

# Ficha 2

## Instrucciones de Control de Flujo

---

### 1.] El proceso de compilación y ejecución de un programa por línea de órdenes.

En la práctica un programador usará un *IDE* (*Entorno Integrado de Desarrollo*) que le permitirá realizar todas las tareas que vamos a documentar a continuación desde una misma herramienta de software para facilitar su trabajo.

Pero es importante que ese programador entienda a la perfección el proceso de compilación y ejecución de Java, ya sea para dominar todos los aspectos técnicos y poder comprender los problemas que podrían surgir, como para poder salir delante de todos modos si le toca trabajar en un contexto donde no tenga acceso a un IDE. En este último caso, deberá poder hacer todo el proceso usando *directamente* los programas del JDK. Y para eso deberá salir a la línea de comandos del sistema operativo o terminal (en Windows, la salida a la línea de órdenes se logra activando el *botón Inicio*, luego escribiendo "cmd" [abreviatura de *command*] (lo cual busca una aplicación con ese nombre) y presionando <Enter>. Aparece la *pantalla negra de la consola* del sistema operativo o *línea de comandos o terminal*).

Nos proponemos a continuación construir un programa que normalmente es utilizado como primer ejemplo en los cursos de programación. El mismo es llamado *Hola Mundo* y tiene por único objetivo mostrar la cadena "*Hola Mundo*" en la pantalla. La primera vez que apareció documentada la idea de construir un *Hola Mundo* fue en el libro *El Lenguaje de Programación C* de *Brian Kernighan* y *Dennis Ritchie*: en dicho libro los autores buscaban la forma de implementar un primer programa que no tenía por objetivo revisar el código necesario para construir el mismo sino los elementos principales requeridos para transitar el camino que va desde la codificación hasta lograr la ejecución de ese primer programa para un lenguaje de programación específico, y la idea fue tan exitosa que luego distintos autores de casi todos los lenguajes de programación la copiaron para introducir este mismo contenido en los lenguajes que estaban documentando. Nosotros nos proponemos entonces ahora construir un "*Hola Mundo*" en Java, siguiendo el mismo concepto. Los pasos a seguir son los siguientes:

#### a.) Generar el archivo de *código fuente*.

El algoritmo para el programa *Hola Mundo* es prácticamente trivial y por ello no nos vamos a detener demasiado aquí al respecto : estaría compuesto simplemente de: *Inicio* -> *Imprimir "Hola Mundo"* -> *Fin*, pero sí es importante comprender que siempre, incluso en este caso, de allí debemos partir.

Luego de determinar el algoritmo lo que sigue es codificar dicho algoritmo en un lenguaje de programación, en este caso Java, y para ellos lo primero que vamos a tener que hacer es crear un archivo de texto, cosa que podemos hacer con cualquier editor de texto. Todos los sistemas operativos incluyen al menos una herramienta de edición de archivos de texto

plano, en el caso de Windows esa herramienta puede el *Block de Notas* (Cuidado: Word NO es un editor de archivos de texto plano).

Al archivo de texto en el que escribimos el código del programa o codificamos el algoritmo, se lo conoce comúnmente en programación como *archivo de código fuente*, y es generalmente el foco principal de nuestro trabajo en el proceso de programar en todos los lenguajes de programación.

En dicho archivo de código fuente vamos a escribir el siguiente código Java:

```

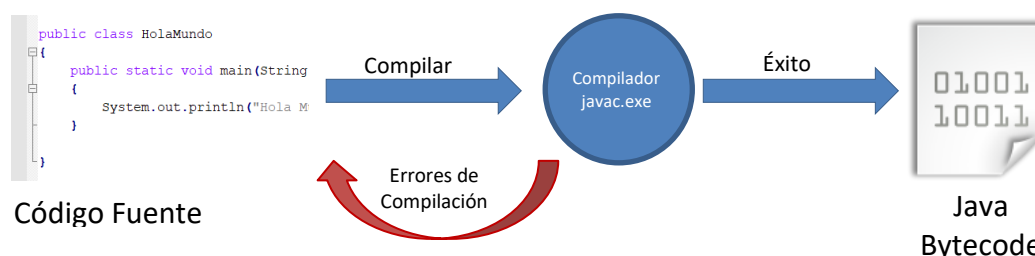
6
7 public class HolaMundo
8 {
9     public static void main(String[] args)
10    {
11        System.out.println("Hola Mundo!!!");
12    }
13
14 }

```

Insistimos en que en este momento no nos interesa abordar elementos de código, pero sí necesitamos recordar la primera restricción que tiene el lenguaje de programación Java y tiene que ver con el nombre de dicho archivo de código fuente. El archivo de código fuente de una clase Java **debe** llamarse exactamente igual que el nombre de la clase (el identificador que aparece luego de la palabra *class* en el código) y además **debe** llevar la extensión *.java*. Si cualquier de estos dos elementos no se cumple no podremos continuar con el proceso ya que obtendremos un error de compilación que a continuación explicaremos.

### b.) Compilar el programa.

Una vez que hemos escrito el programa en lenguaje Java en un archivo de código fuente y lo hemos grabado con el nombre idéntico al de la clase y la extensión *.java* viene el proceso de *compilación*, para el que vamos a utilizar el compilador Java. En la ficha anterior se describieron las funciones que cumple el compilador y no es nuestro objetivo profundizar más en el tema. Desde un punto de vista concreto el compilador Java es un programa que forma parte del JDK, es decir que para contar con dicho programa en nuestra PC debemos haber descargado e instalado el JDK. Dicho programa (en la plataforma Windows) se llama *javac.exe* y reside en la carpeta *bin* de la instalación del JDK.



Para compilar un archivo de código fuente, debemos escribir la siguiente instrucción en la línea de comandos de la terminal (la ruta de directorios indicada en el prompt obviamente puede ser diferente):

```
E:\Facu\111000\Material>javac HolaMundo.java
E:\Facu\111000\Material>
```

Si luego de ejecutar *javac <NombreClase>.java* simplemente vuelve a parecer la línea de comandos, habremos tenido éxito y en ese caso el compilador habrá generado el archivo de *Java Bytecode* que es el resultado de la compilación en Java.

Pero si hemos cometido algún error al codificar el programa, entonces el compilador mostrará en la consola información acerca del error cometido, como muestra el ejemplo siguiente (en lugar de simplemente volver a aparecer la línea de comandos):

```
E:\Facu\111000\Material>javac HolaMundo.java
HolaMundo.java:11: error: ';' expected
    System.out.println("Hola Mundo!!!")
                                ^
1 error
E:\Facu\111000\Material>
```

En este caso el compilador está avisando que en el programa faltó un punto y coma al final de la invocación al método *println()*, por lo que deberemos volver a editar el archivo de código fuente para corregir el error, agregar el punto y coma, y luego repetir el proceso.

En el caso de que el proceso de compilación haya resultado exitoso, obtendremos como resultado un nuevo archivo, que tendrá siempre un nombre compuesto por el nombre de la clase (es decir, el mismo nombre que el archivo *.java*), pero ahora con la extensión *.class*.

Este archivo de código binario de Java es un archivo que ya no es legible fácilmente por el ser humano, pero tampoco es una aplicación directamente ejecutable como en el caso de otros lenguajes de programación que utilizan compiladores. Es lo que en algunas plataformas se denomina *código intermedio* y será la materia prima para el resto del proceso.

Es importante decir en este punto es que este archivo de *bytecode* es una de las principales razones que hacen que Java sea *multiplataforma* como explicaremos más adelante. Y también que es el resultado de nuestro trabajo: si bien más adelante revisaremos mecanismos por los cuales empaquetar las aplicaciones y librerías que programamos, un programa en Java es en esencia un conjunto de archivos *.class*, es decir, un conjunto de archivos *Java Bytecode*.

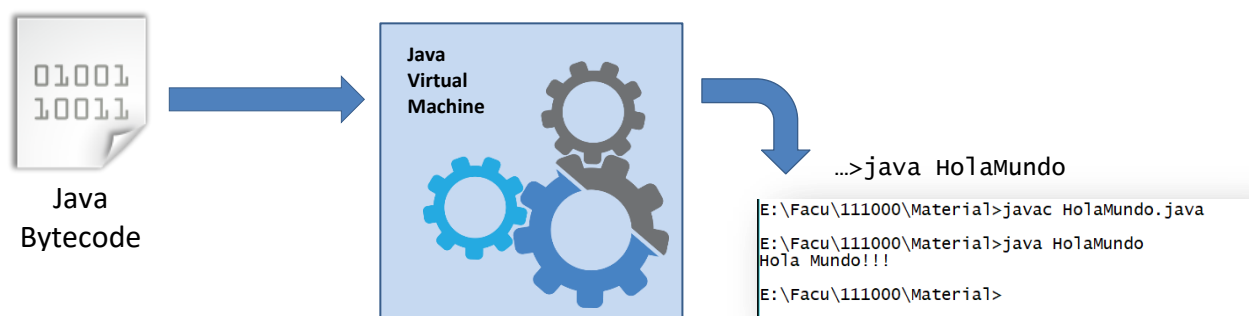
### c.) Ejecutar el programa.

Habiendo obtenido el archivo de código binario de java a partir de un proceso de compilación exitosa, ahora nos resta *ejecutar* el programa, pero como ya hemos dicho antes, esta ejecución no es automática ni independiente pues el código binario Java (el archivo *.class*) **NO ES** una aplicación ejecutable en forma directa.

Para ejecutar el programa contenido en un archivo *.class* nos hace falta la *Máquina Virtual Java* o *JVM* por sus siglas en inglés. Si hemos descargado e instalado el JDK ya contamos con dicha máquina virtual puesto que como explicaremos en el siguiente apartado está

incluida para probar lo que estamos desarrollamos. Si necesitáramos ejecutar nuestro programa en una máquina donde no vamos a programar, no es necesario descargar el JDK sino que tenemos la opción de descargar un *JRE* (por *Java Runtime Environment*) o *Entorno de Ejecución Java* que solo tiene los elementos necesarios para ejecutar una aplicación Java y no las herramientas de programación (el *JRE*, por caso, no provee el compilador *javac.exe*).

Para ejecutar la aplicación, solo debemos escribir *java <NombreClase>* en la línea de órdenes. Notar que **no** escribimos *.class* sino sólo el nombre de la clase:



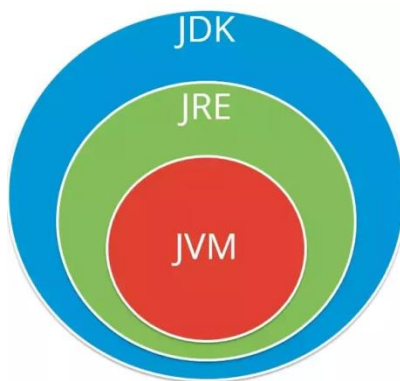
Aquí podría surgir la pregunta: ¿cómo hace la máquina virtual para encontrar el archivo con el bytecode correspondiente a la clase? Si hacemos un poco de memoria, cuando comenzamos planteamos una sola restricción que era que el nombre del archivo de código fuente debía llamarse igual que la clase que contenía; y luego dijimos que el compilador siempre designa al archivo resultado de la compilación con el mismo nombre que el archivo *.java* pero con la extensión *.class*. De allí que si ejecutamos un programa invocando a la máquina virtual y diciéndole el nombre de la clase que tiene que ejecutar, la máquina virtual puede inferir que debe buscar un archivo con el nombre compuesto por *<nombre de la clase>* más la extensión *.class*.

## 2.] ¿Por qué se dice que Java es *multiplataforma*?

Hasta ahora hemos recorrido el camino que nos llevó desde el algoritmo original que nos propusimos para imprimir la cadena *"Hola Mundo"* en la pantalla, hasta el programa que que efectivamente lo hizo.

Ahora bien: ¿dónde explícitamente radica el hecho de que la capacidad multiplataforma? Para comprender este concepto primero debemos al menos revisar la diferencia que existe entre *JDK* y *JRE* específicamente y dónde radica la máquina virtual. Respecto del *JDK* ya hemos explicado bastante y en resumidas cuentas sin descargar e instalar el *JDK* no podemos programar puesto que este paquete es el que provee el compilador Java entre otras muchas herramientas. Por otro lado el *JDK* también provee un *JRE* para probar lo que programamos en la misma computadora. El punto está en que también podemos descargar e instalar de forma separada sólo el *JRE*, y en este caso no estaríamos en condiciones de programar, pero sí de ejecutar un programa ya compilado.

La gráfica siguiente muestra la forma en que la JVM está incluida en el *JRE*, y este a su vez está incluido en el *JDK*:



Entonces podemos desarrollar el proceso de escritura de código y compilación en cualquier plataforma o sistema operativo para el cual contemos con un *JDK*, es decir, para el cual exista una versión de *JDK* que podamos descargar e instalar. Con esto obtendremos como resultado de esta fase un archivo o conjunto de archivos *Java Bytecode* o *.class*, los cuales ya dijimos cumplen un rol importante aquí: es evidente que en la misma máquina donde programamos y compilamos podemos también ejecutar la aplicación, ya que como dijimos, el *JDK* incluye un *JRE*.

Pero si fuese necesario ejecutar la aplicación en una computadora donde no esté previsto programar en Java (y posiblemente nunca esté previsto hacerlo, como por ejemplo una computadora de uso familiar, en la que casi seguro no estará instalado el *JDK*, pero que podría tener instalado un *JRE*), podríamos en ese caso tomar los archivos de bytecode (*.class*) copiarlos a la plataforma destino para la cual cuento con un *JRE* y simplemente ejecutarlos allí, cosa que no sólo es posible, sino que además se lleva a cabo de manera óptima para esa plataforma.

En este escenario que describimos estaríamos en presencia de un proceso multiplataforma real en el cual aprovecharíamos esta característica del lenguaje de programación Java. En esta situación, *Java* estaría manteniendo el concepto inicial de escribir el código y compilarlo una vez en alguna plataforma para la que exista un *JDK* disponible; y luego poder ejecutarlo a partir de los archivos de bytecode, en cualquier plataforma para la que exista una *Máquina Virtual* o lo que es igual, cualquier plataforma para la que exista un *JRE* disponible.

### 3.] Estructuras de Control de Flujo (o Estructuras de Programación) en Java.

Una forma de desarrollar programas consiste en dividir los procesos en bloques (designados como *estructuras de control de flujo*) tan simples como sea posible, que esencialmente se conocen como *secuencia de instrucciones*, *instrucción de selección* (o *condicional*) e *instrucción de iteración* (o *de repetición* o *ciclo*). Estas estructuras están disponibles en todos los lenguajes modernos de programación en forma de instrucciones cuya sintaxis está predefinida en cada lenguaje. Combinando esquemas sencillos se puede llegar a construir sistemas amplios y complejos pero de fácil entendimiento.

Un programa se puede estructurar mediante el uso de tres *estructuras de control de flujo*:

- a. **Secuencia de instrucciones:** Sucesión simple de dos o más operaciones.
- b. **Instrucción condicional:** Bifurcación condicional de una o más operaciones.
- c. **Instrucción repetitiva:** Repetición de una operación mientras se cumpla una condición.

#### 4.] Secuencia de instrucciones.

Una *secuencia de instrucciones* es una sucesión de instrucciones, una debajo de la otra, que se ejecutan en el mismo orden en que aparecen. En Java, una instrucción se separa de la siguiente usando un *punto y coma* (de hecho, el punto y coma en realidad indica el final de una instrucción), y si un bloque de instrucciones pertenece a un bloque mayor (que actúe como contenedor) entonces ese bloque debe encerrarse entre *dos llaves* (el símbolo { para dar inicio al bloque, y el símbolo } para cerrar ese bloque). Un ejemplo muy simple de una *secuencia de instrucciones* podría ser el siguiente (*resaltado en rojo*), incluido dentro de un método *main()* trivial:

```
public static void main(String args[])
{
    int a, b, c;
    a = 10;
    b = 5;
    c = a + b;
    System.out.print("Suma: ", c);
}
```

Note la presencia del par de llaves para delimitar la *secuencia* que integra el bloque de acciones del método, y el uso de los ya citados puntos y comas al final de cada instrucción.

#### 5.] Instrucción condicional.

Una *instrucción condicional* contiene una *expresión lógica* que puede ser evaluada por verdadera o por falsa, y dos bloques de instrucciones adicionales designados en general como *la salida o rama verdadera* y *la salida o rama falsa*. Si un programa alcanza una instrucción condicional y en ese momento la expresión lógica es verdadera, el programa ejecutará las instrucciones de la rama verdadera (y sólo esas). Pero si la expresión es falsa, el programa ejecutará las instrucciones de la rama falsa (y sólo esas).

En el lenguaje Java una *instrucción condicional* típica se escribe (esquemáticamente) así:

```
if (expresión lógica)
{
    instrucciones de la rama verdadera
}
else
{
    instrucciones de la rama falsa
}
```

La palabra reservada *if* da inicio a la estructura condicional que posee básicamente dos salidas o ramas: la rama o *salida por verdadero* y la rama o *salida por falso* (aunque la *rama falsa* es opcional). A continuación se escribe la condición que se evalúa, *siempre* entre paréntesis. Esa es la *expresión lógica* que se quiere evaluar por verdadero o falso, recordando en este contexto, que una *expresión lógica* es una fórmula cuyo resultado es un *valor lógico* (o *valor de verdad*).

La *rama verdadera* se escribe entre llaves, inmediatamente después de cerrar el paréntesis de la expresión lógica, y la *rama falsa* (si está presente) va después de la rama verdadera, también envuelta entre llaves, pero precedida de la palabra reservada *else*. Si al evaluar la

expresión lógica la misma es cierta, se ejecuta *únicamente* el bloque de instrucciones encerrado entre llaves de la rama verdadera, y *se ignora* la rama falsa. Si la condición fuera evaluada por falso, se ejecutará *únicamente* el bloque de la rama *else*, y será ignorada la rama verdadera. Si la rama falsa no está presente y la expresión lógica es evaluada en falso, entonces continuará el programa en forma normal, saltando la instrucción condicional. La forma de escribir una instrucción condicional cuando no está presente la rama falsa es la siguiente:

```
if (expresión lógica)
{
    instrucciones de la rama verdadera
}
```

Como vimos, las instrucciones condicionales (y también las instrucciones repetitivas) se basan típicamente en chequear el valor de una *expresión lógica* para determinar el camino que seguirá el programa en su ejecución.

Para el planteo de expresiones lógicas, todo lenguaje de programación provee operadores que implican la obtención de un valor de verdad como resultado. Los más elementales son los llamados *operadores relacionales* u *operadores de comparación*, que en Java son los siguientes:

**Tabla de operadores relacionales (o de comparación) en Java.**

Operador	Significado	Ejemplo	Observaciones
==	igual que	<code>a == b</code>	retorna <i>true</i> si <i>a</i> es igual que <i>b</i> , o <i>false</i> en caso contrario
!=	distinto de	<code>a != b</code>	retorna <i>true</i> si <i>a</i> es distinto de <i>b</i> , o <i>false</i> en caso contrario
>	mayor que	<code>a &gt; b</code>	retorna <i>true</i> si <i>a</i> es mayor que <i>b</i> , o <i>false</i> en caso contrario
<	menor que	<code>a &lt; b</code>	retorna <i>true</i> si <i>a</i> es menor que <i>b</i> , o <i>false</i> en caso contrario
>=	mayor o igual que	<code>a &gt;= b</code>	retorna <i>true</i> si <i>a</i> es mayor o igual que <i>b</i> , o <i>false</i> en caso contrario
<=	menor o igual que	<code>a &lt;= b</code>	retorna <i>true</i> si <i>a</i> es menor o igual que <i>b</i> , o <i>false</i> en caso contrario

También es posible emplear operadores conocidos como *operadores lógicos* (o *conectores lógicos*) para poder chequear varias expresiones lógicas a la vez. En general, cada una de las expresiones encadenadas por un conector lógico se designa como una *proposición lógica*. Por lo tanto, una proposición lógica es una expresión formada por variables y/o constantes relacionadas entre sí mediante *operadores de comparación* (o *relacionales*), de tal forma que el resultado de la expresión será un verdadero o un falso. Los tres principales conectores lógicos en Java se ven en la tabla siguiente:

**Tabla de conectores lógicos en Java.**

Operador	Significado	Ejemplo	Observaciones
&&	conjunción lógica (y)	<code>a == b &amp;&amp; y != x</code>	ver "Tablas de Verdad" en esta misma sección
	disyunción lógica (o)	<code>n == 1    n == 2</code>	ver "Tablas de Verdad" en esta misma sección
!	negación lógica (no)	<code>! x &gt; 7</code>	ver "Tablas de Verdad" en esta misma sección

Un *conector lógico* es un operador que permite encadenar la comprobación de dos o más expresiones lógicas y obtener un resultado único. En la columna *Ejemplo* de la tabla anterior, las expresiones `a == b`, `y != x`, `n == 1`, `n == 2` y `x > 7` son proposiciones lógicas cuyos

resultados son encadenados mediante los operadores lógicos que figuran en *color azul* en cada ejemplo.

Si llamamos en general  $p$  y  $q$  a dos proposiciones lógicas cualesquiera, podemos mostrar la forma en que operan los conectores lógicos *and* (&&) y *or* (||) mediante sus respectivas tablas de verdad:

Tablas de verdad de los conectores && y ||.

Tabla de verdad del conector &&		
p	q	p && q
True	True	True
True	False	False
False	True	False
False	False	False

Tabla de verdad del conector		
p	q	p    q
True	True	True
True	False	True
False	True	True
False	False	False

Podemos ver que en el caso del conector && la salida o respuesta obtenida sólo será verdadera si las proposiciones conectadas son verdaderas al mismo tiempo, y en todo otro caso, la salida será falsa. Por lo tanto, un *and* es muy útil cuando se quiere estar seguro que todas las condiciones impuestas sean ciertas en un proceso. Por caso, suponga que se cargaron por teclado dos variables *suelo* y *antigüedad* con los datos de un empleado, y se quiere saber si ese empleado gana más de 15000 pesos y tiene al mismo tiempo una antigüedad de por lo menos 10 años, para decidir si se le otorga o no un crédito. Como *ambas* condiciones son exigibles a la vez, la pregunta requiere un conector &&, y en Java puede plantearse así:

```
if(suelo > 15000 && antigüedad >= 10)
{
    System.out.print("Crédito concedido");
}
else
{
    System.out.print("Crédito rechazado");
}
```

Note que si alguna de las dos proposiciones fuese falsa, la condición completa sería falsa y se activaría la rama *else*, rechazando el crédito. Por supuesto, lo mismo ocurriría si ambas proposiciones fuesen falsas al mismo tiempo. *Sólo si ambas fuesen ciertas*, la condición sería



cierta ella misma, y se activaría la rama verdadera. Un detalle adicional a considerar es que si el bloque que se quiere escribir (por caso, la rama verdadera o la falsa de una condición) contuviese una sólo instrucción, entonces las llaves para delimitar ese bloque son opcionales en Java. La misma instrucción que mostramos en el ejemplo anterior podría haberse planteado así:

```
if(sueldo > 15000 && antiguedad >= 10)
    System.out.print("Crédito concedido");
else
    System.out.print("Crédito rechazado");
```

El conector `//` opera de forma diferente: la salida será verdadera si al menos una de las proposiciones es verdadera. Sólo si *todas* las proposiciones al mismo tiempo son falsas, se obtendrá un falso como respuesta. El uso de un `//` es valioso (por ejemplo) cuando se quiere determinar si una variable que se acaba de cargar por teclado vale uno de varios posibles valores que se consideran correctos. Por ejemplo, supongamos que queremos saber si la variable *opcion* fue cargada con un 1, un 3, o un 5 (cualquiera de los tres valores es correcto en este ejemplo). En Java, podemos hacerlo así:

```
if(opcion == 1 or opcion == 3 or opcion == 5)
    System.out.print("Opción correcta");
else
    System.out.print("Opción incorrecta");
```

En este caso, la instrucción condicional se usa para comprobar si el valor de la variable *opcion* coincide con alguno de los números 1, 3 o 5. Cualquiera de las tres proposiciones que fuese cierta, haría cierta también la condición completa y se activaría la rama verdadera. Sólo si todas las proposiciones fuesen falsas, se activaría la rama *else* (por ejemplo, si el valor contenido en la variable *opcion* fuese el 7).

Como vimos, los conectores `&&` y `//` se aplican sobre dos o más proposiciones (en general, si un operador aplica sobre dos operandos, se lo suele designar como un *operador binario*). Pero el operador `!` (*not* o *negación lógica*) aplica sobre una sola proposición (y por lo tanto se lo suele designar como un *operador unario*) y su efecto es obtener el *valor opuesto* al de la proposición negada. Por lo tanto, la tabla de verdad del *negador lógico* es trivial:

Tabla de verdad del conector `!`.

Tabla de verdad del conector <code>!</code>	
p	! p
True	False
False	True

En general, el uso indiscriminado del negador lógico suele llevar a condiciones difíciles de leer y entender por parte de otros programadores, por lo que sugerimos se aplique con cuidado y sentido común. A modo de ejemplo, supongamos que se tiene una variable *edad* con la edad de una persona, y se quiere saber si esa persona tiene al menos 18 años para decidir si puede acceder al permiso de conducir. Una forma de hacerlo en Java, sería preguntar si la edad *no es* menor que 18:

```
if (!edad < 18)
    System.out.print("Puede acceder al permiso");
else
    System.out.print("No puede acceder al permiso");
```

Si bien lo anterior es correcto y cumple efectivamente con el requerimiento, queda claro que la pregunta podría reformularse de forma de eliminar el negador, dejándola más clara. Sólo debemos notar que preguntar si *edad* no es menor que 18, es exactamente lo mismo que preguntar si *edad* es mayor o igual que 18. El ejemplo podría replantearse así:

```
if (edad >= 18)
    System.out.print("Puede acceder al permiso");
else
    System.out.print("No puede acceder al permiso");
```

Debe observarse que los operadores `&&` (*and*) y `//` (*or*) en Java actúan en forma cortocircuitada (como en prácticamente todos los lenguajes modernos): si se analiza la tabla de verdad de ambos operadores, se ve con claridad que si se usa un `&&` (*and*) y la primera proposición es falsa, entonces la condición completa será falsa sin importar el valor de la segunda proposición (ni de ninguna de las que queden). Java reconoce ese tipo de situaciones y cuando chequea una condición que incluye un `&&` (*and*), sale por falso si la primera proposición fuese falsa, sin siquiera analizar la segunda proposición. Algo similar ocurre con el operador `//` (*or*): cuando la primera proposición es verdadera, la condición completa será verdadera sin importar el valor de las proposiciones que sigan, y Java directamente buscará la rama verdadera cuando la primera proposición sea verdadera en una expresión que contenga un `//` (*or*), sin analizar el resto de las proposiciones.

Es posible que en la rama falsa y/o en la rama verdadera de una instrucción condicional se requiera plantear otra instrucción condicional. De hecho, es posible que a su vez cada nueva instrucción condicional incluya otras y así sucesivamente según lo vaya necesitando el programador, sin límites teóricos. Cuando esto ocurre, se tiene lo que se conoce como un *anidamiento de condiciones*:

```
if(condición_01)
{
    // instrucciones que correspondan...
}
else
{
    if(condicion_02)
    {
        // instrucciones que correspondan...
    }
    else
    {
        if(condicion_03)
        {
            // instrucciones que correspondan...
        }
        else
        {
            // instrucciones que correspondan...
        }
    }
}
```

A veces es necesario evaluar si el valor de una única variable o expresión coincide puntualmente con alguno de muchos valores posibles (por ejemplo, cuando en un programa controlado por menú de opciones se quiere saber qué opción eligió el usuario). Si ese fuese el caso, el chequeo se puede resolver con instrucciones condicionales anidadas en forma similar al ejemplo anterior. Sin embargo, si se tiene que el número de alternativas es grande, usar condiciones anidadas puede plantear problemas de escritura y legibilidad del programa, debido a la gran cantidad de llaves, bloques y palabras reservadas repetidas (como *if* y *else*). Es por eso que existe un tipo especial de instrucción condicional en Java, llamada *instrucción condicional múltiple*, que suele usarse para estos casos en particular.

La *instrucción condicional múltiple* evaluará una variable (o también una expresión) que pueda tomar uno entre  $n$  valores distintos, y según cual fuese el valor de la variable o expresión chequeada, se ejecutará una de las  $n$  acciones previstas, o lo que es igual, el flujo del programa seguirá un determinado camino entre los  $n$  posibles.

En Java, la sintaxis para esta instrucción es la siguiente:

```
switch(expresión)
{
    case valor1:
        instrucciones_1;
        break;

    case valor2:
        instrucciones_2;
        break;

    case valorN:
        instrucciones_N;
        break;

    default:
        instrucciones_por_defecto;
}
```

A modo de ejemplo, supongamos que en un programa controlado por menú de opciones el usuario debe seleccionar una entre 4 opciones que se le ofrecen en pantalla. Supongamos que la variable *op* es la variable en la cual el programa asignará el valor cargado por el usuario cuando se le pida ingresar por teclado el número de la opción seleccionada. Si ese fuese el caso, una instrucción *switch* para controlar el valor de *op* podría ser la que se muestra:

```
switch(op)
{
    case 1:
        // instrucciones para la opción 1...
        break;

    case 2:
        // instrucciones para la opción 2...
        break;

    case 3:
```

```
        // instrucciones para la opción 3...
        break;

    case 4:
        // instrucciones para la opción 4...
        break;

    default:
        // instrucciones si el valor de op no
        // coincide con ninguno de los previstos...
}
```

Si no se coloca la instrucción *break* al final de cada rama, la instrucción funcionará de forma que cuando uno de los chequeos se active por verdadero, se ejecutarán también los bloques que correspondan a los valores que siguen: si *op* tuviese el valor 1 y no estuviesen los *break* al final de cada rama, se ejecutará el bloque 1, pero luego se ejecutarán también los bloques 2, 3 y 4 (como si *op* hubiese tenido todos los valores a la vez). En forma similar, si las instrucciones *break* no estuviesen y el valor de *op* fuese el 2, el bloque 1 no será ejecutado, pero se ejecutarán los bloques 2, 3 y 4. La instrucción *break* al final de cada bloque es opcional, pero evita que los bloques correspondientes a cada valor se enganchen entre sí cuando uno de ellos es activado.

La rama *default* también es opcional, y su bloque asociado sólo se ejecuta si el valor de la variable o expresión chequeada no coincide con ninguno de los valores listados en las ramas etiquetadas con *case*. En ese sentido, la rama *default* es equivalente a un *else* para la condición múltiple.

Es importante observar, además, que la variable o expresión a controlar debe ser de alguno de los tipos primitivos *byte*, *short*, *int* o *char*. A partir de la versión 7 de Java, esa variable o expresión también puede ser una referencia de la clase *String* (o sea, el *switch* puede usarse para chequear cadenas de caracteres). Y como se verá más adelante, la variable o expresión chequeada puede ser también una referencia a objetos de las clases *Byte*, *Short*, *Integer* o *Character* (que veremos en breve y se designan como clases *wrapper* o *clases de envoltorio*). Por ejemplo, el siguiente planteo para controlar el valor de la variable *mes* (asumiendo que *mes* es de tipo *String*) es válido:

```
switch (mes)
{
    case "Enero":
        // instrucciones para la opción "Enero"...
        break;

    case "Febrero":
        // instrucciones para la opción "Febrero"...
        break;

    case "Marzo":
        // instrucciones para la opción "Marzo"...
        break;

    ... // otras ramas aquí...
    ... // otras ramas aquí...
}
```

## 6.] Instrucción repetitiva.

En programación una *instrucción repetitiva* (o ciclo) es una *instrucción compuesta* que permite la repetición controlada de la ejecución de cierto conjunto de instrucciones en un programa, determinando si la repetición debe detenerse o continuar de acuerdo al valor de cierta condición que se incluye en el ciclo. Básicamente, en todo lenguaje de programación un *ciclo* consta de dos partes:

- La *cabecera del ciclo*, que incluye una *condición de control* y/o elementos adicionales en base a los que se determina si el ciclo continúa o se detiene. En el lenguaje Java, todos los ciclos repiten cada vez que la condición de control es *verdadera*, y se detienen cuando la misma es *falsa*.
- El *bloque de acciones* o *cuerpo del ciclo*, que es el conjunto de instrucciones cuya ejecución se pide repetir.

Existen dos tipos de ciclos generales, los cuales se designan con los siguientes nombres genéricos (el motivo de estos nombres se verá oportunamente en esta misma lección):

- Ciclos "0 - N"
- Ciclos "1 - N"

Estos ciclos son implementados por los diversos lenguajes en formas que varían ligeramente de un lenguaje a otro. El lenguaje Java implementa los dos tipos de ciclos mediante tres instrucciones específicas, designándolos respectivamente con las palabras reservadas que se marcan a continuación en letra negrita:

- ✓ Ciclo **for** (ciclo del tipo [0, N] )
- ✓ Ciclo **while** (ciclo del tipo [0, N] )
- ✓ Ciclo **do while** (ciclo del tipo [1, N] )

Antes de analizar la forma de trabajo de cada uno de estos tipos de ciclos en Java, es conveniente mostrar la forma en que se implementan en Java las llamadas instrucciones de conteo y de acumulación, muy útiles y de uso generalizado en cualquier lenguaje.

Cuando se necesita llevar a cabo procesos de *conteo* (por ejemplo, determinar cuántas veces apareció un número negativo), o de *sumarización* (por ejemplo, determinar cuánto vale la suma de todos los valores que tomó la variable *x* a lo largo de la ejecución de programa, suponiendo que *x* cambia de valor durante esa ejecución), se utilizan (para el primer caso) *variables de conteo* (o simplemente *contadores*), y el para segundo, se usan *variables de acumulación* (o simplemente *acumuladores*).

En ambas situaciones se trata de variables que en una expresión de asignación aparecen en *ambos miembros*: la misma variable se usa para hacer un cálculo y para recibir la asignación del resultado de ese cálculo. Los siguientes son dos ejemplos de *expresiones de conteo* o *acumulación*. En el primero, la variable *a* se usa como un *contador* (se suma una *constante* al valor previo de la variable que actúa como contador), y en el segundo la variable *b* se usa como un *acumulador* (se suma el valor de *otra variable* al valor previo de la variable que actúa como acumulador):

```
a = a + 1
b = b + x
```

Pero también notemos que cualquier *constante* y cualquier *operador* sirven para formar la expresión general de un *contador*. En los siguientes ejemplos mostramos *contadores* de

formas diversas (asegúrese de entender lo que cada instrucción hace cada vez que se ejecuta):

```
a = a + 1
b = b - 1
c = c + 2
d = d * 4
e = e / 3
```

Del mismo modo, cualquier operador es válido para formar una *expresión de acumulación*, tal como se muestra en los ejemplos que siguen:

```
s = s + x
b = b * z
a = a - y
p = p / t
c = c + 2*x
```

Es interesante notar que en Java (como en otros lenguajes) cualquier *expresión de conteo* o *de acumulación* responde a la *forma general* siguiente:

**variable = variable operador expresión;**

donde *variable* es la variable cuyo valor se actualiza (y aparece en ambos miembros de la expresión de asignación) y *expresión* es una constante, una variable o una expresión propiamente dicha (formada a su vez por constantes, variables y operadores). Y el hecho es que en el lenguaje Java cualquier expresión que venga escrita en la *forma general* anterior, se puede escribir también en la *forma resumida* siguiente:

**variable operador= expresión;**

A modo de ejemplo, veamos las siguientes equivalencias:

Forma general	Forma resumida
a = a + 1;	a += 1;
b = b - 1;	b -= 1;
c = c + 2;	c += 2;
d = d * 3;	d *= 3;
e = e / 4;	e /= 4;
s = s + x;	s += x;
b = b * z;	b *= z;
a = a - x;	a -= x;
p = p / t;	p /= t;
c = c + 2*x;	c += 2*x;

Es bueno hacer notar que en Java, *las expresiones que conteo en las que se suma al contador el valor 1 o el valor -1* (y específicamente esos dos casos...) pueden escribirse en cuatro formas posibles (y no sólo en las dos que hemos presentado antes):

Expresión normal	Resumida	Notación de post-orden	Notación de pre-orden
a = a + 1;	a += 1;	a++; (postincremento)	++a; (preincremento)
a = a - 1;	a -= 1;	a--; (postdecremento)	--a; (predecremento)

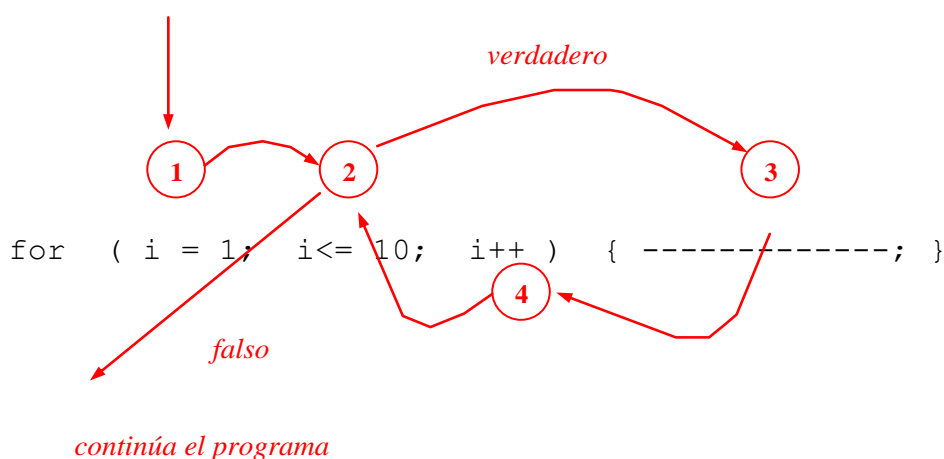
Podemos ahora mostrar la forma de implementar los tres tipos de ciclos que Java provee. Comenzaremos con el *ciclo for*, que comúnmente se usa en circunstancias en las cuales se

conoce de antemano la cantidad exacta de repeticiones que deben realizarse, aunque en el lenguaje Java la sintaxis del ciclo es tan amplia que admite ser usado en cualquier circunstancia que requiera repetición de acciones. De hecho, veremos que un ciclo *for* es una variante de un ciclo *while*. Un ejemplo es la estructura de un *for* que se ve en el modelo siguiente, suponiendo que se desea repetir 10 veces la ejecución de ciertas instrucciones (otros casos requieren variantes directas sobre la estructura mostrada):

```
for(i=1; i<=10; i++)
{
    // bloque de instrucciones a repetir...
}
```

En la cabecera del ciclo existen tres secciones: la primera ( $i=1$  en este caso) se llama *sección de inicialización*, y se usa normalmente para indicar el valor inicial de las variables de control del ciclo. La segunda ( $i<=10$  en este caso) es la *condición de control* del ciclo: si la condición es verdadera el ciclo ejecuta una repetición del bloque de acciones, pero si es falsa corta el ciclo y continúa el programa. La tercera sección ( $i++$  en este caso) es la *sección de incremento*, en la cual normalmente se indica la forma en que cambiará el valor de cada variable de control del ciclo.

Cuando el ciclo comienza, se ejecuta primero (y por única vez) la *sección de inicialización*, dando un valor inicial a la variable de control (ver acción ① en el gráfico que sigue). Luego, se verifica la *condición de control* (②). Si la misma es verdadera, se ejecuta entonces el bloque de acciones del ciclo (③), y al finalizar, se produce un retorno automático a la cabecera, específicamente a la *sección de incremento*, para cambiar el valor de las variables de control (④). Finalmente, se vuelve a verificar la *condición de control* (②). Si vuelve a obtenerse el valor verdadero, se repite lo anterior. Pero si la condición es falsa, el ciclo se detiene. Todo esto se resume en el siguiente esquema:



Observar que si la *condición de control* fuera falsa la primera vez que se evalúa, entonces el bloque de acciones *no sería ejecutado*. Esta es la característica básica de los ciclos tipo  $[0, N]$ : el cero hace referencia a la cantidad *mínima* de veces que se espera que el bloque del ciclo sea ejecutado, y la  $N$  es un valor genérico que se refiere a que una vez comenzado el ciclo, se espera que el bloque de acciones se ejecute  $N$  veces, siendo  $N$  un valor indefinido pero que se supone será conocido al momento de ejecutar el ciclo.

A modo de ejemplo inicial, el siguiente programa muestra en pantalla la sucesión de números enteros que va del 1 al  $n$ , cargando previamente  $n$  por teclado:

```

1  import java.util.Scanner;
2
3  public class Principal {
4
5      public static void main(String args[])
6      {
7          int i, n;
8          Scanner miEscaner = new Scanner (System.in);
9
10         System.out.print("Cuántos números quiere mostrar? ");
11         n = miEscaner.nextInt();
12
13         for( i = 1; i <= n; i++ )
14         {
15             System.out.println("Valor: " + i);
16         }
17     }
18 }

```

Notemos que en el lenguaje Java, el ciclo *for* permite una gran cantidad de variantes, dado que en cada sección de la cabecera no hay prácticamente restricciones a lo que se puede hacer. Los siguientes planteos son válidos:

- La sección de inicialización queda vacía, pero la variable  $c$  fue inicializada antes del ciclo, lo cual es perfectamente válido.

```

int c = 0;
for ( ; c != 20; c++) -----;

```

- Este ciclo incrementa de a dos el valor de la variable de control. Con esto se logra que el ciclo avance “de dos en dos”, y no “de a uno”.

```

int a;
for (a = 1; a <= 100; a += 2) -----;

```

- Este ciclo usa dos variables en la condición de control. Una de ellas ( $a$ ) es controlada por el propio ciclo, y la otra ( $b$ ) se inicializa e incrementa fuera de la cabecera. Notar que la condición de control puede incluir conectores lógicos (en este caso, se usa un conector  $\&\&$  para hacer un *and* entre dos proposiciones).

```

int a, b = 0;
for (a = 0; a < 10 && b != 20; a++)
{
    -----;
    b = miEscaner.nextInt();
}

```

- En éste caso, el ciclo no tiene bloque de acciones (observar el punto y coma inmediatamente después del paréntesis del cierre de la cabecera). Simplemente efectúa diez repeticiones, pero no hace nada en cada una de ellas: se puede pensar



que ejecuta un bloque de acciones vacío (esto, por cierto, también vale para el ciclo *while* o el *do while*). Si bien podría pensarse que un ciclo sin bloque de acciones no tiene ningún sentido, existen situaciones en donde se aplica. A veces, por ejemplo, se pretende que en cierto momento un programa provoque un retardo (o sea, “se demore” unos instantes antes de seguir con la instrucción siguiente). Un ciclo *for* sin bloque de acciones ajustado para efectuar 10000 o 20000 repeticiones, provocaría un retardo de un par de segundos (dependiendo de la velocidad del procesador de la computadora que se esté usando). Por supuesto, la demora será mayor si el ciclo se ajusta a valores mayores. Esta operación de provocar una demora en un programa, se suele denominar *delay* (“retardo”, en inglés).

```
int i;  
for (i = 1; i<=10; i++);
```

El segundo de los ciclos que Java ofrece es el *ciclo while*. En muchas ocasiones necesitamos plantear un ciclo que ejecute en forma repetida un bloque de acciones pero sin conocer previamente la cantidad de vueltas a realizar. Para estos casos la mayoría de los lenguajes de programación, y en particular Java, proveen el *ciclo while*, aunque como veremos, puede ser aplicado sin ningún problema también situaciones en las que se conoce la cantidad de repeticiones a realizar.

Como todo ciclo, un *ciclo while* está formado por una *cabecera* y un *bloque o cuerpo de acciones*, y trabaja en forma general de una manera muy simple. La *cabecera* del ciclo contiene una *expresión lógica* que es evaluada en la misma forma en que lo hace una instrucción condicional *if*, pero con la diferencia que el *ciclo while* ejecuta su bloque de acciones en *forma repetida* siempre que la expresión lógica arroje un valor verdadero. Así como un *if* hace una *única* evaluación de la expresión lógica para saber si es verdadera o falsa, un *ciclo while* realiza *múltiples* evaluaciones: cada vez que termina de ejecutar el bloque de acciones vuelve a evaluar la expresión lógica y si nuevamente obtiene un valor verdadero repite la ejecución del bloque y así continúa hasta que se obtenga un falso.

La característica principal del ciclo *while* es que la condición de control se evalúa por primera vez *antes* de la primera ejecución del bloque de acciones. Al igual que en el ciclo *for*, esto implica que si en la primera evaluación de la condición de control se obtiene un valor *falso*, entonces el bloque del ciclo *no será ejecutado* y por esta causa el ciclo es del tipo  $[0, N]$ . Se se escribe en la forma que se muestra más abajo, y su forma de funcionamiento no ofrece prácticamente diferencias respecto de otros lenguajes (como Python):

```
while(expresión_lógica)  
{  
    // bloque de instrucciones a repetir...  
}
```

Por ejemplo, en el programa de más abajo, se implementa un ciclo que se ejecuta mientras el valor de la variable *cont* se mantenga menor o igual a 5. En cada vuelta, se muestra un mensaje que contiene el número de vuelta en que se encuentra el ciclo. La variable *cont* recibió un valor inicial (el *cero* en este caso) antes de comenzar la ejecución del ciclo (de no ser así, la evaluación de la condición la primera vez no tendría sentido...). El bloque del ciclo contiene varias instrucciones y figura encerrado entre llaves. Cuando el programa llega al ciclo por primera vez, evalúa la condición. En este caso, el valor inicial de *cont* es cero y la

condición es cierta: *cont* es menor que 5. Siendo cierta la condición, se ejecuta el bloque de acciones, con lo cual *cont* suma 1 y luego se muestra el mensaje: Vuelta número: 1. Termina allí el bloque de acciones, pero como se trata de un ciclo, automáticamente el programa regresa a verificar otra vez la condición: si el nuevo valor de *cont* sigue siendo menor que 5, el bloque se ejecuta otra vez y así seguirá hasta que en algún retorno la condición sea falsa. Cuando *cont* tenga el valor 5, se mostrará el mensaje Vuelta número: 5 y el ciclo se detendrá. En ese momento se ejecutarán las instrucciones que se hayan escrito debajo del ciclo.

```
1  public class Principal {  
2  
3      public static void main(String args[])  
4      {  
5          int cont = 0;  
6          while ( cont < 5)  
7          {  
8              cont++;  
9              System.out.println("Vuelta número: " + cont);  
10         }  
11     }  
12 }
```

El tercer y último tipo de ciclo previsto por Java es el *ciclo do – while*, cuya característica básica es que la condición de control se evalúa por primera vez *después* de la primera ejecución del bloque de acciones. Esto implica que *al menos una* vez el bloque de acciones del ciclo siempre será ejecutado, independientemente del valor inicial de la condición de corte (por eso este ciclo es del tipo [1, N]: una vez como mínimo se ejecuta el bloque, y luego puede llegar a *N* repeticiones). La forma de escribir un ciclo *do while* en Java es la siguiente:

```
do  
{  
    // bloque de instrucciones a repetir...  
}  
while(expresión_lógica);
```

El esquema de funcionamiento del ciclo *do while* es el siguiente: cuando se llega a una línea que comienza con *do*, se ejecuta el bloque de acciones que sigue a continuación, sin importar el valor de la condición de control. Luego de ejecutar ese bloque, se evalúa la condición de control. Si dicha condición arroja un falso, el ciclo corta y continúa el programa con la instrucción que esté debajo del ciclo. Si la condición se evalúa en verdadero, se provoca un *retorno automático* a la línea marcada con *do*, y se vuelve a ejecutar el bloque de acciones del ciclo. El proceso continuará de esta forma, hasta obtener un falso en la condición de control.

El siguiente programa usa un ciclo *do while* para cargar una serie de valores, deteniendo el proceso cuando el conteo de esos valores sea igual a 10, y mostrando el valor de la suma de dichos números:

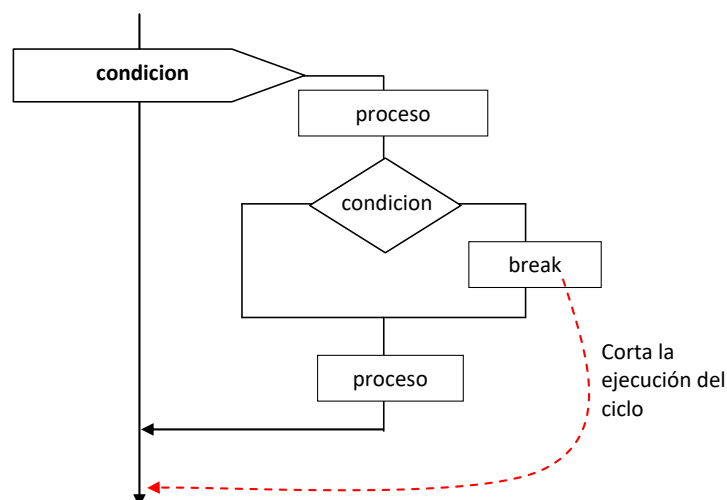
```

1  import java.util.Scanner;
2
3  public class Principal {
4
5      public static void main(String args[])
6      {
7          int x, c, ac;
8          ac = 0;
9          c = 0;
10         Scanner miEscaner = new Scanner(System.in);
11
12         do
13         {
14             System.out.print("\nIngrese un valor: ");
15             x = miEscaner.nextInt();
16             ac += x;
17             c++;
18         } while ( c < 10 );
19
20         System.out.println("\nLa suma de los valores es: " + ac);
21
22     }
23 }

```

Por supuesto, queda a criterio del programador la elección del tipo de ciclo a usar en cada situación: cualquiera de los tres ciclos provistos por Java puede usarse en cualquier situación que requiera repetición de instrucciones, mientras el programador sepa cómo adaptarlos según el caso. No obstante, es cierto que existen situaciones en las cuales algunos ciclos son más fáciles de aplicar que otros. En nuestro caso, el ciclo *do while* es especialmente útil en al menos dos situaciones típicas y muy conocidas de la programación: la *validación de valores* cargados por teclado; y el desarrollo de *programas controlados por menú de opciones*.

Habiendo mostrado la forma en que se plantean los tres tipos de ciclos en Java, completamos agregando que el bloque de acciones de un ciclo cualquiera en Java puede incluir una instrucción *break* para cortar el ciclo de inmediato sin retornar a la cabecera para evaluar la expresión lógica de control, en forma similar a otros lenguajes. El siguiente diagrama muestra el flujo de control al ejecutarse una instrucción *break* (cualquiera sea el ciclo que la contenga):



El siguiente programa tiene el objetivo de cargar por teclado cinco números positivos y calcular la suma o acumulación de todos ellos. Pero contiene un *ciclo while* que se interrumpirá con una instrucción *break* si se carga por teclado un número cero o negativo, *aún cuando no se haya llegado a las 5 repeticiones que se esperaba en el ciclo*:

```
1  import java.util.Scanner;
2
3  public class Principal {
4
5      public static void main(String args[])
6      {
7          int num, suma, i;
8          suma = 0;
9          i = 1;
10         Scanner miEscaner = new Scanner(System.in);
11
12         while (i <= 5)
13         {
14             System.out.print( "Ingrese un número mayor a cero: " );
15             num = miEscaner.nextInt();
16
17             if (num <= 0)
18                 break;
19
20             suma += num;
21             i++;
22         }
23
24         System.out.println("\nLa suma de los números ingresados es: " + suma);
25     }
26 }
```

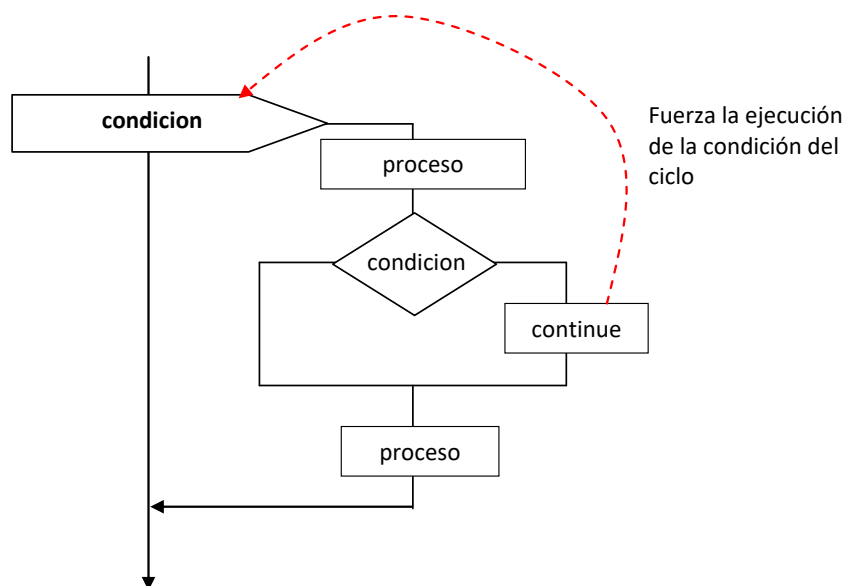
En el ejemplo anterior, el bloque de acciones del *ciclo while* carga por teclado un número *num*, y chequea con un *if* si ese número es cero o negativo. En caso de serlo, se ejecuta la instrucción *break* y su efecto es *cancelativo con respecto al ciclo*: el ciclo se interrumpe, y el programa continúa con la ejecución del *println()* que se encuentra a la salida del ciclo. Insistimos: se ejecuta el *break* y el ciclo se interrumpe, sin volver a la cabecera para chequear la expresión lógica de control del ciclo, por lo que su valor es ignorado en ese caso. Si el conjunto de números a procesar fuese {2, 4, 5, -2, 7, 1} este programa cargaría los tres primeros y los acumularía sin problemas con el ciclo, pero el ciclo se interrumpiría en el cuarto (el -2) pues siendo negativo activaría la instrucción *break*. La salida de este programa para esa secuencia de números de entrada, sería:

```
run:
Ingrese un número mayor a cero: 2
Ingrese un número mayor a cero: 4
Ingrese un número mayor a cero: 5
Ingrese un número mayor a cero: -2

La suma de los números ingresados es: 11
BUILD SUCCESSFUL (total time: 20 seconds)
```

que es la suma 2 + 4 + 5, sin incluir al resto de los positivos del conjunto.

De forma similar, pero a la inversa, un ciclo cualquiera puede incluir una instrucción *continue* para *forzar una repetición del ciclo* sin terminar de ejecutar las instrucciones que queden por debajo de la invocación a *continue*. El siguiente diagrama muestra el flujo de control al ejecutarse la instrucción *continue*, sea cual sea el ciclo que la contenga:



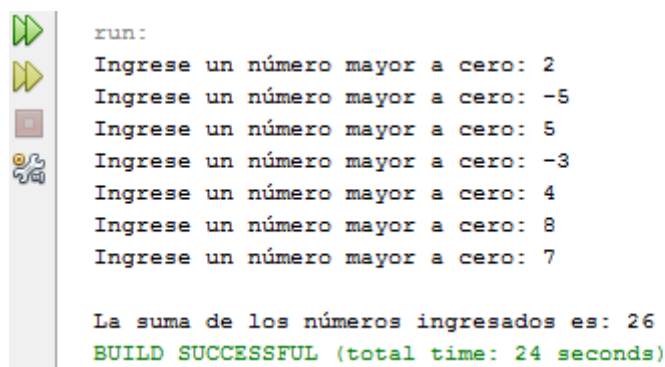
El siguiente ejemplo es una variante del que mostramos antes para cargar números positivos con un *while*, pero haciendo ahora que el ciclo *force la siguiente vuelta* si el valor cargado fue cero o negativo (pero observe que entonces en este caso, el programa *siempre* pedirá los cinco números positivos, incluso si en el medio se cargó alguno negativo o cero):

```

1  import java.util.Scanner;
2
3  public class Principal {
4
5      public static void main(String args[])
6      {
7          int num, suma, i;
8          suma = 0;
9          i = 1;
10         Scanner miEscaner = new Scanner(System.in);
11
12         while (i <= 5)
13         {
14             System.out.print( "Ingrese un número mayor a cero: " );
15             num = miEscaner.nextInt();
16
17             if (num <= 0)
18                 continue;
19
20             suma += num;
21             i++;
22         }
23
24         System.out.println("\nLa suma de los números ingresados es: " + suma);
25     }
26 }
  
```

Al ejecutarse la instrucción *continue*, el ciclo no cortará su ejecución: volverá a la cabecera para volver a chequear la expresión lógica de control, pero no ejecutará en ese caso las instrucciones *suma += n* ni *i++*. Como de esta forma el contador *i* sólo se incrementa si entró un positivo, entonces ese ciclo se detendrá sólo si alguna vez se cargan cinco positivos.

Por ejemplo:



```
run:
Ingrese un número mayor a cero: 2
Ingrese un número mayor a cero: -5
Ingrese un número mayor a cero: 5
Ingrese un número mayor a cero: -3
Ingrese un número mayor a cero: 4
Ingrese un número mayor a cero: 8
Ingrese un número mayor a cero: 7

La suma de los números ingresados es: 26
BUILD SUCCESSFUL (total time: 24 seconds)
```

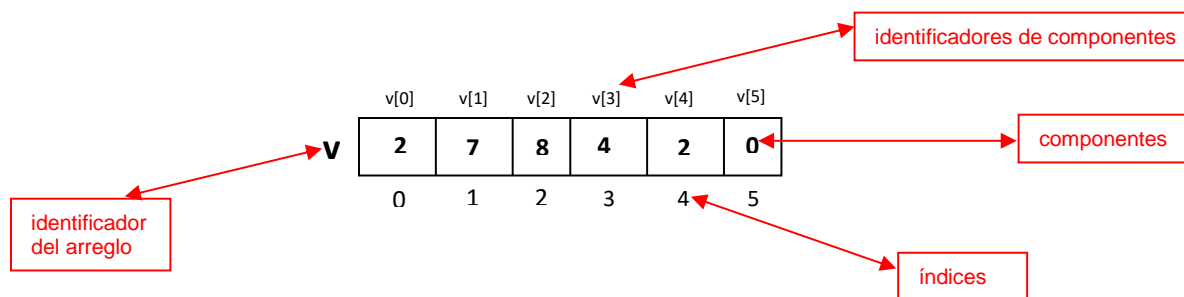
## 7.] Creación y procesamiento de arreglos en Java.

Una estructura de datos muy importante en cualquier lenguaje es la que se conoce como *arreglo*. En términos muy generales, en Java un arreglo es una colección de valores que *deben ser del mismo tipo* y que se organiza de tal forma que cada valor o componente individual es identificado automáticamente por uno o más números designados como índices. El uso de los índices permite el acceso y posterior uso de cada componente en forma individual.

La cantidad de índices que se requieren para acceder a un elemento individual, se llama *dimensión del arreglo*. Los *arreglos unidimensionales* se denominan así porque solo requieren *un índice para acceder a un componente*. Por otra parte, dada la similitud que existe entre el concepto de arreglo unidimensional y el concepto de vector en Álgebra, se suele llamar también *vectores* a los *arreglos unidimensionales*. Como veremos, también podemos definir y crear arreglos bidimensionales, en los que cada elemento se accede con dos índices, y por su parecido con la idea de *matriz* algebraica, se suele hablar de *matrices* para referirse a los *arreglos bidimensionales*. Por supuesto, en Java es posible crear arreglos de cualquier dimensión, siempre y cuando el arreglo creado quepa en la memoria disponible (aunque en la práctica es poco común necesitar arreglos de dimensión mayor a 2).

El siguiente gráfico muestra la forma conceptual de entender un *arreglo unidimensional*. Se supone que la variable arreglo se denomina *v* y que la misma está dividida en seis casilleros, de forma que en cada casillero puede guardarse un valor. Se supone también que el tipo de valor que puede guardarse en el arreglo *v* del ejemplo, es *int*.

Observar que cada casillero es automáticamente numerado con índices, los cuales en Java comienzan siempre a partir del cero: la primera casilla del arreglo siempre es subíndicada con el valor cero, en forma automática. A partir del índice, cada elemento del arreglo *v* puede accederse en forma individual usando el identificador del componente: se escribe el nombre del arreglo, luego un par de corchetes, y entre los corchetes el valor del índice de la casilla que se quiere acceder. En ese sentido, el *identificador* del componente cuyo índice es 2, resulta ser *v[ 2 ]*:



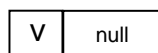
Para declarar una variable de tipo arreglo unidimensional en Java, hay que recordar primero que en Java los arreglos de cualquier dimensión son *objetos*, y por lo tanto deben ser creados con el operador *new*. Lo primero es declarar entonces una *referencia con la cual se va a apuntar al arreglo que se quiere crear*. En Java, esto se hace escribiendo el tipo de valor que se almacenará en el arreglo (ese tipo se conoce como el tipo base del arreglo), luego el nombre del arreglo y finalmente un par de corchetes vacíos:

```
int v[];
```

Note que en Java, el par de corchetes puede ir a la derecha o a la izquierda del nombre de la variable. La declaración anterior es equivalente a esta otra:

```
int []v;
```

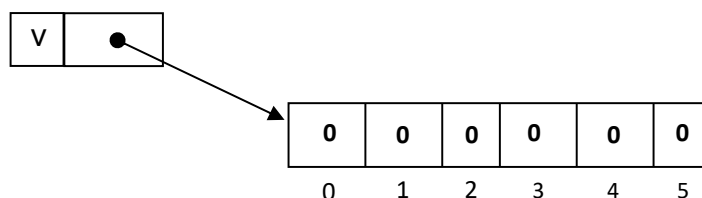
Al declarar una referencia de tipo arreglo, lo que se está haciendo es declarar una variable que luego será capaz de contener la dirección de memoria de un arreglo. El valor inicial de esa referencia es *null*, y el arreglo aún no existe en memoria:



Luego se usa el operador *new* para crear el objeto arreglo: se escribe *new*, seguido nuevamente del tipo base del arreglo, y otra vez el par de corchetes pero de forma que esta vez, se escribe dentro de ellos el tamaño del arreglo que se quiere crear:

```
v = new int [6];
```

La instrucción anterior crea un arreglo de seis componentes capaces de almacenar cada uno un valor *int*, inicializa en cero cada casilla de ese arreglo, y retorna la dirección del mismo (que en este caso se almacena en la referencia *v* que declaramos antes):



Obviamente se puede lograr lo mismo en una sola instrucción:

```
int v[] = new int[6];
```

Observar que si el tamaño de un arreglo es 6, entonces la última casilla del mismo lleva el índice 5 debido a que los índices comienzan siempre desde el 0. En un arreglo en Java, no existe una casilla cuyo índice coincida con el tamaño del arreglo.

Una vez que se creó el arreglo con *new*, se usa la referencia que lo apunta para acceder a sus componentes, colocando a la derecha de ella un par de corchetes y el índice del valor que se quiere acceder. Los siguientes son ejemplos de las operaciones que pueden hacerse con los componentes de un arreglo (tomamos como modelo el arreglo *v* anteriormente creado):

```
7  import java.util.Scanner;
8
9  public class Ejemplo01 {
10
11
12     public static void main(String[] args) {
13         int v[];
14         v = new int[6];
15         Scanner miScanner = new Scanner(System.in);
16         v[3] = 4;
17         v[1]++;
18         System.out.println( v[0] );
19         v[4] = v[1] - v[0];
20         v[5] = miScanner.nextInt();
21         v[2] = v[2] - 8;
22     }
23
24 }
25
```

Si se desea procesar un arreglo de forma que la misma operación se efectúe sobre cada uno de sus componentes, es normal usar un ciclo *for*, de forma se aproveche la variable de control del ciclo como índice para entrar a cada componente. Los siguientes esquemas muestran la forma de hacer una carga por teclado y una visualización por pantalla de un arreglo unidimensional de seis componentes de tipo *int*:

<pre>import java.util.Scanner;  public class Ejemplo02 {     public static void main(String[] args) {         Scanner miScanner = new Scanner(System.in);         int v[];         v = new int[6];         for( int i=0; i&lt;6; i++ )         {             System.out.print("Ingrese v["+i+"]: ");             v[i] = miScanner.nextInt();         }     } }</pre>	<pre>for( int i=0; i&lt;6; i++ ) {     System.out.print(v[i]); }</pre>
--	--

Un detalle interesante es que todo objeto arreglo en Java provee un *atributo* llamado *length*, que contiene el tamaño del arreglo tal como fue declarado al crear ese arreglo con *new*. Ese atributo es de naturaleza pública (*public*), por lo que puede accederse directamente



mediante el identificador de la variable referencia que apunta al arreglo. Los dos ciclos anteriores, podrían escribirse así:

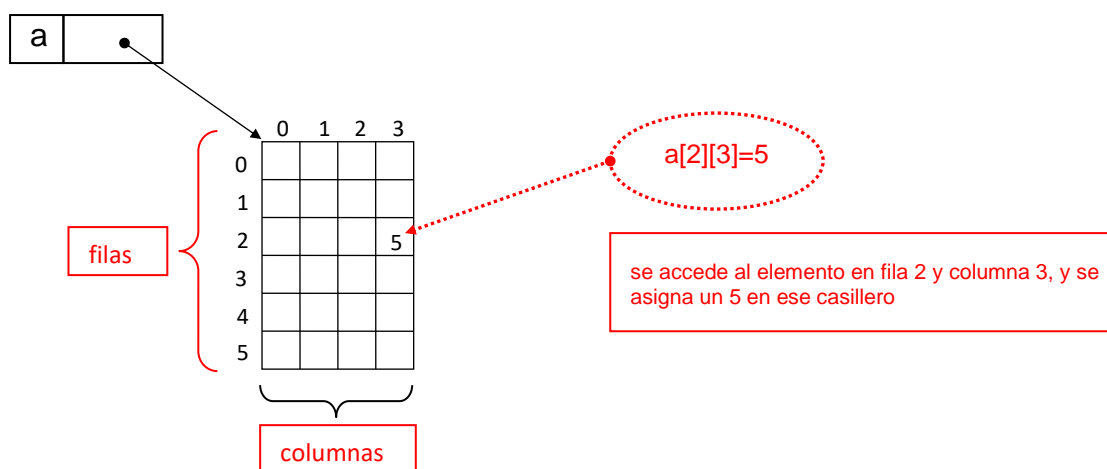
```
int v[];
v = new int[6];
for( int i=0; i<v.length; i++ )
{
    System.out.print("Ingrese v["+i+"]: ");
    v[i] = miScanner.nextInt();
}
```

```
for( int i=0; i<v.length; i++ )
{
    System.out.print(v[i]);
}
```

Por otra parte, un *arreglo bidimensional* o *matriz* es un arreglo cuyos elementos están dispuestos en forma de tabla, con varias filas y columnas. Aquí llamamos *filas* a las disposiciones horizontales del arreglo, y *columnas* a las disposiciones verticales.

Para entrar a un componente, debe darse el índice de la fila del mismo y también el índice de la columna. Como los índices requeridos son dos, el arreglo es de dimensión dos. El siguiente esquema ilustra la manera de declarar y crear un arreglo bidimensional de componentes *int* en Java, la forma conceptual de representarlo, y la manera de acceder a sus componentes:

```
int a[][]; // una referencia al arreglo, con valor inicial null.
a = new int[6][4]; // crea el arreglo, con 6 filas y 4 columnas.
a[2][3] = 5; // accede a una casilla y se asigna un valor en ella.
```



Para declarar la referencia al arreglo se usan ahora *dos pares de corchetes vacíos*. Y para crear el arreglo con *new*, en el primer par de corchetes se escribe la cantidad de filas que se necesitan y en el segundo par se escribe la cantidad de columnas.

Observar que para acceder a un elemento, se coloca el nombre de la referencia al arreglo, luego el número de la *fila* del elemento que se quiere acceder, pero encerrado entre corchetes, y por último el número de la *columna* de ese elemento, también encerrado entre corchetes. Notar además, que en el lenguaje Java los arreglos de cualquier dimensión están *basados en cero*, lo cual significa que el *primer índice de cada dimensión es siempre cero*. En la figura anterior, puede verse que el arreglo tiene seis filas, pero numeradas del 0 (cero) al 5 (cinco), y cuatro columnas, numeradas del 0 (cero) al 3 (tres). No hay excepciones a esta

regla, por lo cual debe tenerse cuidado de ajustar correctamente los ciclos para procesamiento de arreglos.

Para cargar por teclado una matriz  $a$  de  $n$  filas y  $m$  columnas (y en general, *para procesar secuencialmente una matriz*), se pueden usar dos ciclos *for* anidados, de forma que el primero recorra las filas de la matriz, y el segundo las columnas:

La idea básica del proceso aquí definido, es que la variable  $i$  del ciclo más externo se usa para indicar qué *fila* se está procesando en cada vuelta. Dado un valor de  $i$ , se dispara otro ciclo controlado por  $j$ , cuyo objetivo es el de recorrer todas las *columnas* de la fila indicada por  $i$ . Notar que mientras avanza el ciclo controlado por  $j$  permanece fijo el valor de  $i$ . Sólo cuando corta el ciclo en  $j$ , se retorna al ciclo en  $i$ , cambiando ésta de valor y comenzando por ello con una nueva fila. El proceso de recorrer secuencialmente una matriz avanzando fila por fila empezando desde la cero, como aquí se describe, se denomina *recorrido en orden de fila creciente*.

```
int i, j;
System.out.println("Cargue los números del arreglo: ");
for (i = 0; i < n; i++)
    for (j = 0; j < m; j++)
    {
        System.out.print("mat[" + i + "][ " + j + "]:");
        mat[i][j] = miScanner.nextInt();
    }
```

Si se desea un *recorrido en orden de columna creciente (o decreciente)*, sólo deben invertirse los ciclos: el ciclo en  $j$  debe ir por fuera, y el ciclo en  $i$  debe ir por dentro. De esta forma, el valor de  $j$  no cambia hasta que el ciclo en  $i$  termine todo su recorrido. Sin embargo, no debe olvidarse que si queremos que  $j$  indique una columna, entonces  $j$  debe usarse en el *segundo par de corchetes* al acceder a la matriz. Y si la variable  $i$  va a indicar filas, entonces debe usarse en el *primer par de corchetes*. Esto es independiente del orden en que se presenten los ciclos para hacer cada recorrido:

**La variable que indica la *fila* va en el primer par de corchetes, y la variable que indica la *columna* va en el segundo, sin importar cuál ciclo va por fuera y cuál por dentro.**

El siguiente esquema, muestra un recorrido en orden de columna creciente, para leer por teclado una matriz  $a$  de  $n$  filas y  $m$  columnas:

```
int i, j;
System.out.println("Cargue los números del arreglo: ");
for (j = 0; j < m; j++)
    for (i = 0; i < n; i++)
    {
        System.out.print("mat[" + i + "][ " + j + "]:");
        mat[i][j] = miScanner.nextInt();
    }
```

## 8.] Profundización del Principio de Ocultamiento.

Hemos visto que la POO busca identificar *actores* o *entidades* en el dominio de un problema, que sean capaces de ejecutar ciertas acciones. Los lenguajes orientados a objetos usan *clases* para describir la forma y el comportamiento general de estos actores (que se llaman genéricamente *objetos*). Una clase contiene definiciones de *atributos* (o *variables* o también *campos*) que indican la forma de cada objeto. Y una clase también contiene *métodos* (o procesos) que indican el comportamiento que puede desplegar cada objeto de esa clase.

La propiedad de la POO que permite que una clase agrupe atributos y métodos se llama *encapsulamiento*, debido a que mediante esa agrupación una clase se asemeja a una *cápsula de software* que permitirá crear objetos provistos cada uno de todos los elementos de datos y procesos que necesitarán para desenvolverse en un programa.

El *encapsulamiento* da lugar al ya citado *Principio de Ocultamiento*: sólo los métodos de una clase deberían tener acceso directo a los atributos de esa clase, para impedir que un atributo sea modificado en forma insegura, o no controlada por la propia clase. El *Principio de Ocultamiento* es la causa por la cual en general los atributos se declaran como privados (*private*), y los métodos se definen públicos (*public*). Los calificadores *private* y *public* (así como *protected*, que se verá más adelante) tienen efecto a nivel de compilación: si un atributo de una clase es privado, y se intenta acceder a él desde un método de otra clase, se producirá en error de compilación:

```
public class Principal
{
    public static void main ( String args[] )
    {
        Cuenta a = new Cuenta();
        a.numero = 201; // error de compilación: numero es privado
        a.setNumero(201); // ok...
    }
}
```

En el ejemplo anterior, se crea un objeto de la clase *Cuenta* en el método *main()* que está en la clase *Principal*. Si en la clase *Cuenta* el atributo *numero* se declaró privado, el método *main()* no tiene acceso libre a ese atributo en el objeto *a*: la instrucción *a.numero = 201;* no compilará. Note que si el atributo *numero* se hubiera declarado *public* en la clase *Cuenta*, entonces no se produciría el error: un elemento público es accesible desde cualquier método de cualquier clase.

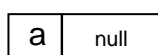
Debe entenderse un detalle: el *Principio de Ocultamiento* es una *convención* y no una *regla*. En general es buena idea hacer que los atributos sólo sean accesibles desde métodos de la misma clase, pero eso no es obligatorio. Pueden darse situaciones en las que el programador decida que un atributo sea público, o que un método sea privado (si esto es así, ese método sólo puede ser invocado por otros métodos de la misma clase y no desde métodos externos a la clase). Sin embargo, y hasta que se logre experiencia y mayores conocimientos en el tema, mantendremos la idea de declarar privados a los atributos y públicos a los métodos.

### 9.] Manejo de referencias. El Garbage Collector.

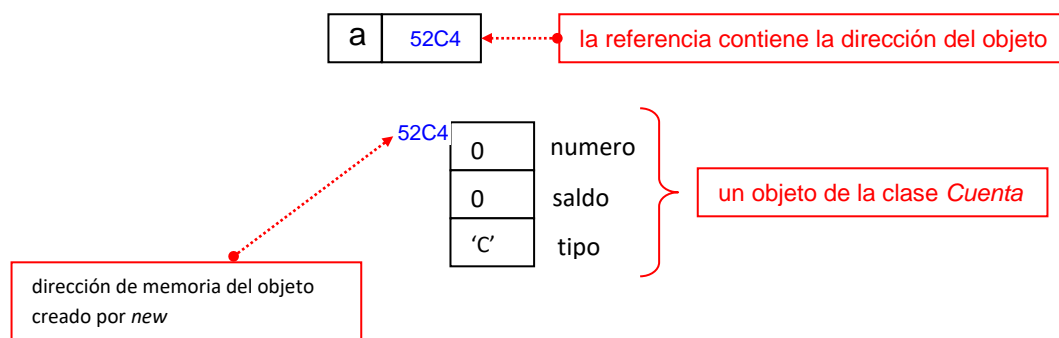
Sabemos que en Java un objeto se crea usando el operador *new*, el cual reserva memoria para el objeto creado, invoca a un constructor de la clase para iniciar sus atributos, y finalmente *retorna la dirección de memoria del objeto creado*. Esa dirección se almacena en una variable que recibe el nombre genérico de *variable referencia*:

```
Cuenta a;           // declara una variable referencia...
a = new Cuenta( );  // crea un objeto y guarda su dirección en la referencia
```

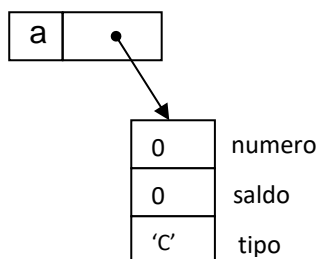
En el fragmento anterior, la primera línea declara una variable *a* de tipo *Cuenta*. Esa variable *no* es un objeto de la clase *Cuenta*, sino que puede contener la *dirección* de un objeto de la clase *Cuenta*. Hasta que ese objeto se cree, podemos asumir en general que la variable *a* contiene lo que se conoce como la *dirección nula* (*null*):



La segunda línea es la que usando *new* crea un objeto de la clase *Cuenta*, y lo inicializa invocando al constructor sin parámetros de la clase. Si suponemos que el objeto quedó guardado en memoria en la dirección *52C4*, entonces *new* retornaría ese valor y lo asignaría en la referencia *a*, quedando así la memoria:



Como se ve, una cosa es la referencia y otra cosa es el objeto creado con *new*. Sin embargo, una vez que la referencia contiene la dirección del objeto, se usa esa referencia para manejar al objeto (se dice que la referencia es un manejador del objeto). En un gráfico es común que en lugar de mostrar el número que representa la dirección, se muestre simplemente una flecha indicando que la referencia apunta al objeto (y esto quiere decir lo que ya dijimos: la referencia contiene la dirección del objeto). El gráfico siguiente equivale conceptualmente al anterior:



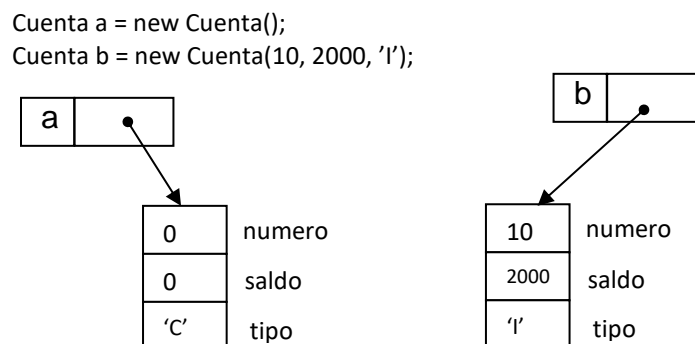
Cuando decimos que una referencia que apunta a un objeto se usa como un manejador para ese objeto, queremos decir que a partir de allí la referencia se usa como si ella fuera el objeto, y a través de ella se invocan los métodos públicos del objeto (mediante el operador punto colocado entre la referencia y el nombre del método):

```
a.setNumero(200);
a.setSaldo(3000);
a.setTipo('C');
```

Notar que si una variable referencia vale *null*, la misma *no puede* usarse para acceder a los miembros del objeto, simplemente porque el objeto *no existe*. El intento de hacerlo provocará una excepción de puntero nulo (*NullPointerException*) al ejecutar el programa, y el programa se interrumpirá:

```
Cuenta a = null;
a.setNumero(2034); // NullPointerException y el programa se interrumpe
```

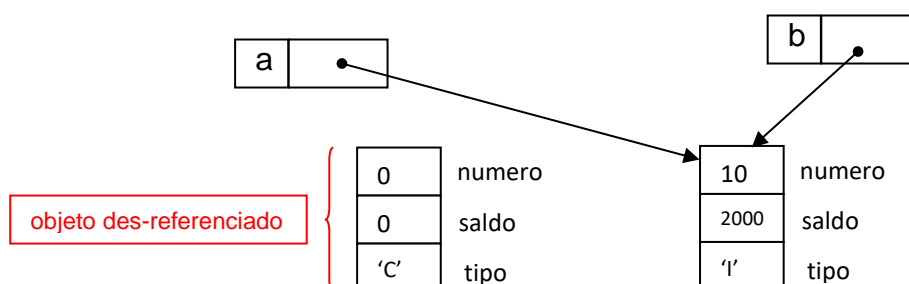
A partir de aquí, el programador debe tener cuidado de entender bien lo que hace cuando opera con referencias: supongamos que dos referencias *a* y *b* apuntan a dos objetos diferentes, creados ambos con *new*. El gráfico de memoria se vería así:



Si luego se hace una asignación como la siguiente:

```
a = b;
```

entonces ambas referencias pasarán a apuntar al mismo objeto (el que originalmente era apuntado por *b*). Por lo tanto, si se invoca un método desde la referencia *a*, tal como *a.setNumero(20)* para cambiar el contenido de algún atributo de *a*, se estará modificando también el objeto referido por *b*. La asignación *a=b*; NO COPIA los contenidos del objeto apuntado por *b* hacia el objeto apuntado por *a*. Lo que copia es la dirección contenida en *b* en la variable *a*:



El objeto apuntado originalmente por *a* queda des-referenciado. En Java eso no es necesariamente un error (como lo sería en C++), pues en Java existe un sistema de *recolección de residuos de memoria* que se encarga de chequear la memoria en busca de objetos "perdidos" en tiempo de ejecución. Cuando un programa se ejecuta, en otro hilo de ejecución paralelo corre también un proceso llamado *garbage collector*, que se encarga de los objetos des-referenciados.

Técnicamente, el *garbage collector* liberará la memoria ocupada por esos objetos cuando determine que se está próximo a una situación de *out of memory* (una situación en la que la memoria está a punto de agotarse). De otro modo, aun cuando existan objetos perdidos, el *garbage collector* posiblemente no llegue a activarse. El programador puede confiar en que el *garbage collector* no afectará el rendimiento de su aplicación, pero también debe saber que no podrá intervenir en el esquema de trabajo del mismo. A los sumo, el programador puede requerir que el *garbage collector* considere intervenir y revisar la memoria en un momento dado, invocando al método *System.gc()*. Pero incluso en este caso, si el *garbage collector* determina que no hay peligro de *out of memory*, no liberará la memoria ocupada por objetos perdidos:

```
Cuenta a, b, c;

// creamos tres objetos...
a = new Cuenta();
b = new Cuenta(10, 2000, 'C');
c = new Cuenta(20);

// des-referenciamos los objetos manejados por a y b...
a = null;
b = null;

// pedimos intervención al garbage collector...
// ... los objetos des-referenciados podrían ser eliminados ahora
// ... pero eso lo decide el garbage collector!!!
System.gc();
```

## 10.] Ejemplo de aplicación simple.

Mostraremos ahora el desarrollo de un ejercicio en base a la POO en Java. Comenzaremos con un modelo simple, cuyo enunciado es el siguiente:

*Se conocen dos números distintos. Calcular la superficie de un cuadrado, suponiendo como lado del mismo al mayor de los números dados y la superficie de un círculo suponiendo como radio del mismo al menor de los números dados.*

La lógica de la solución es directa: en lugar de identificar los *procesos principales* (que eran los subproblemas de buscar el mayor y el menor; y de calcular las áreas) como se hace normalmente en *Programación Estructurada*, lo que buscamos en POO es identificar los *objetos que deberían hacerse cargo del trabajo*, e intentar describir en forma amplia las clases de esos objetos.

En un primer análisis, lo que hacemos es buscar *sustantivos* en el enunciado, como candidatos a ser las clases del problema. Podemos ver rápidamente unos cuantos: *cuadrado, círculo, lado, radio, superficie...* Sin embargo, también podemos ver que algunos sustantivos

representan en realidad *atributos* de las clases que realmente nos importan: *lado* y *superficie* son *atributos* de un *cuadrado*, y *radio* y *superficie* son *atributos* de un *círculo*... En general, el programador deberá esforzarse por notar la diferencia entre sustantivos que serán claramente *clases* en el modelo final, y sustantivos que serán características o propiedades de esas clases (*atributos*...)

Proponemos el siguiente modelo simple. Contaremos con dos clases *Cuadrado* y *Circulo*, cada una de las cuales calculará el área de esas figuras. La primera tendrá un atributo *lado* y la segunda un atributo *radio*. La clase *Principal* contendrá el *main()*, y se encargará de la responsabilidad de determinar cuál de los dos números cargados es el mayor y cuál es el menor. Observe que el método *toString()* de cada una de las clases *Cuadrado* y *Circulo* invoca al método *superficie()* de cada clase, y retorna una cadena que contiene ese valor (por ese motivo, *main()* no tiene necesidad de pedirle al objeto cuadrado y al objeto circulo que calculen la superficie: al mostrar los datos de ambos, aparecerán las superficies)

El esquema propuesto podría basarse en otras clases adicionales: por ejemplo, se podría contar con una clase *Ordenador* que se encargue de representar dos números y ordenarlos de menor a mayor (el método *ordenar()* podría estar en esta clase en lugar de estar en la clase *Principal*). Por el momento nuestra solución quedará como se propuso inicialmente y el programa completo quedaría así:

```
public class Circulo
{
    private float radio;

    public Circulo()
    {
        radio = 0;
    }

    public Circulo(float r)
    {
        radio = r;
    }

    public float getRadio()
    {
        return radio;
    }

    public void setRadio(float r)
    {
        radio = r;
    }

    public float superficie()
    {
        return 3.14f * radio * radio;
    }

    public String toString()
    {
        return "Circulo - Radio: " + radio + "\tArea: " + superficie();
    }
}
```

```

7   public class Cuadrado
8   {
9       private float lado;
10
11      public Cuadrado()
12      {
13          lado = 0;
14      }
15
16      public Cuadrado(float lad)
17      {
18          lado = lad;
19      }
20
21      public float getLado()
22      {
23          return lado;
24      }
25
26      public void setLado(float lad)
27      {
28          lado = lad;
29      }
30
31      public float superficie()
32      {
33          return lado * lado;
34      }
35
36      public String toString()
37      {
38          return "Cuadrado - Lado: " + lado + "\tArea: " + superficie();
39      }

```

```

6   import java.util.Scanner;
7
8   public class Principal
9   {
10      private static float a, b, men, may;
11
12      public static void main (String args[])
13      {
14          Scanner miScanner = new Scanner(System.in);
15          System.out.print("Primer número: ");
16          a = miScanner.nextFloat();
17          System.out.print("Segundo número: ");
18          b = miScanner.nextFloat();
19
20          ordenar();
21          Cuadrado cuad = new Cuadrado(may);
22          Circulo circ = new Circulo (men);
23
24          System.out.println(cuad.toString());
25          System.out.print(circ.toString());
26      }
27

```



```

28      public static void ordenar()
29      {
30          if ( a < b )
31          {
32              may = a;
33              men = b;
34          }
35          else
36          {
37              may = b;
38              men = a;
39          }
40      }
41  }

```

Ahora bien: ¿cuál es la ventaja de semejante forma de trabajo? En este modelo de solución hemos tenido que programar considerablemente más líneas de código que en las soluciones básicas que se producirían con el paradigma de la programación estructurada... y eso sin considerar otras posibles clases que podrían haberse usado (como se sugirió con la clase *Ordenador*...). El programa hace básicamente lo mismo que podría haber hecho con el modelo de programación estructurada, pero ahora aparecen (*sólo*...) dos o cuatro clases, y muchísimos métodos que *a priori* el simple enunciado del problema no pedía...

La POO tiene un objetivo a horizonte más largo que la programación estructurada... Recordemos que la POO aparece como una forma de aportar claridad y mayor control en *sistemas verdaderamente grandes*, brindando herramientas y principios que favorecen la re-usabilidad de código o el planteo de procesos genéricos... Puede que ahora este simple ejemplo no muestre la potencia del paradigma, pero a medida que el estudiante avanza, notará que su productividad como programador aumenta, cada vez más, hasta que en algún momento podrá llegar a plantear programas en los que simplemente deberá combinar objetos de clases que ya tenía preparadas en otros sistemas, o provistas por el propio Java... y entonces entenderá.

### 11.] Caso de análisis: técnicas de ordenamiento implementadas en Java.

Cerraremos esta ficha de estudios mostrando una clase *Vector* muy general que implementa y encapsula el concepto de arreglo unidimensional (o vector) e incorpora entre sus métodos básicos a todos los métodos de ordenamiento clásicos que se han presentado en materias anteriores.

La clase *Vector* simplemente declara como único atributo una referencia *v* para apuntar a un arreglo unidimensional de números enteros. La misma clase incluye un constructor sencillo, que toma como parámetro el tamaño *n* del arreglo y crea el arreglo con *new*. Además, incluye un par de métodos de acceso llamados *get(i)* (para retornar el valor contenido en la casilla *i* del arreglo) y *set(i, x)* para asignar el valor *x* en la casilla *i*:

```

public class Vector
{
    private int v[];

    public Vector(int n)
    {
        if(n <= 0) { n = 100; }
    }
}

```

```
        v = new int[n];
    }

    public int get(int i)
    {
        return v[i];
    }

    public void set(int i, int x)
    {
        v[i] = x;
    }

    // resto del contenido de la clase aquí...
}
```

La clase incluye también un par de métodos muy útiles si se trabaja con pruebas sobre técnicas de ordenamiento: uno de esos métodos se llama *generate()* y simplemente llena el arreglo con *n* números enteros aleatorios, siendo *n* el tamaño del arreglo. El otro método se llama *checkOrder()* y sólo comprueba si el arreglo está ordenado (retornando *true* en ese caso) o no (en cuyo caso retorna *false*):

```
public void generate ()
{
    for( int i=0; i<v.length; i++ )
    {
        v[i] = (int) Math.round(100 * Math.random());
    }
}

public boolean checkOrder ()
{
    for ( int i=0; i < v.length - 1; i++)
    {
        if (v[i] > v[i+1]) { return false; }
    }
    return true;
}
```

Los dos métodos que siguen implementan las ya conocidas técnicas de búsqueda secuencial (método *linearSearch()*) y búsqueda binaria (método *binarySearch()*):

```
public int linearSearch(int x)
{
    for(int i=0; i<v.length; i++)
    {
        if(x == v[i]) return i;
    }
    return -1;
}

public int binarySearch(int x)
{
    int n = v.length;
    int izq = 0, der = n-1;
    while(izq <= der)
    {
        int c = (izq + der)/2;
        if(x == v[c]) return c;
    }
}
```

```

        if(x < v[c]) der = c - 1;
        else izq = c + 1;
    }
    return -1;
}

```

Y finalmente aparece un amplio conjunto de métodos que implementan los métodos clásicos de ordenamiento. Los primeros tres de ese grupo corresponden a los *algoritmos simples o directos* (que conocemos como *ordenamiento de burbuja o de intercambio directo*, *ordenamiento por selección directa* y *ordenamiento por inserción directa*):

```

public void bubbleSort()
{
    boolean ordenado = false;
    int n = v.length;
    for( int i=0; i<n-1 && !ordenado; i++ )
    {
        ordenado = true;
        for( int j=0; j<n-i-1; j++ )
        {
            if( v[j] > v[j+1] )
            {
                int aux = v[j];
                v[j] = v[j+1];
                v[j+1] = aux;
                ordenado = false;
            }
        }
    }
}

public void selectionSort()
{
    int n = v.length;
    for(int i = 0; i < n - 1; i++ )
    {
        for( int j = i + 1; j < n; j++ )
        {
            if( v[i] > v[j] )
            {
                int aux = v[i];
                v[i] = v[j];
                v[j] = aux;
            }
        }
    }
}

public void insertionSort()
{
    int n = v.length;
    for( int j = 1; j < n; j++ )
    {
        int k, y = v[j];
        for( k = j-1; k >= 0 && y < v[k]; k-- )
        {
            v[k+1] = v[k];
        }
        v[k+1] = y;
    }
}

```

```
}
```

Los métodos que restan (casi todos públicos y unos pocos privados que sirven como auxiliares de algunos de los públicos) implementan los métodos clásicos de *ordenamiento compuesto*. Los dos primeros (público el primero, privado el segundo) corresponden al algoritmo *Quick Sort*: el método público *quickSort()* es el que se invoca desde el exterior de la clase para lanzar el proceso, y el método privado y recursivo *quick()* es el que realmente hace todo el trabajo:

```
public void quickSort()
{
    quick(0, v.length - 1);
}

private void quick(int izq, int der)
{
    int i = izq, j = der, y;
    int x = v[(izq + der) / 2];
    do
    {
        while( v[i] < x && i < der ) { i++; }
        while( x < v[j] && j > izq ) { j--; }
        if( i <= j )
        {
            y = v[i];
            v[i] = v[j];
            v[j] = y;
            i++;
            j--;
        }
    }
    while( i <= j );
    if( izq < j ) { quick( izq, j ); }
    if( i < der ) { quick( i, der ); }
}
```

Los dos que siguen (*heapSort()* y *shellSort()*) corresponden a los algoritmo *Heap Sort* y *Shell Sort*:

```
public void heapSort()
{
    int n = v.length;

    // crear el grupo inicial...
    for( int i = 1; i < n; i++ )
    {
        int e = v[i];
        int s = i;
        int f = (s-1)/2;
        while( s>0 && v[f] < e )
        {
            v[s] = v[f];
            s = f;
            f = (s-1)/2;
        }
        v[s] = e;
    }

    // extraer la raiz, y reordenar el vector y el grupo...
```

```

for(int i = n-1; i>0; i-- )
{
    int valori = v[i];
    v[i] = v[0];
    int f = 0, s;
    if( i == 1 ) { s = -1; } else { s = 1; }
    if( i > 2 && v[2] > v[1] ) { s = 2; }
    while( s >= 0 && valori < v[s] )
    {
        v[f] = v[s];
        f = s;
        s = 2*f + 1;
        if( s + 1 <= i - 1 && v[s] < v[s+1] ) { s++; }
        if( s > i - 1 ) { s = -1; }
    }
    v[f] = valori;
}
}

```

Por fin, los tres métodos finales implementan el algoritmo *Merge Sort*. El primero de los tres (*mergeSort()*) es público y es el que se invoca desde fuera de la clase para lanzar el proceso. El segundo método (*sort()*) es privado y recursivo y se encarga de dividir en dos al arreglo en cada entrada recursiva, invocando en cada una de esas entradas al tercer método (*merge()*), también privado, que hace la fusión entre las mitades ordenadas (los detalles lógicos de este algoritmo, así del Heap Sort y un repaso del Quick Sort, serán oportunamente analizados):

```

public void mergeSort()
{
    int n = v.length;
    int temp[] = new int[n];
    sort(0, n-1, temp);
}

private void sort(int izq, int der, int temp[])
{
    if(izq < der)
    {
        int centro = (izq + der) / 2;
        sort(izq, centro, temp);
        sort(centro + 1, der, temp);
        merge(izq, centro, der, temp);
    }
}

private void merge(int izq, int centro, int der, int temp[])
{
    for(int i = izq; i <= der; i++) { temp[i] = v[i]; }

    int i = izq, j = centro + 1, k = izq;
    while(i <= centro && j <= der)
    {
        if(temp[i] <= temp[j])
        {
            v[k] = temp[i];
            i++;
        }
        else
        {
            v[k] = temp[j];
            j++;
        }
        k++;
    }
}

```

```

        j++;
    }
    k++;
}

while(i <= centro)
{
    v[k] = temp[i];
    k++;
    i++;
}
}

```

El proyecto *TSB [Ordenamiento]* que acompaña a esta Ficha incluye la clase *Vector* que acabamos de describir, y una clase *Prueba* que contiene un método *main()* para poner a funcionar todo a través de un menú de opciones:

```

package modelo;
import java.util.Scanner;

public class Prueba
{
    public static void main (String[] args)
    {
        long t1, t2, tf;
        int op;

        Scanner sc = new Scanner(System.in);

        System.out.print ("Ingrese cantidad del elementos del vector: ");
        int n = sc.nextInt();
        Vector v = new Vector(n);
        do
        {
            System.out.println ("\nOpciones de Ordenamiento");
            System.out.println ("0. Generar el Arreglo");
            System.out.println ("1. Intercambio Directo (Burbuja)");
            System.out.println ("2. Seleccion Directa");
            System.out.println ("3. Insercion Directa");
            System.out.println ("4. Quick Sort");
            System.out.println ("5. Heap Sort");
            System.out.println ("6. Shell Sort");
            System.out.println ("7. Merge Sort");
            System.out.println ("8. Verificar si esta ordenado");
            System.out.println ("9. Salir");
            System.out.print ("Ingrese opcion: ");
            op = sc.nextInt();
            switch (op)
            {
                case 0:
                    System.out.print("Se vuelve a generar el vector...");
                    v.generate();
                    System.out.print("\nVector generado...");
                    break;

                case 1:
                    System.out.print("Se ordena por Intercambio...");
                    t1 = System.currentTimeMillis();
                    v.bubbleSort();
                    t2 = System.currentTimeMillis();
                    tf = t2 - t1;

```

```
        System.out.print("\nTiempo: " + tf + " milisegundos...");
        break;

    case 2:
        System.out.print("Se ordena por Seleccion...");
        t1 = System.currentTimeMillis();
        v.selectionSort();
        t2 = System.currentTimeMillis();
        tf = t2 - t1;
        System.out.print("\nTiempo: " + tf + " milisegundos...");
        break;

    case 3:
        System.out.print("Se ordena por Insercion...");
        t1 = System.currentTimeMillis();
        v.insertionSort();
        t2 = System.currentTimeMillis();
        tf = t2 - t1;
        System.out.print("\nTiempo: " + tf + " milisegundos...");
        break;

    case 4:
        System.out.print("Se ordena por Quick sort...");
        t1 = System.currentTimeMillis();
        v.quick();
        t2 = System.currentTimeMillis();
        tf = t2 - t1;
        System.out.print("\nTiempo: " + tf + " milisegundos...");
        break;

    case 5:
        System.out.print("Se ordena por Heap Sort...");
        t1 = System.currentTimeMillis();
        v.heapSort();
        t2 = System.currentTimeMillis();
        tf = t2 - t1;
        System.out.print("\nTiempo: " + tf + " milisegundos...");
        break;

    case 6:
        System.out.print("Se ordena por Shell sort...");
        t1 = System.currentTimeMillis();
        v.shellSort();
        t2 = System.currentTimeMillis();
        tf = t2 - t1;
        System.out.print("\nTiempo: " + tf + " milisegundos...");
        break;

    case 7:
        System.out.print("Se ordena por Merge Sort...");
        t1 = System.currentTimeMillis();
        v.mergeSort();
        t2 = System.currentTimeMillis();
        tf = t2 - t1;
        System.out.print("\nTiempo: " + tf + " milisegundos...");
        break;

    case 8:
        System.out.println("Se verifica si esta ordenado...");
        if(v.checkOrder())
```

```
        {
            System.out.println("Esta ordenado...");
        }
        else
        {
            System.out.println ("No esta ordenado...");
        }
        break;

        case 9: ;
    }
}
while (op != 9);
}
```

Note que la clase *Prueba* se ha pensado como una simple clase para incluir un *main()* y un menú de opciones para probar el contenido de la clase *Vector*. Podría pensarse que la clase *Vector* no es necesaria: todo el contenido de esa clase podría haber sido incluido en la misma clase *Prueba* (o al revés: el método *main()* podría haber sido incluido en la clase *Vector*) y con eso tendríamos una sola clase, dando la impresión de hacer todo más sencillo...

Sin embargo, el hecho de haber diseñado la clase *Vector* por separado, tomando incluso ciertos recaudos para hacerla genérica, sin dependencias externas ni acoplamiento entre procesos e interfaz de usuario, deja a esa clase lista para ser reutilizada en cuando proyecto sea requerida, sin tener que contar con un "molesto" método *main()* cuyo objetivo era solamente proporcionar un marco de testing. Las responsabilidades funcionales de la clase *Vector* (contener un arreglo unidimensional y proporcionar algoritmos básicos de ordenamiento y búsqueda para ese arreglo) son claramente diferentes de las responsabilidades de la clase *Prueba* (ofrecer un contexto simple de testing para la clase *Vector*).

Y esa es una línea de diseño muy fuerte en POO: trate de diseñar clases diferentes para responsabilidades diferentes y bien definidas, aún cuando finalmente le queden clases muy sencillas o breves. Mientras más especializada está una clase para cubrir sus responsabilidades, más reusable será esa clase y por cierto, más sencilla para mantener y modificar.

A lo largo del curso veremos distintas formas de replantear la misma clase, aplicando elementos de control de errores, testing automático, patrones de diseño, control de proyectos y diversas convenciones de trabajo de la POO que por ahora escapan al espíritu introductorio simple de esta ficha. Tenga paciencia...