lo lógico, no siempre el resultado obtenido será el que hubiésemos esperado: si se usa el operador *división (/)* y los dos números que se dividen son de tipo *int*, entonces el operador calculará la llamada *división entera* (o *cociente entero*) entre ambos, lo cual significa que la parte decimal del resultado será truncada. Veamos el siguiente ejemplo:

```
int a, b, c;
a = 5;
b = 2;
c = a / b;
```

El valor de *c*, será de 2, y no de 2.5, ya que la división se calculó en forma entera al ser de tipo *int* las variables *a* y *b*.

En ese sentido, el operador *resto (%)* es muy útil porque permite calcular el resto de una división entera (y esto a su vez es muy valioso en casos en que se quiere aplicar conceptos de divisibilidad): la expresión  $r = x \% y;$  calcula el resto de dividir en forma entera a *x* por *y*, y asigna ese resto en la variable *r*. En el caso citado antes, si los valores ingresados fueran  $x = 11$  y  $y = 2$ , el resto calculado sería 1.

Note que para usar y aplicar el *operador resto* NO ES NECESARIO usar antes el operador división. Ambos operadores se pueden aplicar sin tener que usar el otro. Si usted sólo desea calcular el resto de una división, simplemente use el operador % para obtenerlo y no use el operador división.

## 7.] Estructura de un Programa Completo Mínimo en Java.

Un programa Java típico (sin importar lo pequeño o simple que sea) debe incluir un proceso especial, llamado **método main**. Ese método es el primero que se ejecuta cuando se pide ejecutar el programa, y desde allí se lanzan otros procesos que el programa pudiera necesitar.

Los procesos o métodos, se declaran dentro de alguna clase (descriptor que veremos más adelante). Al desarrollar un programa en Java, el programador creará por lo general varias clases que luego trabajarán juntas. Es común (aunque no obligatorio) declarar una clase cuyo único objetivo sea contener al método *main*. En el contexto, seguiremos la convención de designar a esa clase con el nombre *Principal*. Pero debe tenerse en cuenta un detalle importante: el archivo de código fuente donde se almacene una clase, debe llevar el mismo nombre que la clase (y obviamente con extensión *.java*). Si la clase se llama *Principal*, el archivo fuente debe llamarse *Principal.java*.

La **cabecera** (es decir, la primera línea) del método *main* debe escribirse como se muestra:

```
public static void main (String args[])
```

La única variante admitida es el nombre de la variable *args*, que puede llamarse de cualquier forma en lugar de "args", y la ubicación de los corchetes junto a esa variable: los mismos podrían ir antes el nombre de la variable. La siguiente sería una declaración válida también:

```
public static void main (String [] x)
```

Las acciones llevadas a cabo por el método se encierran entre dos llaves. A modo de ejemplo simple, mostramos una clase que sólo incluye un método *main* que muestra un mensaje en la consola estándar, usando la instrucción *System.out.print()* que veremos en la sección siguiente.

```
1 public class Principal {
2
3     public static void main(String[] args) {
4
5         System.out.print("Primer Ejemplo");
6     }
7 }
8
```

Cuando escribimos código en general es útil realizar comentarios explicativos. Los comentarios no tienen efecto como instrucciones para el lenguaje, simplemente sirven para que cuando un programador lea el código pueda comprender mejor lo que lee.

En Java se pueden usar dos tipos de comentarios:

- *Comentario en una línea o al final de una línea*: se introduce con el símbolo //
- *Comentario multilínea*: se abre con el símbolo /\* y se cierra con el símbolo \*/

El **comentario en una línea** será ignorado por el compilador de Java. Lo que sea que se escriba a la derecha del comentario de línea, será tomado como un texto fuera del programa por Java, y su efecto dura hasta el final de la línea.

```
//Asignación inicial de un valor int a la variable n
n = 20
```

El comentario multilínea será ignorado por el compilador de Java. Lo que sea que se escriba dentro de los símbolos /\* y \*/, será tomado como un texto fuera del programa por Java, y su efecto dura hasta el final del comentario. Este caso se usa cuando se desea introducir un comentario que ocupe más de una línea, como en el ejemplo que sigue:

```
/*
    Un comentario de párrafo...
    Asignación inicial de un valor int
    a la variable n
*/
n = 20
```

Todo el bloque escrito en rojo en el ejemplo anterior, será ignorado por Java como si simplemente no existiese. Existe otro tipo de comentario, que se usa con el objetivo de documentar un sistema y no sólo para intercalar texto libre en un programa, en la que Java tiene un sistema normalizado de comentarios llamado javadoc, que estudiaremos más adelante.

Usando el ejemplo anterior, le agregamos comentarios:

```
1  /*
2   * Este es el primer ejemplo
3   */
4  public class Principal {
5
6      public static void main(String[] args) {
7          // muestro un mensaje
8          System.out.print("Primer Ejemplo");
9      } //fin del main
10
11 }
```

## 8.] Visualización por Pantalla (salida por Consola Estándar).

Al ejecutar un programa, lo normal es que antes de finalizar el mismo muestre por pantalla los resultados obtenidos. En el lenguaje Java la instrucción más básica para hacer eso es `System.out.print()`. Esta instrucción permite mostrar en pantalla tanto el contenido de una variable como también mensajes formados por cadenas de caracteres, lo cual es útil para lograr salidas de pantalla “amigables” para quien use nuestros programas. La forma de usar la instrucción se muestra en los siguientes ejemplos:

```
1  public class Principal
2  {
3      public static void main ( String args [])
4      {
5          //declaración de variables
6          int a , b;
7          //asignación de variables
8          a = 3;
9          //operación, le suma 1 al valor de a
10         b = a + 1;
11         //muestra el resultado
12         System.out.print( b );
13     } //fin del main
14 }
```

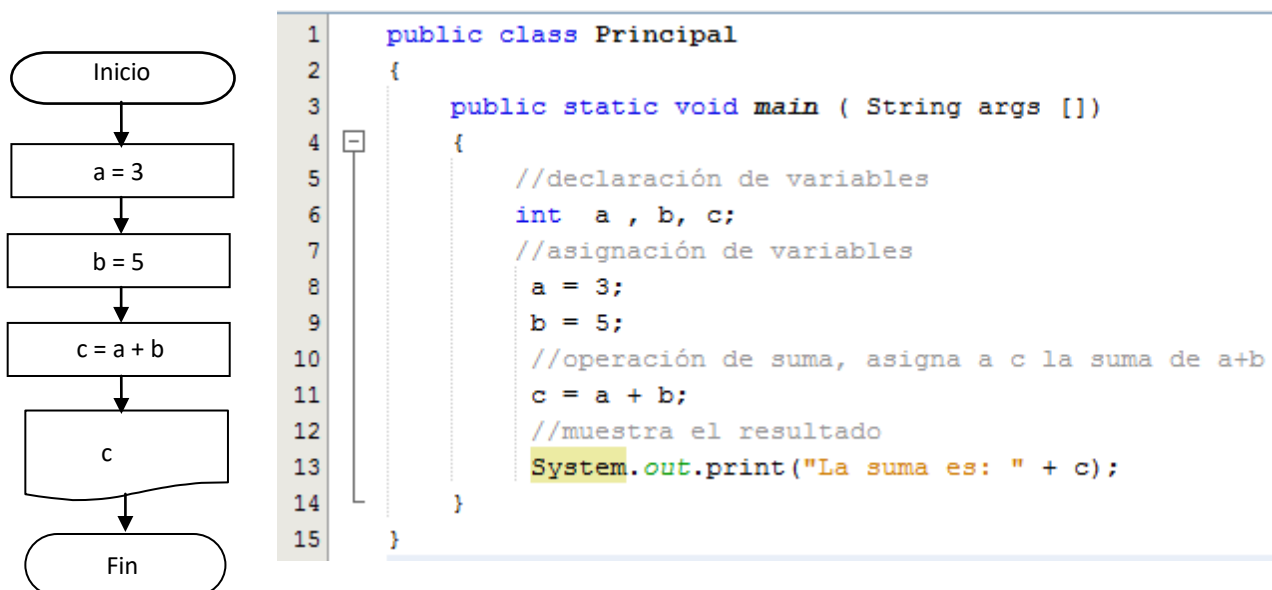
Aquí, la instrucción de la línea 12), muestra en pantalla el valor contenido en la variable `b`, o sea, el número 4. Sin embargo, una salida más elegante sería acompañar al valor en pantalla con mensaje aclaratorio:

```
System.out.print( "El resultado es: " + b );
```

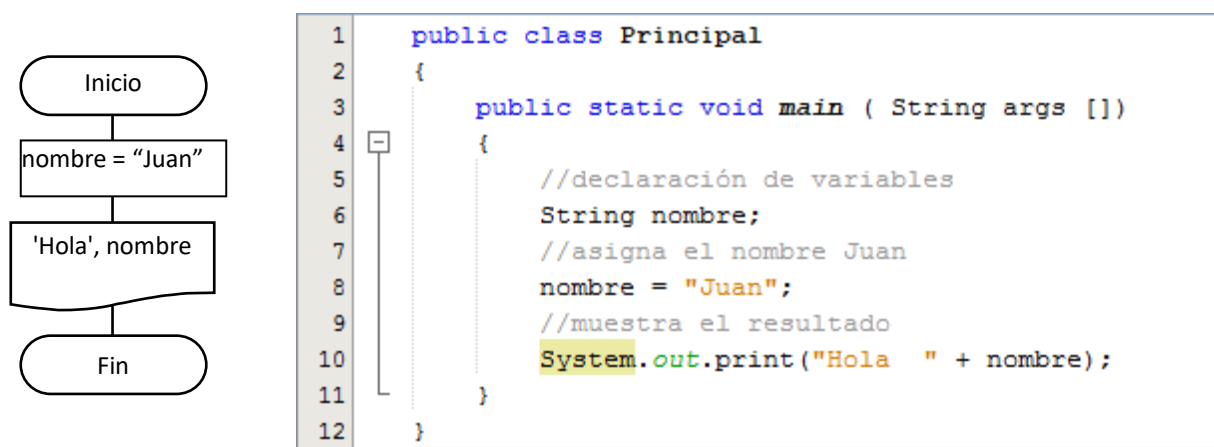
Observar que ahora no sólo aparecerá el valor contenido en `b`, sino también la cadena “*El resultado es:* ”, precediendo al valor. Notar también que para mostrar el valor de una variable, el nombre de la misma no lleva comillas, pues en ese caso se tomaría al nombre en forma literal. La instrucción que sigue, muestra literalmente la letra ‘b’ en pantalla:

```
System.out.print("b");
```

**Ejemplo 1.)** Un programa que obtiene y muestra la suma de dos números:



**Ejemplo 2.)** Un programa que asigna el nombre de una persona y le muestra un saludo:



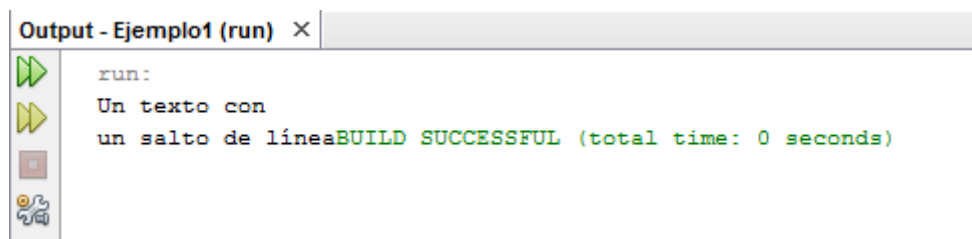
También se encuentra disponible la instrucción `System.out.println()`, que muestra el texto y luego deja un salto de línea (como si presionara enter) mientras que `print()` no lo hace.

Si se necesita ingresar saltos de línea entre los textos se puede utilizar la secuencia de escape `\n`, que introducirá uno nuevo donde se lo coloque, por ejemplo:

```

1  public class Principal
2  {
3      public static void main ( String args [])
4      {
5          System.out.print("Un texto con\nun salto de línea");
6      }
7  }
  
```

Lo cual nos generará lo siguiente por pantalla:



```

Output - Ejemplo1 (run) x
run:
Un texto con
un salto de línea
BUILD SUCCESSFUL (total time: 0 seconds)
  
```

Java proporciona las **secuencias de escape**, una combinación de la barra invertida \ y un carácter que nos permite insertar caracteres especiales dentro de un texto:

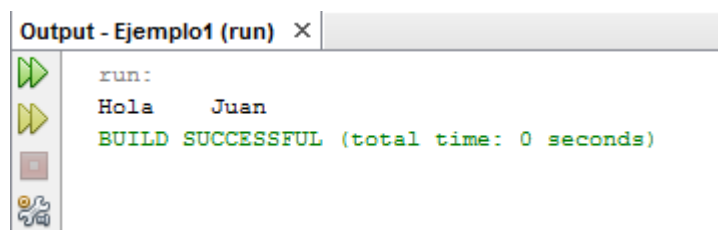
Secuencia de Escape	Descripción
\n	Salto de línea
\t	Tabulador
\\	Diagonal Inversa \
\"	Comillas Dobles
\'	Comilla Simple
\r	Retorno de Carro (Solo en modo Administrador)
\b	Borrado a la Izquierda (Solo en modo Administrador)

Otros ejemplos: Para tabular se utiliza \t.

```

1  public class Principal
2  {
3      public static void main ( String args [])
4      {
5          System.out.println("Hola \tJuan");
6      }
7  }
  
```

Resultado:



```

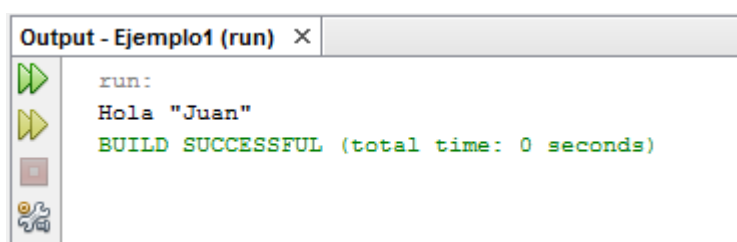
Output - Ejemplo1 (run) x
run:
Hola      Juan
BUILD SUCCESSFUL (total time: 0 seconds)
  
```

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

```

1  public class Principal
2  {
3      public static void main ( String args [])
4      {
5          System.out.println("Hola \"Juan\"");
6      }
7  }
  
```

Resultado:



```
run:
Hola "Juan"
BUILD SUCCESSFUL (total time: 0 seconds)
```

## 9.] Entrada de Datos por teclado en Java.

Esta operación permite mayor generalidad en la carga de datos de un programa. En el ejemplo anterior, se analizó un programa que permitía sumar dos números, en base al esquema de asignación directa de valores. Podemos darnos cuenta rápidamente que un programa así planteado es muy poco útil, pues indefectiblemente el resultado mostrado en pantalla será 8... El programa tiene muy poca flexibilidad debido a que el valor inicial de la variable *a* es siempre 5 y el de *b* es siempre 3. Lo ideal sería que mientras el programa se ejecuta, pueda pedir que el usuario ingrese por teclado un valor para la variable *a*, luego otro para *b*, y que luego se haga la suma (en forma similar a como permite hacerlo una calculadora...)

Java a partir de la versión 5 provee una clase que está preparada para capturar una entrada por teclado en la consola estándar, de forma simple y sin necesidad de mayores conceptos previos.

Además, cuando se llega a desarrollar programas basados en ventanas Java provee un amplio conjunto de métodos para hacer tales cargas, pero en un programa sencillo orientado a la consola estándar (y por lo tanto que no use ventanas de usuario de alto nivel), Java provee la herramienta que analizaremos a continuación como alternativa básica.

Java a partir de la versión 5 incluye la clase **Scanner**, la cual entre otras varias y amplias funcionalidades que implementa agrega también la posibilidad de realizar de manera simple y concreta la lectura de valores numéricos, caracteres o cadenas de caracteres desde teclado a través de la consola estándar. Para utilizar la clase *Scanner* lo único especial que hay que agregar es la siguiente línea de código que debe quedar antes de la declaración de la clase en el archivo de código.

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

Los métodos de la clase *Scanner* que nos interesan en este punto se presentan a continuación y cabe aclarar que la clase *Scanner* tiene varios métodos más, que no nos interesa analizar por ahora:

Tipo de retorno	Método	descripción
<i>String</i>	<code>nextLine()</code>	Retorna la próxima carga de cadena de caracteres.
<i>boolean</i>	<code>nextBoolean()</code>	Retorna la próxima carga como un valor booleano.
<i>byte</i>	<code>nextByte()</code>	Retorna la próxima carga como un valor de tipo byte.

<i>double</i>	nextDouble()	Retorna la próxima carga como un valor de tipo double.
<i>float</i>	nextFloat()	Retorna la próxima carga como un valor de tipo float.
<i>int</i>	nextInt()	Retorna la próxima carga como un valor de tipo int.
<i>long</i>	nextLong()	Retorna la próxima carga como un valor de tipo long.
<i>short</i>	nextShort()	Retorna la próxima carga como un valor de tipo short.

Para utilizar estos métodos es necesario realizar algunas tareas previas, en primer lugar, debemos crear el escáner, es decir debemos crear el objeto que va a escanear la entrada estándar para luego utilizar los métodos, ese bloque de código quedaría como sigue:

```

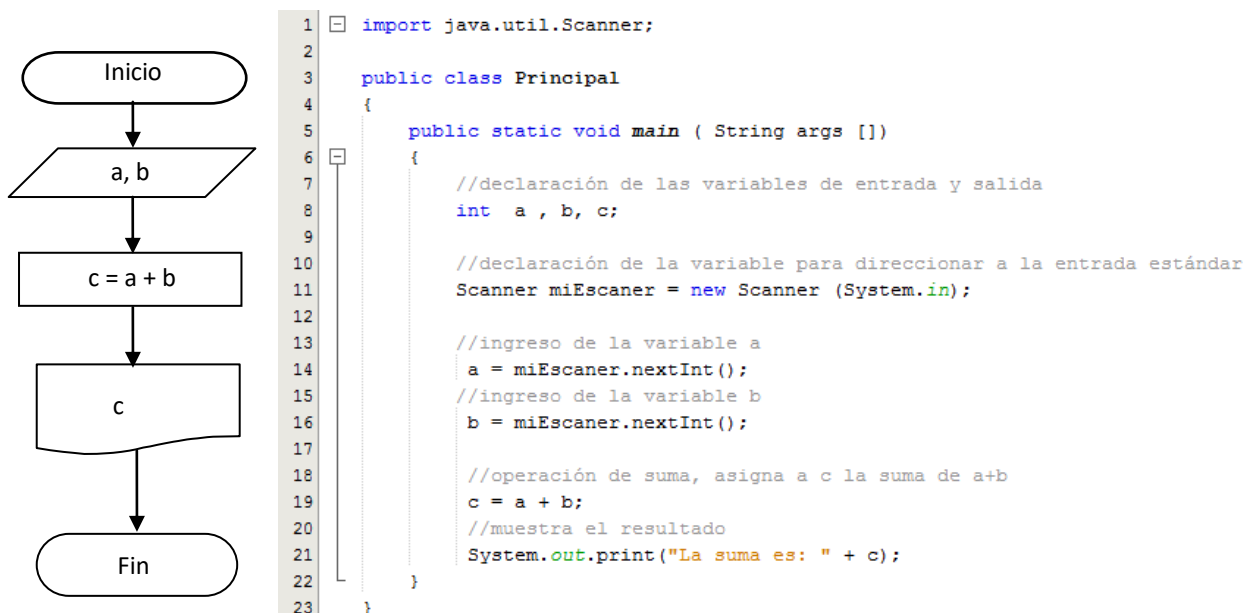
1  import java.util.Scanner;
2
3  public class Principal
4  {
5      public static void main ( String args [])
6      {
7          //declarar la variable scanner que direcciona a la entrada estándar
8          Scanner miEscaner = new Scanner (System.in);
9
10         //para leer un número entero
11         int i = miEscaner.nextInt();
12
13         //para leer un número double
14         double d = miEscaner.nextDouble();
15
16         //para leer un String
17         String s = miEscaner.nextLine();
18     }
19 }
```

Cada uno de esos métodos se invoca escribiendo primero el nombre del objeto de la clase Scanner que creamos en la primera línea (en el ejemplo referenciado por <miEscaner>) seguido de un punto, y luego el nombre del método. Cada método, al ejecutarse, provoca que el programa entre en modo de espera transitoria, sin ejecutar ninguna otra instrucción hasta que alguien ingrese por teclado un valor y se presione la tecla <Enter>. El valor así ingresado, se asigna en la variable indicada en la instrucción a la izquierda del signo igual, y luego prosigue normalmente el programa. En el ejemplo anterior, el segmento de programa mostrado provoca la carga por teclado de un número entero en la variable i, luego un valor de coma flotante en la variable d, y finalmente una cadena de caracteres en la variable s.

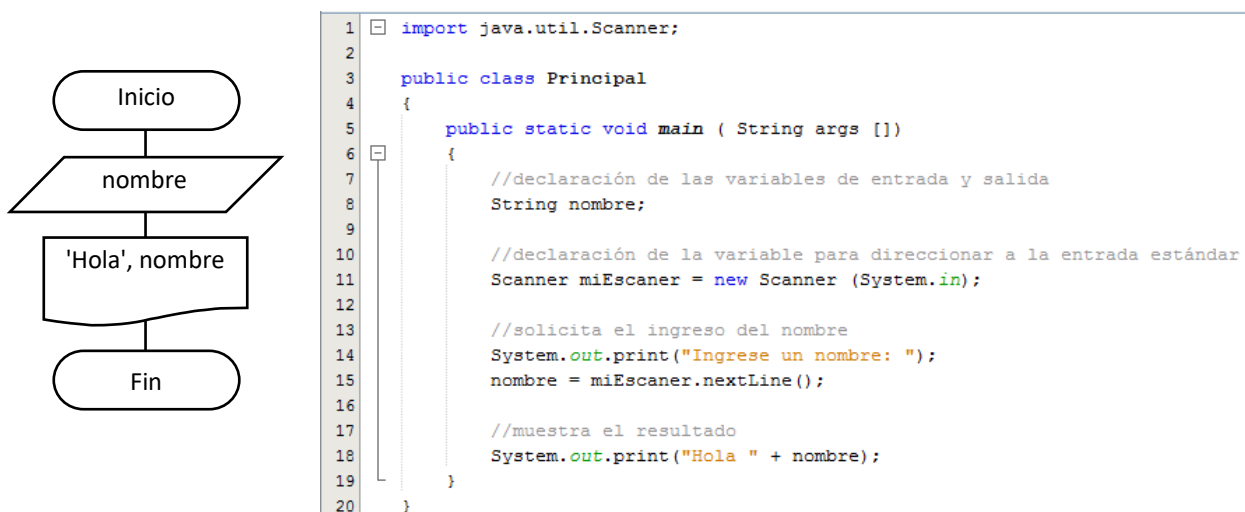
Observar que, de esta forma, cada vez que se ejecuta el programa se puede cargar un valor distinto en cada variable, y obtener diferentes resultados sin tener que modificar y recompilar el programa. Es necesario aclarar que los valores reales, tanto float como double, deben escribirse con el separador de decimal coma (,).



**Ejemplo 1):** Un programa que obtiene y muestra la suma de dos números, quedará representado y codificado de la siguiente forma:



**Ejemplo 2.)** Un programa que asigna el nombre de una persona y le muestra un saludo:



## 10.] Introducción a la Programación Orientada a Objetos (POO) en Java.

La *Programación Orientada a Objetos* (abreviada de ahora en más como *POO*) es un conjunto de reglas y principios de programación (o sea, un *paradigma* de programación) que busca representar las *entidades* u *objetos* del dominio (o enunciado) del problema dentro de un programa, en la forma más natural posible.

En el paradigma de programación tradicional o *estructurado*, el programador busca identificar los *procesos* (en forma de subproblemas) que aplicados sobre los datos permitan obtener los resultados buscados. Y esta forma de proceder no es en absoluto incorrecta: la estrategia de descomponer un problema en subproblemas es una técnica elemental de

resolución de problemas que los programadores orientados a objetos siguen usando dentro del nuevo paradigma. Entonces, ¿a qué viene el paradigma de la POO?

Los lenguajes de programación estructurada se orientaron a esa forma de trabajar descomponiendo procesos en subprocesos y programando a cada uno como *rutina*, *función*, o *procedimiento* (todas formas de referirse al mismo concepto, que luego la POO designaría en forma más general como *método*) Pero en la práctica, la orientación al subproblema del paradigma estructurado resulta ser una forma de programar que hace que sea difícil de aplicar en el desarrollo de sistemas *realmente* grandes (estamos hablando de 50000 líneas de código o más...) o en el desarrollo de sistemas de mucha complejidad en cuanto a las relaciones entre los procesos detectados. La sola orientación a la descomposición en subproblemas no alcanza cuando el sistema es tan complejo: se vuelve difícil de visualizar su estructura general, se hace complicado realizar hasta las más pequeñas modificaciones sin que estas reboten en la lógica de un número elevado de otras rutinas, y finalmente se torna una tarea casi imposible replantear el sistema para agregar nuevas funcionalidades complejas que permitan que ese sistema simplemente siga siendo útil frente a continuas nuevas demandas...

La POO significó una nueva visión en la forma de programar, buscando aportar claridad y naturalidad en la manera en que se plantea un programa. Ahora, el objetivo primario no es identificar procesos sino identificar *actores*: las *entidades* u *objetos* que aparecen en el escenario o dominio del problema, tales que esos objetos tienen no sólo datos asociados sino también algún comportamiento que son capaces de ejecutar. Piense el lector en un *objeto* como en un *robot virtual*: el programa tendrá muchos robots virtuales (objetos de software) que serán capaces de realizar eficiente y prolijamente ciertas tareas en las que serán expertos, e interactuando con otros robots virtuales (objetos...) serán capaces de resolver el problema que el programador esté planteando.

Para hacer eso, los lenguajes orientados a objetos (como Java) usan descriptores de entidades conocidos como *clases*. Básicamente, una *clase* es la descripción de una entidad u objeto de forma que luego esa descripción pueda usarse para crear muchos objetos que respondan a la descripción dada. Para establecer analogías, se puede pensar que una clase se corresponde con el concepto de *tipo de dato* de la programación estructurada tradicional, y los objetos creados a partir de la clase (llamados *instancias* en el mundo de la POO) se corresponden con el concepto de *variable* de la programación tradicional. Así como el *tipo* es uno solo y describe la forma que tienen todas las muchas *variables* de ese tipo, la *clase* es única y describe la forma y el comportamiento de los muchos *objetos* de esa clase.

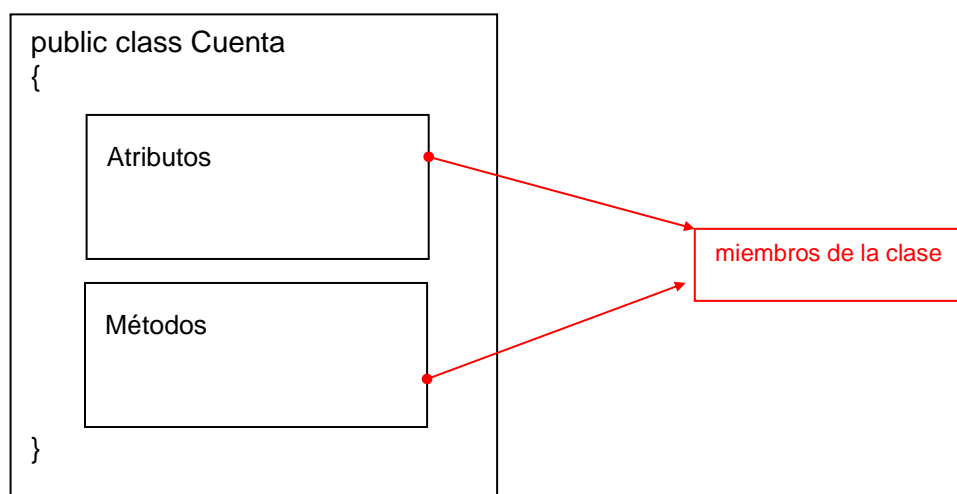
Para describir objetos que responden a las mismas características de forma y comportamiento, se declara una *clase*. La definición de una clase incluye esencialmente dos elementos:

- **Atributos:** Son *variables* que se declaran dentro de la clase, y sirven para indicar la *forma* de cada objeto representado por esa clase. Los atributos, de alguna manera, muestran lo que cada objeto *es*, o también, lo que cada objeto *tiene*.
- **Métodos:** Son funciones, procedimientos o rutinas declaradas dentro de la clase, usados para describir el *comportamiento* de los objetos descriptos por esa clase. Los métodos, de alguna manera, muestran lo que cada objeto *hace*.

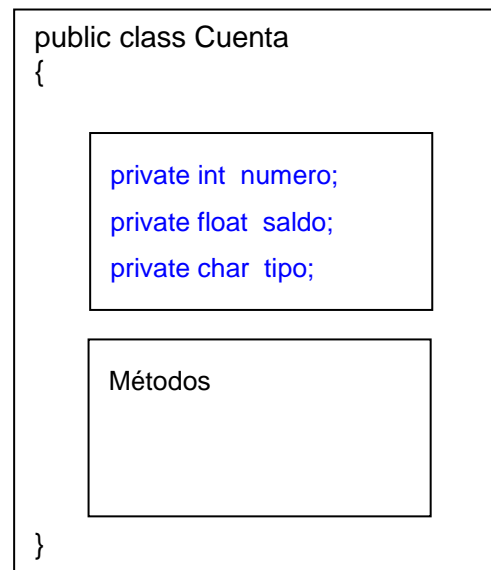
Como sabemos, una aplicación o programa Java típico debe incluir un método especial, llamado *main()*. Ese método es el primero que se ejecuta cuando se pide lanzar la aplicación, y desde allí se administra la participación de cada *instancia* u *objeto* creado.

Al desarrollar una aplicación en Java, el programador creará por lo general varias clases que luego se usarán a su vez para crear objetos que trabajarán juntos. Es común en ese sentido declarar una clase cuyo único objetivo sea contener al método *main()*. En el contexto de nuestro curso, eso es lo que hemos hecho y seguiremos haciendo mediante nuestra convención de declarar una clase con el nombre *Principal*, para que sea ella la que contenga a *main()*. No obstante, digamos que *main()* podría estar incluido en cualquier clase, y que incluso cada clase del proyecto podría tener un *main()*. No es obligatorio que una clase se llame *Principal*, y que esta deba contener a *main()*. Es sólo una convención de trabajo en el curso.

La definición de una clase en Java se hace mediante la palabra reservada *class*, seguida de un par de llaves que delimitan su contenido. Es común (pero no obligatorio) que los atributos de la clase se declaren antes que los métodos. El conjunto de atributos y métodos de una clase se conoce como el conjunto de *miembros* de la clase. Supongamos que se desea desarrollar un programa que sea capaz de manejar cuentas abiertas en un banco. En tal caso, el programa podría contar con una clase llamada *Cuenta* que describa la idea de cuenta bancaria, y luego podrán crearse tantos objetos de esa clase como cuentas se desee manejar en el programa:



Los atributos de la clase son variables, que pueden ser de tipo primitivo o pueden ser objetos de otras clases. De hecho, *String* es una clase de Java y no un tipo primitivo, por lo cual si algún atributo de la clase *Cuenta* fuera un *String*, tendríamos un atributo que es un objeto de otra clase, y así los objetos comienzan a colaborar entre ellos... Para la clase *Cuenta*, mínimamente sería necesario contar con un atributo que indique el número de la cuenta, otro para indicar el saldo en la misma, y quizás un tercero que indique de qué tipo es la cuenta: podríamos usar un atributo de tipo *char* tal que si se trata de una cuenta corriente ese atributo se asigne con una 'C' y si se trata de una cuenta de inversión (como una caja de ahorro o un plazo fijo) el atributo se asigne con 'I'. Ahora la clase podría verse así:



Existe en el ámbito de la POO un conocido principio de buena práctica, llamado *Principio de Ocultamiento*, por el cual se sugiere que al definir una clase, no se permita que los atributos sean accesibles en forma directa desde el exterior de la clase, sino que se usen métodos de la misma para consultar sus valores o modificarlos. La idea detrás del *Principio de Ocultamiento* es que el programador que use una clase predefinida no deba preocuparse por los detalles de implementación interna de la clase, sino que simplemente use los métodos que la misma provee y se maneje en un nivel de abstracción más alto. Así, si usamos objetos de la clase *String*, no debemos preocuparnos por cómo está representada esa cadena dentro del objeto. Sabemos que dentro hay una cadena implementada de alguna forma convincente, y la clase brinda todos los métodos para hacer lo que necesitemos hacer. Además, el Principio de Ocultamiento permite que la clase controle qué valores tiene cada atributo y de qué forma esos valores deberían cambiar. Si se permite el acceso a un atributo de la clase desde fuera de ella, podría provocarse que un valor incorrecto, no validado, sea asignado en ese atributo haciendo que desde allí en adelante algún proceso interno de la clase falle...

### 11.] Modificadores de acceso.

La forma que Java provee para que un programador obligue a respetar el Principio de Ocultamiento son los llamados *calificadores de acceso*. Se trata de ciertas palabras reservadas que colocadas delante de la declaración de un atributo o de un método de una clase, hacen que ese atributo o ese método tengan accesibilidad más amplia o menos amplia desde algún método que no esté en la clase. Así, los calificadores de acceso en Java pueden ser cuatro: *public*, *private*, *protected*, y *"default"* (este último no es una palabra reservada: es el estado en el que queda un miembro si no se antepone ninguno de los otros tres calificadores anteriores):

- **public:** un miembro público es accesible tanto desde el interior de la clase (por sus propios métodos), como desde el exterior de la misma (por métodos de otras clases).
- **private:** sólo es accesible desde el interior de la propia clase, usando sus propios métodos.
- **protected:** aplicable en contextos de herencia (tema que veremos más adelante), hace que un miembro sea público para sus clases derivadas y para clases en el mismo paquete, pero los hace privados para el resto. ☹!!!

- **"default"**: un miembro que no sea marcado con ningún calificador de acceso, asume estatus de acceso *por defecto*, lo cual significa que será público para clases en el mismo paquete, pero privado para el resto. Notar (otra vez) que la palabra "default" en realidad no es una palabra reservada, sino sólo el nombre del estado en que queda el miembro.

Por todo lo expresado, es que en la clase *Cuenta* se han declarado los atributos como *private*, y es también el motivo por el cual se sugería que las variables de la clase *Principal* en los ejemplos desarrollados hasta ahora se definieran privados...

Entre los métodos de una clase, el más característico es el llamado método *constructor*. Un constructor es un método cuyo objetivo básico es el de inicializar los atributos de un objeto en el momento en que ese objeto o instancia se crea. Como veremos en breve, un objeto se crea en Java usando un operador del lenguaje llamado *new*, y el constructor se invoca *automáticamente* al crear con *new* una instancia de la clase. Un constructor lleva el mismo nombre que la clase que lo contiene. No se debe indicar ningún tipo devuelto para él (ni siquiera *void*), y puede recibir parámetros como un método normal. Por otra parte, tanto los constructores como cualquier otro método de la clase, pueden ser *sobrecargados*. Eso significa que se pueden definir *varias versiones del mismo método*. El compilador distingue entre las diversas sobrecargas de un método *por la forma de su lista de parámetros*. Por lo tanto, si un método tiene varias versiones definidas en una clase, las distintas versiones deben diferir en la cantidad de parámetros, en el tipo de los parámetros, o en ambas características. Notar que el cambio en el tipo devuelto por el método no define una nueva sobrecarga: solo las formas diversas en la lista de parámetros. En base a esto, la clase *Cuenta* podría tener este aspecto si agregamos algunos constructores:

```
6 public class Cuenta
7 {
8     private int numero;
9     public float saldo;
10    private char tipo;
11
12    public Cuenta( )
13    {
14        numero = 0;
15        saldo = 0;
16        tipo = 'C';
17    }
18
19    public Cuenta(int num, float sal, char t)
20    {
21        numero = num;
22        saldo = sal;
23        tipo = t;
24    }
25
26    public Cuenta(int num)
27    {
28        numero = num;
29        saldo = 0;
30        tipo = 'C';
31    }
32 }
```

## 12.] Referencias y creación de objetos.

Así como está definida, la clase *Cuenta* está todavía muy incompleta pero ya puede usarse para crear objetos y mostrar la forma básica de usar un constructor. Supongamos entonces

que se quieren crear dos o tres objetos de la clase *Cuenta* para comenzar a trabajar con ellos. El método *main()* de nuestra clase *Principal* podría hacerlo:

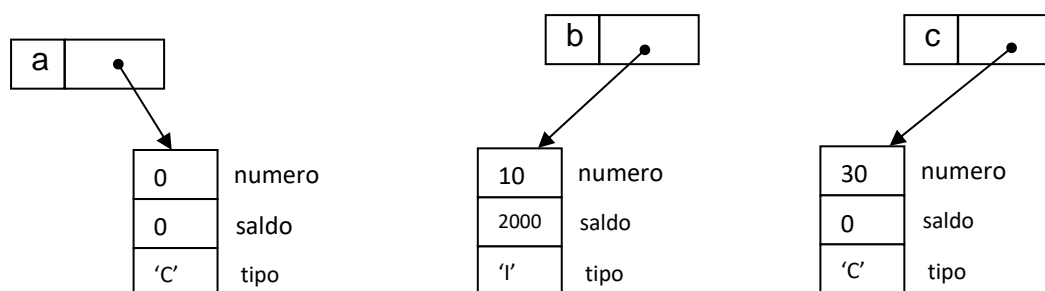
```

6
7 public class Principal
8 {
9     public static void main ( String args[] )
10    {
11        // se declaran tres variables para manejar tres objetos
12        Cuenta a, b, c;
13
14        // se crean tres objetos ahora...
15        a = new Cuenta( ); // se invoca al primer constructor...
16        b = new Cuenta( 10, 2000, 'I' ); // se invoca al segundo...
17        c = new Cuenta( 30 ); // se invoca al tercero...
18    }
19 }
20

```

Note el uso del operador *new* para crear los objetos: ese operador crea un objeto de la clase que se le indique, cada vez que se lo usa. Pero en cada creación, *new* llama a algún constructor que figure en la clase del objeto que se pide crear, y la decisión de cuál constructor llamar depende de la lista de parámetros actuales que se pase a ese constructor al invocarlo: puede verse que al crear el objeto *a*, se está invocando al primero de los constructores pues la lista de parámetros actuales se dejó vacía (y eso coincide con la forma de la lista de parámetros del primer constructor...)

En síntesis: el primer *new* crea un objeto que será manejado por la variable *a*, y en ese objeto los atributos *numero*, *saldo* y *tipo* quedan valiendo los valores { 0, 0, 'C' } respectivamente. El objeto manejado por *b* queda valiendo {10, 2000, 'I'} y el objeto manejado por *c* quedará valiendo los valores {30, 0, 'C'}:



### 13.] Metodos de acceso de lectura y escritura.

Más adelante seguiremos mostrando la forma de manejar estos objetos desde *main()*. Por ahora, sigamos con la clase *Cuenta*. Si los atributos de una clase se definen privados, entonces la clase debe proveer métodos que permitan el acceso al valor de esos atributos, tanto para consultar su valor como para modificarlo (si es decisión del programador de la clase permitir esas operaciones). Una convención fuerte en Java es que en los identificadores de esos métodos intervengan las palabras "get" (para los métodos de consulta) y "set" para los métodos modificadores, junto al nombre del atributo al que se quiere proveer acceso. Así, si un atributo se llama "clave", el método de consulta para el mismo podría llamarse "getClave()" y el modificador "setClave()". En la clase *Cuenta*, incluimos ahora este conjunto de *métodos accesores*:

```
6      public class Cuenta
7      {
8          private int numero;
9          public float saldo;
10         private char tipo;
11
12         public Cuenta( )
13         {
14             numero = 0;
15             saldo = 0;
16             tipo = 'C';
17         }
18
19         public Cuenta(int num, float sal, char t)
20         {
21             numero = num;
22             saldo = sal;
23             tipo = t;
24         }
25
26         public Cuenta(int num)
27         {
28             numero = num;
29             saldo = 0;
30             tipo = 'C';
31         }
32
33
34         public int getNumero()
35         {
36             return numero;
37         }
38
39         public void setNumero(int num)
40         {
41             numero = num;
42         }
43
44         public float getSaldo()
45         {
46             return saldo;
47         }
48
49         public void setSaldo(float sal)
50         {
51             if(sal > 0) saldo = sal;
52         }
53
54         public char getTipo()
55         {
56             return tipo;
57         }
58
59         public void setTipo (char t)
60         {
61             tipo = t;
62         }
63     }
64
```

Note que cada uno de los métodos de la clase se ha marcado como *público* (incluso los constructores). Esto es consecuencia directa de respetar el *Principio de Ocultamiento*, que en última instancia aconseja declarar privados a los atributos y públicos a los métodos de una clase. Los métodos accesoros son muy sencillos: cada uno de los métodos tipo *get* retorna el valor del atributo con el cual se asocia, y cada método tipo *set* cambia el valor de ese atributo tomando el nuevo valor como parámetro.

Una vez que se han creado objetos de una clase (por ejemplo, lo que vimos en el método *main()*), estos objetos pueden comenzar a *invocar métodos* para que se apliquen sobre sus atributos. La forma de hacerlo, consiste en usar la variable que maneja al objeto y *colocar luego de ella un punto, seguido del nombre del método* que se quiere invocar para ese objeto:

```
5
6 public class Principal
7 {
8     public static void main ( String args[] )
9     {
10         // se declaran tres variables para manejar tres objetos
11         Cuenta a, b, c;
12
13         // se crean tres objetos ahora...
14         a = new Cuenta( ); // se invoca al primer constructor...
15         b = new Cuenta( 10, 2000, 'I' ); // se invoca al segundo...
16         c = new Cuenta( 30 ); // se invoca al tercero...
17
18
19         // se invocan métodos para esos objetos...
20         int x = b.getNumero();
21         a.setNumero(x);
22         char t = b.getTipo();
23         c.setTipo(t);
24
25     }
26 }
27
```

Se dice que el objeto desde el cual se invoca al método está *llamando* al método, o también se dice que a ese objeto se le está *pasando un mensaje*. En el ejemplo, el objeto *b* está llamando al método *getNumero()* (o también decimos que *b* recibe el mensaje de retornar su número). Note que al invocar un método, ese método se aplica sobre el objeto que lo invocó (y sobre sus atributos internos, obviamente). Si se dice: *b.getNumero()*, será el objeto *b* quien retorne su número, y no los objetos manejados por *a* o por *c*.

Finalmente, digamos que toda clase Java en última instancia hereda o se define a partir de otra muy general llamada *Object*, la cual provee ya definidos una serie de métodos elementales. Varios de esos métodos se usan tal como vienen desde *Object*, pero algunos deberían ser redefinidos por cada clase. El método *toString()* es uno de ellos, y se usa para retornar una cadena de caracteres con el contenido del objeto invocante, de forma que sea adecuadamente visualizable en un dispositivo de salida. Si no se redefine, el método *toString()* retorna una cadena con el nombre de la clase a la cual pertenece el objeto, más la dirección de memoria de ese objeto en formato hexadecimal. En general, nuestras clases



deberían contar con una versión propia del método *toString()*, lo cual es normalmente fácil de hacer. Mostramos la clase *Cuenta* completa con ese método incluido al final:

```
6   public class Cuenta
7   {
8       private int numero;
9       public float saldo;
10      private char tipo;
11
12      public Cuenta( )
13      {
14          numero = 0;
15          saldo = 0;
16          tipo = 'C';
17      }
18
19      public Cuenta(int num, float sal, char t)
20      {
21          numero = num;
22          saldo = sal;
23          tipo = t;
24      }
25
26      public Cuenta(int num)
27      {
28          numero = num;
29          saldo = 0;
30          tipo = 'C';
31      }
32
33
34      public int getNumero()
35      {
36          return numero;
37      }
38
39      public void setNumero(int num)
40      {
41          numero = num;
42      }
43
44      public float getSaldo()
45      {
46          return saldo;
47      }
48
49      public void setSaldo(float sal)
50      {
51          if(sal > 0) saldo = sal;
52      }
53
54      public char getTipo()
55      {
56          return tipo;
57      }
58
59      public void setTipo (char t)
60      {
61          tipo = t;
62      }
63
64      // nuestra propia versión del método toString()
65      public String toString()
66      {
67          return "Número: " + numero + "\tSaldo: " + saldo + "\tTipo: " + tipo;
68      }
69
70  }
```

De esta forma, si queremos mostrar en la consola de salida una representación del contenido de un objeto de la clase *Cuenta*, podemos hacerlo así:

```

6  import java.util.Scanner;
7
8  public class Principal
9  {
10     public static void main ( String args[] )
11     {
12         Scanner miScanner = new Scanner(System.in);
13         // se declaran tres variables para manejar tres objetos
14         Cuenta a, b, c;
15
16         // se crean tres objetos ahora...
17         a = new Cuenta( ); // se invoca al primer constructor...
18         b = new Cuenta( 10, 2000, 'I' ); // se invoca al segundo...
19         c = new Cuenta( 30 ); // se invoca al tercero...
20
21
22         // se invocan métodos para esos objetos...
23         int x = b.getNumero();
24         a.setNumero(x);
25         char t = b.getTipo();
26         c.setTipo(t);
27
28
29         System.out.print( "Ingrese un saldo para la primera cuenta: " );
30         float s = miScanner.nextFloat();
31         a.setSaldo( s );
32
33         c.setTipo('I');
34
35         x = b.getNumero();
36         a.setNumero(x);
37
38         t = b.getTipo();
39         c.setTipo(t);
40
41         System.out.println( "Cuenta a: " + a );
42         System.out.println( "Cuenta b: " + b.toString() );
43         System.out.println( "Cuenta c: " + c.toString() );
44     }
45 }
46
System.out.println("Cuenta a: " + a.toString());

```

El método está invocado requiriendo:

también puede escribirse así, y el resultado es exactamente el mismo:

```
System.out.println("Cuenta a: " + a);
```