EMPEZANDO A PROGRAMAR CON JAVA™

CARLOS E. CIMINO

"Si lo puedes imaginar, lo puedes programar"

Índice de contenido

Introducción....5
Primer Programa....6
Comentarios....7
Salida por consola....8
Escribir en la consola....8
Instrucción print....8
Instrucción println....8
Caracteres de escape....9
Definición....9
Listado de caracteres de escape....10
Operadores....11
Tipos....11
Asignación....11
Aritméticos....12
Relacionales....12
Lógicos....13
Incrementales....13
Concatenación....13
Jerarquía....14
Paréntesis....15
Variables....16
Definición....16
Tipos de datos primitivos....16
Conversiones de tipo....17
Tipos de dato por referencia....19
Nombres de variables....19
Declaración de variables....20
Sin inicializar....20
Inicializadas....21
Entrada por consola....22
La clase Scanner....22
Importar la clase....22
Generar el objeto....22
Leer datos....22
Tipos numéricos....22
Tipos alfanuméricos....23
Estructuras de control....24
Estructura secuencial....24
Estructuras de selección....24
IF....25
IF...ELSE....25
IF...ELSE anidados....26
SWITCH....27
Estructuras de repetición....28
WHILE....29
DO-WHILE....30
FOR....31
Arreglos....32
Definición....32
Declaración....32
Sin inicializar....32
Inicializado por defecto....32
Inicializado con valores....33
Obtener la longitud de un arreglo....34
Obtener un valor de un arreglo....34
Reemplazar un valor de un arreglo....35

Mostrar todos los valores de un arreglo....36
Ordenamiento de arreglos....37
Ordenamiento por burbujeo (Bubble Sort)....38
Ordenamiento por inserción (Insertion Sort)....38
Ordenamiento por selección (Selection Sort)....40
Ordenamiento Shell (Shell Sort)....41
Búsqueda en arreglos....42
Búsqueda secuencial....42
Búsqueda binaria....43
Arreglos multidimensionales....44
Definición....44
Declaración....44
Sin inicializar....44
Inicializada por defecto....45
Inicializada con valores....46
Conocer la longitud de una matriz....46
Obtener un valor de una matriz....47
Reemplazar un valor de una matriz....48
Mostrar todos los valores de una matriz....49
Utilidades....50
Tabla de caracteres ASCII imprimibles....50
Bibliografía y referencias....51

Introducción

Java es un lenguaje de programación de propósito general.

Su intención es permitir que los desarrolladores de aplicaciones escriban el programa una vez y lo ejecuten en cualquier dispositivo (conocido en inglés como WORA, o "write once, run anywhere"), lo que quiere decir que el código que es ejecutado en una plataforma no tiene que ser recompilado para correr en otra.

A partir de 2012, es uno de los lenguajes de programación más populares en uso, particularmente para aplicaciones de cliente-servidor de web, con unos 10 millones de usuarios reportados.

Fue originalmente desarrollado por James Gosling de Sun Microsystems (la cual fue adquirida por la compañía Oracle) y publicado en 1995 como un componente fundamental de la plataforma Java de Sun Microsystems.

Su sintaxis deriva en gran medida de C y C++, pero tiene menos utilidades de bajo nivel que cualquiera de ellos.

El lenguaje Java se creó con cinco objetivos principales:

- Debería usar el paradigma de la programación orientada a objetos.
- Debería permitir la ejecución de un mismo programa en múltiples sistemas operativos.
- Debería incluir por defecto soporte para trabajo en red.
- Debería diseñarse para ejecutar código en sistemas remotos de forma segura.
- Debería ser fácil de usar y tomar lo mejor de otros lenguajes orientados a objetos, como C++.

Primer Programa

Todo programa en Java tiene como punto de entrada la función **main** (en inglés, "**principal**"), que pertenece a la clase que el usuario defina como nombre de la misma (en este caso, "**JavaApplication1**").

```
public class JavaApplication1 {
    public static void main(String[] args) {
        // AQUI VA NUESTRO CODIGO
    }
}
```

Código 1: Programa vacío

No nos vamos a detener a analizar conceptos como "función" o "clase", ya que los mismos serán tratados posteriormente.

Por ahora, solo nos importa saber que todo nuestro código estará dentro de esta función llamada `main`, delimitada entre dos llaves `{}`.

Comentarios

Los programadores insertan comentarios para documentar los programas y mejorar su legibilidad. Los comentarios también ayudan a otras personas a leer y comprender un programa. El compilador de Java ignora estos comentarios, de manera que la computadora no hace nada cuando el programa se ejecuta.

```
public class JavaApplication1 {
    public static void main(String[] args) {
        // COMENTARIO DE LINEA

        suma = a + b; // A PARTIR DE AQUI ES UN COMENTARIO

        /* Comentario de bloque:
                permite agrupar más de
                una línea para no anteponer
                la // por cada una de ellas */
    }
}
```

Código 2: Tipos de comentarios

Un comentario que comienza con `//` se llama comentario de fin de línea (o de una sola línea), ya que termina al final de la línea en la que aparece. Un comentario que se especifica con `//` puede empezar también en medio de una línea, y continuar solamente hasta el final de esa línea.

Los comentarios tradicionales (también conocidos como comentarios de múltiples líneas), se distribuyen en varias líneas. Este tipo de comentario comienza con el delimitador `/*` y termina con `*/`. El compilador ignora todo el texto que esté entre los delimitadores.

Los programadores usan líneas en blanco y espacios para facilitar la lectura de los programas. En conjunto, las líneas en blanco, los espacios y las tabulaciones se conocen como espacio en blanco. (Los espacios y tabulaciones se conocen específicamente como caracteres de espacio en blanco). El compilador ignora todo espacio en blanco.

Salida por consola

Escribir en la consola

Instrucción print

La instrucción que permite generar una salida hacia la consola es:

`System.out.print();`

Entre los paréntesis `()` irá el contenido que se desea mostrar (una cadena, una expresión, el contenido de una variable, etc.).

La instrucción mencionada imprime por pantalla y **deja el cursor posicionado en el último caracter mostrado**. El contenido de la próxima salida por consola que hubiere sería visualizado en la misma línea.

```
public class JavaApplication1 {
    public static void main(String[] args) {
        System.out.print("Hola Mundo");
        System.out.print("Otra línea");
    }
}
```

Código 3: Imprimir por consola con print

```
run:
Hola MundoOtra líneaBUILD SUCCESSFUL (total time: 0 seconds)
```

Salida 1: Imprimir por consola con print

La cadena "**BUILD SUCCESSFUL (total time: 0 seconds)**" la imprime la consola automáticamente indicando que el programa terminó su ejecución correctamente. Debido a lo explicado anteriormente, queda en la misma línea que nuestro "Hola mundo".

Instrucción println

La instrucción que permite generar una salida con salto de línea hacia la consola es:

`System.out.println();`

Entre los paréntesis `()` irá el contenido que se desea mostrar (una cadena, una expresión, el contenido de una variable, etc.)

La instrucción mencionada imprime por pantalla y **deja el cursor posicionado en la próxima línea.**

Si para nuestro "`Hola Mundo`" hubiéramos utilizado instrucciones `println`, nuestra salida sería:

```
public class JavaApplication1 {
    public static void main(String[] args) {
        System.out.println("Hola Mundo");
        System.out.println("Otra línea");
    }
}
```

Código 4: Imprimir por consola con println

```
run:
Hola Mundo
Otra línea
BUILD SUCCESSFUL (total time: 0 seconds)
```

Salida 2: Imprimir por consola con println

Caracteres de escape

Definición

Indican a los métodos `print` y `println` de `System.out` que se va a imprimir un "carácter especial".

Cuando aparece una barra diagonal inversa `\` en una cadena de caracteres, Java la combina junto al siguiente carácter para formar una secuencia de escape.

Por ejemplo, la secuencia de escape `\n` representa el carácter de nueva línea.

Cuando aparece un carácter de nueva línea en una cadena que se va a imprimir con `System.out`, el carácter de nueva línea hace que el cursor de salida de la pantalla se desplace al inicio de la siguiente línea en la ventana de comandos.

```
public class JavaApplication1 {
    public static void main(String[] args) {
        System.out.println("Tres\nlíneas\ndiferentes");
    }
}
```

Código 5: Caracteres de escape

```
run:
Tres
líneas
diferentes
BUILD SUCCESSFUL (total time: 0 seconds)
```

Salida 3: Caracteres de escape

Listado de caracteres de escape

Secuencia de escape	Descripción
`\n`	Nueva línea. Coloca el cursor de la pantalla al inicio de la siguiente línea.
`\t`	Tabulador horizontal. Desplaza el cursor de la pantalla hasta la siguiente posición de tabulación.
`\r`	Retorno de carro. Coloca el cursor de la pantalla al inicio de la línea actual; no avanza a la siguiente línea. Cualquier carácter que se imprima después del retorno de carro sobrescribe los caracteres previamente impresos en esa línea.
`\b`	Retroceso. Borra el último carácter mostrado.
`\\`	Barra diagonal inversa. Se usa para imprimir un carácter de barra diagonal inversa.
`\”`	Doble comilla. Se usa para imprimir un carácter de doble comilla.
`\’`	Comilla simple. Se usa para imprimir un carácter de comilla simple.

Operadores

Tipos

Analizaremos algunos de los operadores presentes en el lenguaje Java y los clasificaremos por tipo.

Asignación

Son operadores que se utilizan para asignar valores a variables. Tienen como particularidad que se leen de derecha a izquierda. **"Lo que está a la derecha se asigna a lo que está a la izquierda"**.

El operador más sencillo es el `=`.

Hay operadores especiales como el `+=`, `-=`, `*=`, `/=` y el `%=` que permiten hacer operaciones con el valor de la variable y guardar el resultado en sí misma.

Como ejemplo, la expresión `a += 4` podría escribirse de la forma `a = a + 4`. Este mismo criterio se cumple para los otros cuatro operadores siguientes.

Son muy útiles para hacer acumuladores.

Supongamos que tenemos una variable llamada x cuyo valor es 9.

Operador	Nombre	Ejemplo	Nuevo valor de x	Descripción
`=`	Asignación	x = 3	3	Asigna a la variable "x" la expresión que está a su derecha.
`+=`	Suma y asignación	x += 3	12	Suma el valor actual de la variable "x" con la expresión a la derecha y lo reasigna en "x". (Lo anterior se pierde).
`-=`	Resta y asignación	x -= 3	6	Resta el valor actual de la variable "x" con la expresión a la derecha y lo reasigna en "x". (Lo anterior se pierde).
`*=`	Multiplicacion y asignación	x *= 3	27	Multiplica el valor actual de la variable "x" con la expresión a la derecha y lo reasigna en "x". (Lo anterior se pierde).
`/=`	División y asignación	x /= 3	3	Divide el valor actual de la variable "x" con la expresión a la derecha y lo reasigna en "x". (Lo anterior se pierde).
`%=`	Módulo y asignación	x %= 3	0	Calcula el resto del cociente entre el valor actual de la variable "x" con la expresión a la derecha y lo reasigna en "x". (Lo anterior se pierde).

Aritméticos

Son los operadores fundamentales de la aritmética con el agregado del `%`, que devuelve el resto entero que se produce al realizar un cociente entre dos expresiones numéricas.

Operador	Nombre	Ejemplo	Resultado	Descripción
`+`	Suma	12 + 3	15	Devuelve la suma de dos expresiones.
`-`	Resta	12 - 3	9	Devuelve la resta de dos expresiones.
`*`	Multiplicación	12 * 3	36	Devuelve el producto de dos expresiones.
`/`	División	12 / 3	4	Devuelve el cociente de dos expresiones.
`%`	Módulo o Residuo	12 % 3	0	Devuelve el resto del cociente entre dos expresiones.

Relacionales

Son operadores que se utilizan para determinar el valor de verdad de la comparación entre dos expresiones. El resultado únicamente puede ser `true` (verdadero) o `false` (falso).

Operador	Nombre	Ejemplo	Resultado	Descripción
`==`	Igual que	12 == 3	false	Determina si una expresión es igual a la otra.
`!=`	Distinto de	12 != 3	true	Determina si una expresión es distinta a la otra.
`<`	Menor que	12 < 3	false	Determina si una expresión es menor a la otra.
`<=`	Menor o igual que	12 <= 3	false	Determina si una expresión es menor o igual a la otra.
`>`	Mayor que	12 > 3	true	Determina si una expresión es mayor a la otra.
`>=`	Mayor o igual que	12 >= 3	true	Determina si una expresión es mayor o igual a la otra.

Lógicos

Son operadores que permiten construir expresiones lógicas, a través de operandos de tipo lógico (booleanos).

Operador	Nombre	Ejemplo	Resultado	Descripción
`!`	NO	!true	false	Operador unitario. Niega una expresión lógica.
`||`	O	true || false	true	Operador de disyunción lógica. Devuelve true cuando cualquier operando sea true. Si todas los operandos son false, devuelve false.
`&&`	Y	true && false	false	Operador de conjunción lógica. Devuelve true cuando todos los operandos sean true. Si algún operando es false, devuelve false.

Incrementales

Son operadores unitarios que permiten incrementar o decrementar el valor de una variable.

Como ejemplo, la expresión `a++` podría escribirse de la forma `a = a + 1`. Este mismo criterio se cumple para el operador `--`.

Son muy útiles para hacer contadores.

Supongamos que tenemos una variable llamada `x` cuyo valor es 5.

Operador	Nombre	Ejemplo	Nuevo valor de x	Descripción
`++`	Incremento	x++	6	Le suma 1 al valor actual de la variable “x” y lo reasigna en “x”. (Lo anterior se pierde).
`--`	Decremento	x--	4	Le resta 1 al valor actual de la variable “x” y lo reasigna en “x”. (Lo anterior se pierde)..

Concatenación

Permite concatenar cadenas de texto con otras cadenas, números y expresiones.

Como el operador coincide con el aritmético de suma, es muy importante saber en qué contexto ubicarlo para prevenir salidas inesperadas.

Veremos varios ejemplos:

Ejemplo	Resultado	Explicación
2 + 3 + 4	9	Si todas las expresiones son numéricas, se comporta como operador de suma.
“2” + “3” + “4”	“234”	“2” y “3” son cadenas. El resultado de concatenarlas es “23”, que además se concatena con “4” y devuelve “234”.
”2” + 3 + 4	“234”	“2” es una cadena y 3 es un número. Al haber al menos una cadena como operando, se convierte en la concatenación y devuelve “23”, que además se concatena con “4” y devuelve “234”.
2 + 3 + “4”	“54”	2 + 3 equivale a 5, que además se concatena con “4” y devuelve “54”.
2 + (3 + “4”)	“234”	Los paréntesis tienen mayor prioridad. Primero se evalúa 3 y “4”, devuelve “34”. La concatenación entre el número 2 y la cadena “34” devuelve “234”.
‘2’ + 3 + 4	57	El carácter ‘2’ tiene asociado el código Ascii 50. Este número se suma con 3 y devuelve 53, que además se suma con 4 y devuelve 57.
2 + ‘3’ + ‘4’	105	El número 2 se suma con el código Ascii asociado al carácter ‘3’ que es 51. Esto devuelve el número 53 que se suma con el código Ascii asociado al carácter ‘4’, que es el 52, dando como resultado el número 105.

Recordemos que los caracteres (`char`), por más que se representen en la salida como símbolos, no dejan de ser números. Internamente, cada carácter está asociado a un número según la tabla de codificación ASCII.

Jerarquía

Al tener expresiones más complejas, donde estén involucrados muchos operadores, es necesario saber que éstas expresiones se evalúan de acuerdo a la siguiente jerarquía u orden de precedencia.

Operador/es	Asociatividad
++ -- !	Derecha a izquierda
* / %	Izquierda a derecha
+ -	Izquierda a derecha
< <= > >=	Izquierda a derecha
== !=	Izquierda a derecha
&&	Izquierda a derecha
||	Izquierda a derecha
= += -= *= /= %=	Derecha a izquierda

Paréntesis

Es posible agrupar expresiones entre paréntesis al igual que en las matemáticas para poder establecer manualmente la importancia de una evaluación.

Las expresiones entre paréntesis tienen la mayor prioridad.

Variables

Definición

En programación, una variable es un espacio de memoria reservado para almacenar un valor. Es representada y usada a través de una etiqueta (un nombre) que le asigna el programador.

Como su nombre lo indica, puede variar su valor durante la ejecución del programa.

Java es un **lenguaje de programación fuertemente tipado**, es decir, que dado el valor de una variable de un tipo concreto, no se puede usar como si fuera de otro tipo distinto a menos que se haga una conversión.

Tipos de datos primitivos

Son los tipos de datos elementales que maneja el lenguaje Java. Existen ocho tipos, de los cuales, uno guarda datos lógicos (`boolean`), cuatro permiten guardar números enteros (`byte`, `short`, `int` y `long`), dos que permiten guardar números reales (`float` y `double`) y uno que permite guardar caracteres (`char`).

Nombre	Declaración	Tamaño en bits	Rango	Descripción
Booleano	`boolean`	Según la JVM	True - False	Define una bandera que puede tomar dos posibles valores: true o false.
Byte	`byte`	8	[-128 .. 127]	Representación del número de menor rango con signo.
Entero Pequeño	`short`	16	[-32,768 .. 32,767]	Representación de un entero cuyo rango es pequeño.
Entero	`int`	32	$[-2^{31} .. 2^{31}-1]$	Representación de un entero estándar.
Entero largo	`long`	64	$[-2^{63} .. 2^{63}-1]$	Representación de un entero de rango ampliado.
Real	`float`	32	$[\pm 3{,}4 \cdot 10^{-38} .. \pm 3{,}4 \cdot 10^{38}]$	Representación de un real estándar. La precisión se amplía con números más próximos a 0 y disminuye cuanto más se aleja del mismo.
Real largo	`double`	64	$[\pm 1{,}7 \cdot 10^{-308} .. \pm 1{,}7 \cdot 10^{308}]$	Representación de un real de mayor precisión. Double tiene el mismo efecto con la precisión que float.
Caracter	`char`	16	['\u0000' .. '\uffff'] o [0 .. 65.535]	Carácter o símbolo. Para componer una cadena es preciso usar la clase **String.**

Conversiones de tipo

Consiste en almacenar el valor de una variable de un determinado tipo primitivo en otra variable de distinto tipo.

No todas las conversiones entre los distintos tipos de dato son posibles. Por ejemplo, en Java no es posible convertir valores booleanos a ningún otro tipo de dato y viceversa. Además, en caso de que la conversión sea posible es importante evitar la pérdida de información en el proceso.

En general, existen dos categorías de conversiones:

- **De ensanchamiento**: Por ejemplo, pasar de un valor entero a un real.
- **De estrechamiento**: Por ejemplo, pasar de un valor real a un entero.

Las **conversiones de ensanchamiento** transforman un dato de un tipo a otro con el mismo o mayor espacio en memoria para almacenar información. En estos casos puede haber una cierta pérdida de precisión al convertir un valor entero a real al desechar algunos dígitos significativos. Las conversiones de ensanchamiento en Java se resumen en la siguiente tabla.

Tipo de origen	Tipo de destino
`byte`	`short, int, long, float, double`
`short`	`int, long, float, double`
`char`	`int, long, float, double`
`int`	`long, float, double`
`long`	`double`
`float`	`float, double`

Las **conversiones de estrechamiento** son más comprometidas ya que transforman un dato de un tipo a otro con menor espacio en memoria para almacenar información. En estos casos se corre el riesgo de perder o alterar sensiblemente la información. Las conversiones de estrechamiento en Java se resumen en la siguiente tabla.

Tipo de origen	Tipo de destino
`byte`	`char`
`short`	`byte, char`
`char`	`byte, short`
`int`	`byte, short, char`
`long`	`byte, short, char, int`
`float`	`byte, short, char, int, long`
`double`	`byte, short, char, int, long, float`

Las conversiones se realizan por medio de los siguientes mecanismos:

Por asignación: cuando una variable de un determinado tipo se asigna a una variable de otro tipo. Sólo admite conversiones de ensanchamiento.

Por ejemplo: si `n` es una variable de tipo `int` que vale `25` y `x` es una variable de tipo `double`, entonces se produce una conversión por asignación al ejecutarse la sentencia

```
x = n;
```

la variable `x` toma el valor `25.0` (valor en formato real). El valor de `n` no se modifica.

Por promoción aritmética: como resultado de una operación aritmética. Como en el caso anterior, sólo admite conversiones de ensanchamiento.

Por ejemplo, si `producto` y `factor1` son variables de tipo `double` y `factor2` es de tipo `int` entonces al ejecutarse la sentencia

```
producto = factor1 * factor2;
```

el valor de `factor2` se convierte internamente en un valor en formato real para realizar la operación aritmética que genera un resultado de tipo `double`. El valor almacenado en formato entero en la variable `factor2` no se modifica.

Con casting o "moldes": con operadores que producen la conversión entre tipos. Admite las conversiones de ensanchamiento y estrechamiento indicadas anteriormente.

Por ejemplo: si se desea convertir un valor de tipo `double` a un valor de tipo `int` se utilizará el siguiente código:

```
int n;
double x = 82.4;
```

```
n = (int) x;
```

La variable `n` toma el valor `82` (valor en formato entero). El valor de `x` no se modifica.

El siguiente código ilustra algunas de las conversiones que pueden realizarse entre datos de tipo numérico.

```
public class JavaApplication1 {
    public static void main(String[] args) {
        int a = 2;
        double b = 3.0;
        float c = (float) (20000 * a / b + 5);
        System.out.println("Valor en formato float: " + c);
        System.out.println("Valor en formato double: " + (double) c);
        System.out.println("Valor en formato byte: " + (byte) c);
        System.out.println("Valor en formato short: " + (short) c);
        System.out.println("Valor en formato int: " + (int) c);
        System.out.println("Valor en formato long: " + (long) c);
    }
}
```

Código 6: Conversiones entre tipos de dato

```
run:
Valor en formato float: 13338.333
Valor en formato double: 13338.3330078125
Valor en formato byte: 26
Valor en formato short: 13338
Valor en formato int: 13338
Valor en formato long: 13338
BUILD SUCCESSFUL (total time: 0 seconds)
```

Salida 4: Conversiones entre tipos de dato

Tipos de dato por referencia

Los tipos de datos referencia, indican que vamos a trabajar con instancias de clases, no con tipos primitivos. De esta manera, una variable de tipo referencia establece una conexión hacia un objeto, y a través de esta conexión podemos acceder a sus métodos y atributos.

Veremos en lecciones posteriores de qué trata el tema clases y objetos.

Nombres de variables

Toda variable debe tener un identificador (nombre) único en el contexto del programa.

Además, debe seguir las siguientes reglas:

- No puede ser una palabra reservada del lenguaje o un literal booleano (true o false).

- Puede contener cualquier carácter Unicode, pero no puede comenzar con un número.
- No debe contener los símbolos que se utilicen como operadores (`+` , `/` , `?`, etc.)

Por convención, los nombres de variables comienzan con una letra en minúscula.

Si un nombre consiste en más de una palabra, se escribirá sin espacios entre ellas y cada palabra (salvo la primera) comenzará con una letra mayúscula. Dicha notación se llama **CamelCase** (por ejemplo: `estaBienEsteNombre`).

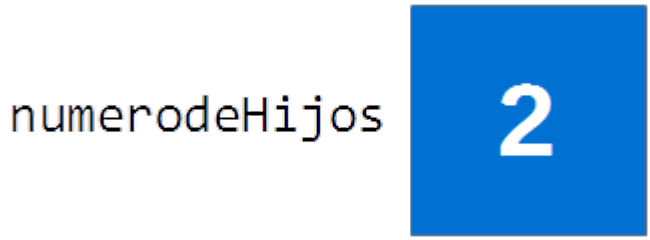

Ilustración 1: Variables

En la ilustración vemos como ejemplo a la variable `numeroDeHijos` que apunta a una celda de memoria que guarda el valor `2`.

Declaración de variables

Sin inicializar

Las variables se declaran usando la siguiente sentencia:

`Tipo_de_Dato Nombre_de_Variable;`

Donde `Tipo_de_Dato` será una palabra clave de la columna "Declaración" de la tabla mostrada en la sección "Tipos de dato…" y `Nombre_de_Variable` será un identificador que usted elija siguiendo las reglas descriptas en la sección "Nombres de variables".

El caracter `;` es obligatorio y se encarga de cerrar la sentencia.

La variable ya está definida, pero aún no tiene valor. Cualquier referencia a ella devolverá como resultado **null**.

```
public class JavaApplication1 {
      public static void main(String[] args) {
            boolean continuarCiclo;
            byte edad;
            short stockDelProducto;
            int dni;
            long masaDelSol;
            float superficie;
            double saldoEnCuenta;
            char opcion;
      }
}
```

Código 7: Declaración de variables sin inicializar

Inicializadas

Es posible declarar una variable e inicializarla con un valor, usando la siguiente sentencia:

`Tipo_de_Dato Nombre_de_Variable = Valor_Inicial;`

Donde `Tipo_de_Dato` será una palabra clave de la columna "Declaración" de la tabla mostrada en la sección "Tipos de dato…" y `Nombre_de_Variable` será un identificador que usted elija siguiendo las reglas descriptas en la sección "Nombres de variables".

El caracter `=` es el operador de asignación, quien guardará en dicha variable lo que esté a su derecha.

`Valor_Inicial` será un valor que contendrá la variable una vez definida. Es muy importante que dicho valor sea del mismo tipo de dato que la variable.

El caracter `;` es obligatorio y se encarga de cerrar la sentencia.

```
public class JavaApplication1 {
      public static void main(String[] args) {
            boolean continuarCiclo = true;
            byte edad = 18;
            short stockDelProducto = 15000;
            int dni = 35234567;
            long masaDelSol = 1054829483;
            float superficie = 200.31;
            double saldoEnCuenta = 12345.444444;
            char opcion = 'a';
      }
}
```

Código 8: Declaración de variables inicializadas

Entrada por consola

En este apartado aparecerán términos como “clase”, “instancia”, “objeto” o “método”. Los mismos serán explicados en lecciones posteriores.

La clase Scanner

Importar la clase

Para leer datos desde teclado en la consola se debe utilizar un objeto de la clase **Scanner**, que no está incluida de forma automática.

En este caso la clase **Scanner** es parte del paquete **util** que es a su vez parte de la clase **java**.

Para importar la clase **Scanner**, es necesario explicitar la siguiente línea antes de comenzar a definir la clase principal.

`import java.util.Scanner;`

Generar el objeto

Una vez tenemos la clase **Scanner** importada, debemos generar una instancia (objeto) de dicha clase para comenzar a trabajar.

El siguiente código declara e inicializa un objeto de la clase **Scanner** llamada `entrada` que lee datos desde la entrada estándar (`System.in`).

```
import java.util.Scanner;

public class JavaApplication1 {
    public static void main(String[] args) {
        Scanner entrada = new Scanner(System.in);
    }
}
```

Código 9: Declaración e inicialización de un objeto entrada

Leer datos

Tipos numéricos

La clase **Scanner** tiene métodos que permiten leer datos de tipo numérico. Es muy importante saber el tipo de dato esperado para que no se generen errores.

Los métodos son `nextByte()`, `nextShort()`, `nextInt()`, `nextLong()`, `nextFloat()` y `nextDouble()`.

```
import java.util.Scanner;

public class JavaApplication1 {
	public static void main(String[] args) {
		Scanner entrada = new Scanner(System.in);

		byte edad = entrada.nextByte();
		short stockDelProducto = entrada.nextShort();
		int dni = entrada.nextInt();
		long masaDelSol = entrada.nextLong();
		float superficie = entrada.nextFloat();
		double saldoEnCuenta = entrada.nextDouble();
	}
}
```

Código 10: Leer datos numéricos

Tipos alfanuméricos

La clase **Scanner** no tiene un método para leer un simple carácter.

Utilizaremos el método `nextLine()` que permite leer una línea completa en formato **String**. Una vez capturada esa línea, nos quedaremos con el primer carácter de la misma a través del método `charAt(0)`.

```
import java.util.Scanner;

public class JavaApplication1 {
	public static void main(String[] args) {
		Scanner entrada = new Scanner(System.in);

		char opcion = entrada.nextLine().charAt(0);
	}
}
```

Código 11: Leer datos alfanuméricos

Estructuras de control

Generalmente, en un programa las instrucciones se ejecutan una después de otra, en el orden en que están escritas. Este proceso se conoce como ejecución secuencial.

Hay instrucciones en Java que nos permiten especificar que la siguiente instrucción a ejecutarse tal vez no sea la siguiente en la secuencia. Esto se conoce como transferencia de control.

Durante los ‘60, se hizo evidente que el uso indiscriminado de las transferencias de control era el origen de muchas de las dificultades que experimentaban los grupos de desarrollo de software. A quien se señaló como culpable fue a la instrucción **goto** (utilizada en la mayoría de los lenguajes de programación de esa época), la cual permite al programador especificar la transferencia de control a uno de los muchos posibles destinos dentro de un programa.

La noción de la llamada programación estructurada se hizo casi un sinónimo de la “eliminación del **goto**”. (Java no tiene una instrucción **goto**; sin embargo, la palabra **goto** está reservada y no debe usarse como identificador en los programas).

Se atribuyen generalmente a Corrado Böhm y Giuseppe Jacopini las bases para enunciar el teorema del programa estructurado, que demuestra que los programas pueden escribirse sin instrucciones **goto**. El reto de la época para los programadores fue cambiar sus estilos a una “programación sin **goto**”.

No fue sino hasta los ‘70 cuando los programadores tomaron en serio la programación estructurada. Los resultados fueron impresionantes. Los grupos de desarrollo de software reportaron reducciones en los tiempos de desarrollo, mayor incidencia de entregas de sistemas a tiempo y más proyectos de software finalizados sin salirse del presupuesto.

La clave para estos logros fue que los programas estructurados eran más claros, más fáciles de depurar y modificar, y había más probabilidad de que estuvieran libres de errores desde el principio.

El trabajo de Böhm y Jacopini demostró que todos los programas podían escribirse en términos de solo tres estructuras de control: la estructura secuencial, la estructura de selección y la estructura de repetición.

Estructura secuencial

La estructura secuencial está integrada en Java. A menos que se le indique lo contrario, la computadora ejecuta las instrucciones en Java una después de otra, en el orden en que estén escritas; es decir, en secuencia.

Java permite tantas acciones como deseemos en una estructura secuencial: donde quiera que se coloque una sola acción, podrán colocarse varias acciones en secuencia.

Estructuras de selección

Java tiene tres tipos de instrucciones de selección.

La instrucción if (selección simple) realiza una acción si la condición es verdadera, o evita la acción si la condición es falsa.

La instrucción if...else (selección doble) realiza una acción si la condición es verdadera, o realiza una acción distinta si la condición es falsa.

La instrucción switch (selección múltiple) realiza una de entre varias acciones distintas, dependiendo del valor de una expresión.

IF

Una estructura de selección simple en Java se define con la palabra clave `if` seguida de la condición entre paréntesis (obligatorio) seguida de llaves de apertura y cierre `{}`.

Todas las sentencias que estén delimitadas entre `{}` serán ejecutadas si la condición es verdadera, de lo contrario, continúa con las acciones por debajo de la llave de cierre `}`.

```
public class JavaApplication1 {
      public static void main(String[] args) {
            int edad = 20;
            if (edad >= 18)
            {
                  System.out.println("Sos mayor de edad");
            }
      }
}
```

Código 12: Instrucción de selección simple IF

```
run:
Sos mayor de edad
BUILD SUCCESSFUL (total time: 0 seconds)
```

Salida 5: Instrucción de selección simple IF

IF...ELSE

Una estructura de selección doble puede tomar acciones en los casos en que el resultado de la condición sea falso. Para contemplar este caso, se agrega la palabra `else` seguida de sus propias llaves de apertura y cierre `{}`, luego de la llave de cierre `}` del `if`. Las acciones que van a ser ejecutadas en los casos falsos irán dentro de ese bloque.

```
public class JavaApplication1 {
      public static void main(String[] args) {
            int edad = 15;
            if (edad >= 18)
            {
                  System.out.println("Sos mayor de edad");
            }
            else
            {
                  System.out.println("Sos menor de edad");
            }
      }
}
```

Código 13: Instrucción de selección doble anidada IF..ELSE

```
run:
Sos menor de edad
BUILD SUCCESSFUL (total time: 0 seconds)
```

Salida 6: Instrucción de selección doble IF...ELSE

IF...ELSE anidados

Una sentencia `if` puede ser seguida por un opcional `else if`, que es muy útil para comprobar varias condiciones.

```
public class JavaApplication1 {
      public static void main(String[] args) {
            int horaDelDia = 16;
            if (horaDelDia < 0 || horaDelDia > 23)
            {
                  System.out.println("Hora inválida.");
            }
            else if (horaDelDia < 12)
            {
                  System.out.println("Es la mañana.");
            }
            else if (horaDelDia == 12)
            {
                  System.out.println("Es el mediodía.");
            }
            else if (horaDelDia < 20)
            {
                  System.out.println("Es la tarde.");
            }
            else
            {
                  System.out.println("Es la noche.");
            }
      }
}
```

Código 14: Instrucción de selección doble anidada IF...ELSE IF...ELSE

```
run:
Es la tarde.
BUILD SUCCESSFUL (total time: 0 seconds)
```

Salida 7: Instrucción de selección doble anidada IF...ELSE IF...ELSE

SWITCH

Para evaluar casos múltiples, que dependan del valor concreto de una variable podemos usar la estructura condicional múltiple.

En Java, se define con la palabra `switch` seguida de la variable o expresión entera a evaluar entre paréntesis (obligatorio). A continuación, llaves de apertura y de cierre `{}` para delimitar las acciones que están dentro del bloque.

Cada caso que se quiera evaluar tendrá la palabra `case` seguida de la expresión entera que se espera obtener, seguido del carácter `:`.

```
public class JavaApplication1 {
      public static void main(String[] args) {
            int diaDeLaSemana = 4;
            switch (diaDeLaSemana)
            {
                  case 1:
                        System.out.println("LUNES");
                        break;
                  case 2:
                        System.out.println("MARTES");
                        break;
                  case 3:
                        System.out.println("MIERCOLES");
                        break;
                  case 4:
                        System.out.println("JUEVES");
                        break;
                  case 5:
                        System.out.println("VIERNES");
                        break;
                  case 6:
                        System.out.println("SABADO");
                        break;
                  case 7:
                        System.out.println("DOMINGO");
                        break;
                  default:
                        System.out.println("DIA INVALIDO");
            }
      }
}
```

Código 15: Instrucción de selección múltiple SWITCH

```
run:
JUEVES
BUILD SUCCESSFUL (total time: 0 seconds)
```

Salida 8: Instrucción de selección múltiple SWITCH

Las siguientes reglas se aplican a una sentencia `switch`:

- La variable que se utiliza para comparar en una sentencia `switch` sólo puede ser un `byte`, `short` , `int`, o `char`.
- Podemos tener cualquier número de sentencias case dentro de un `switch`. Cada caso es seguido del valor a ser comparado.
- El valor de un caso debe ser el mismo tipo de datos que la variable en el `switch`.
- Cuando la variable del `switch` es igual a un caso, las instrucciones que siguen a ese caso se ejecutarán hasta que se alcanza una sentencia `break`.
- Cuando se llega a una sentencia `break`, el caso termina, y el flujo de control pasa a la siguiente línea después de la sentencia `switch`.
- No todos los casos tienen que contener un `break`.
- Una sentencia `switch` puede tener un caso por defecto (`default`). Es opcional y debe aparecer al final del `switch`. Podría utilizarse para realizar una tarea cuando ninguno de los casos es cierto.

Estructuras de repetición

Una estructura de repetición nos permite especificar que un programa debe repetir una acción o más acciones mientras cierta condición sea verdadera. Cuando la condición pase a ser falsa, el programa continuará con la ejecución de las demás sentencias debajo de la estructura de control.

Este tipo de estructuras se denominan **ciclos**, **bucles** o **loops**. Un ciclo puede tener muchas repeticiones, las cuales son normalmente llamadas **vueltas** o **iteraciones**.

La condición que evalúa un ciclo debe tener al menos una variable involucrada, de lo contrario, sería eternamente verdadera o falsa.

Toda estructura repetitiva tiene tres conceptos que deben estar bien analizados.

- **Inicialización**: Cómo será el valor inicial de la(s) variable(s) en la condición.
- **Condición**: Qué tiene que ocurrir para que el ciclo continúe su ejecución.
- **Actualización**: La(s) variable(s) en la condición debe(n) cambiar su valor por cada iteración.

Los ciclos pueden controlarse de dos maneras: **por contador** y **por bandera**.

En el primer caso, siempre se ejecuta un número definido de iteraciones. Una variable entera oficia de contadora de repeticiones. Comenzará con un valor inicial. Cada iteración actualizará el valor de la variable contadora. El ciclo seguirá dando vueltas mientras la variable contadora supere o no (según el caso) a cierto valor.

En el segundo caso, no está definido un número de iteraciones. Una variable booleana oficia de “centinela”. Comenzará con un valor inicial verdadero. En cada iteración se evaluará si hay un cambio de estado en la variable centinela. El ciclo seguirá dando vueltas mientras el centinela no cambie su estado.

Hay tres sentencias que nos permiten formular repeticiones: **WHILE**, **DO-WHILE** y **FOR**.

Generalmente se usan los bloques **WHILE** y **DO-WHILE** para formular algoritmos repetitivos controlados por bandera y el **FOR** para algoritmos repetitivos controlados por contador.

WHILE

Una de las estructuras de repetición en Java se define con la palabra clave `while`, luego la condición entre paréntesis (obligatorio) seguida de llaves de apertura y cierre `{}`.

Todas las sentencias que estén delimitadas entre `{}` serán ejecutadas mientras la condición sea verdadera. Al llegar a la llave de cierre `}`, el flujo vuelve hacia arriba a volver a evaluar la condición. Si resulta verdadera, vuelve a ejecutar las acciones entre las llaves `{}`, de lo contrario, continúa con las acciones por debajo de la llave de cierre `}`.

En este tipo de ciclos la condición se evalúa al principio, por lo que las acciones puede que no siempre se ejecuten.

```
public class JavaApplication1 {
      public static void main(String[] args) {
            int contador = 1;
            while (contador <= 5)
            {
                  System.out.println("Ciclo Nº " + contador);
                  contador++;
            }
      }
}
```

Código 16: Instrucción de repetición WHILE

```
run:
Ciclo Nº 1
Ciclo Nº 2
Ciclo Nº 3
Ciclo Nº 4
Ciclo Nº 5
BUILD SUCCESSFUL (total time: 0 seconds)
```

Salida 9: Instrucción de repetición WHILE

DO-WHILE

Otra de las estructuras de repetición en Java se define con la palabra clave `do` seguida de llaves de apertura y cierre `{}`. Luego de la llave de cierre `}` se escribe la palabra clave `while` seguida de la condición entre paréntesis (obligatorio) y un `;` para cerrar el bloque.

Todas las sentencias que estén delimitadas entre las llaves de apertura y cierre `{}` serán ejecutadas mientras la condición sea verdadera. Al llegar a la llave de cierre `}`, se evalúa la condición luego de la palabra `while`. Si resulta verdadera, vuelve hacia arriba a ejecutar las acciones entre las llaves `{}`, de lo contrario, continúa con las acciones luego del `;`.

En este tipo de ciclos la condición se evalúa al final, por lo que las acciones siempre se ejecutarán al menos una vez.

```
public class JavaApplication1 {
      public static void main(String[] args) {
            int contador = 1;
            do
            {
                  System.out.println("Ciclo Nº " + contador);
                  contador++;
            } while (contador <= 5);
      }
}
```

Código 17: Instrucción de repetición DO...WHILE

```
run:
Ciclo Nº 1
Ciclo Nº 2
Ciclo Nº 3
Ciclo Nº 4
Ciclo Nº 5
BUILD SUCCESSFUL (total time: 0 seconds)
```

Salida 10: Instrucción de repetición DO...WHILE

FOR

La última estructura de repetición en Java se define con la palabra clave `for` seguida de un paréntesis que abre `(`. Se escribe una inicialización de una variable contadora, seguido de un `;`, luego la condición, otro `;` y por último la actualización de la variable contadora. Al final se cierra el paréntesis `)` y se abren llaves de apertura y cierre `{}`.

Todas las sentencias que estén delimitadas entre las llaves de apertura y cierre `{}` serán ejecutadas mientras la condición sea verdadera. Al llegar a la llave de cierre `}`, el flujo vuelve hacia arriba a volver a evaluar la condición. Si resulta verdadera, vuelve a ejecutar las acciones entre las llaves `{}`, de lo contrario, continúa con las acciones por debajo de la llave de cierre `}`.

En este tipo de ciclos la inicialización, condición y actualización están en la cabecera. Son muy adecuados para hacer ciclos controlados por contador.

```
public class JavaApplication1 {
      public static void main(String[] args) {
            for (int contador = 1; contador <= 5; contador++) {
                  System.out.println("Ciclo Nº " + contador);
            }
      }
}
```

Código 18: Instrucción de repetición FOR

```
run:
Ciclo Nº 1
Ciclo Nº 2
Ciclo Nº 3
Ciclo Nº 4
Ciclo Nº 5
BUILD SUCCESSFUL (total time: 0 seconds)
```

Salida 11: Instrucción de repetición FOR

Arreglos

Definición

Los arreglos son estructuras de datos que consisten de elementos de datos relacionados, del mismo tipo. Son entidades de longitud fija, es decir, conservan la misma longitud una vez creados, aunque puede reasignarse una variable tipo arreglo de tal forma que haga referencia a un nuevo arreglo de distinta longitud.

Los arreglos son objetos, por lo que se consideran como tipos de referencia. Como veremos pronto, lo que consideramos generalmente como un arreglo es en realidad una referencia a un objeto arreglo en memoria. Los elementos de un arreglo pueden ser tipos primitivos o de referencia.

Para hacer referencia a un elemento en un arreglo, debemos especificar el nombre de la referencia al arreglo y el número de la posición del elemento en el arreglo. El número de la posición del elemento se conoce formalmente como el **índice** del elemento. El índice de un arreglo comienza siempre a **contar desde** el número **0**.

Declaración

Sin inicializar

Para declarar un objeto arreglo, debemos especificar el tipo de dato que alojará junto a corchetes de apertura y cierre `[]` (que indican que se trata de un conjunto de datos de ese tipo) y luego el nombre del arreglo (siguiendo el mismo criterio de nomenclatura explicado en la sección "Nombres de variables".

El siguiente código declara un arreglo de enteros llamado `numeros`.

```
public class JavaApplication1 {
      public static void main(String[] args) {
            int[] numeros;
      }
}
```

Código 19: Declaración de un arreglo sin inicializar

Por ahora, `numeros` apunta a una referencia nula. Todavía no tenemos creado el arreglo.

Inicializado por defecto

Podemos declarar y crear el arreglo en la misma sentencia (mismo criterio que con las variables de tipos primitivos).

Ahora a `numeros` le asignaremos (mediante el operador de asignación `=`) un nuevo arreglo de **5** elementos.

Para ello, utilizamos la palabra clave `new`, seguido del tipo de dato del arreglo, y entre los corchetes de apertura y cierre `[]` irá el número de elementos de los que consta el arreglo.

El siguiente código declara un arreglo de enteros llamado `numeros` que consta de **5** elementos.

```
public class JavaApplication1 {
      public static void main(String[] args) {
            int[] numeros = new int[5];
      }
}
```

Código 20: Declaración de un arreglo inicializado con valores por defecto

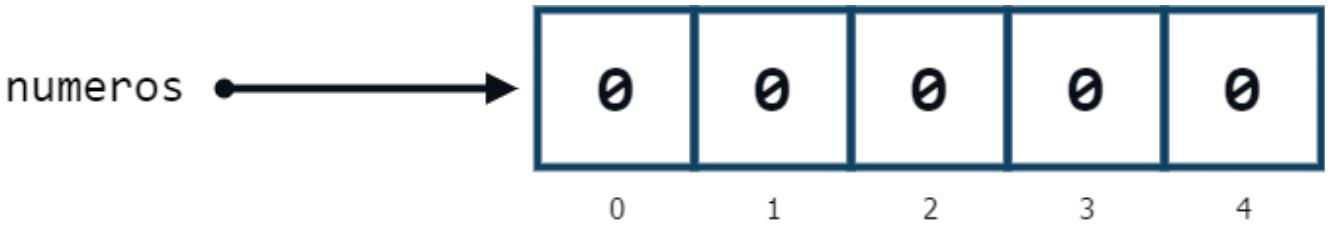

Ilustración 2: Arreglo inicializado con valores por defecto

Tenemos entonces un arreglo de enteros llamado `numeros` de tamaño **5**. Pero, ¿qué pasa con cada elemento guardado en el arreglo?

La respuesta depende del tipo de dato guardado. Los valores numéricos (`char`, `byte`, `short`, `int`, `long`, `float` y `double`) se iniciarán con el valor por defecto `0`. Los valores booleanos (`boolean`) se iniciarán con el valor por defecto `false` y cualquier **dato no primitivo** iniciará con valor por defecto `null`.

Inicializado con valores

Podemos declarar e inicializar un arreglo con nuestros valores elegidos.

Para ello, encerramos nuestros valores separados por comas `,` entre llaves de apertura y cierre `{}`. El tamaño del arreglo entonces coincidirá con la cantidad de elementos entre las llaves `{}`.

El siguiente código declara un arreglo de enteros llamado `numeros` inicializado con los valores **7**, **4**, **9**, **-3** y **0**. El tamaño del arreglo es **5**.

```
public class JavaApplication1 {
      public static void main(String[] args) {
            int[] numeros = {7, 4, 9, -3, 0};
      }
}
```

Código 21: Declaración de un arreglo inicializado con valores definidos

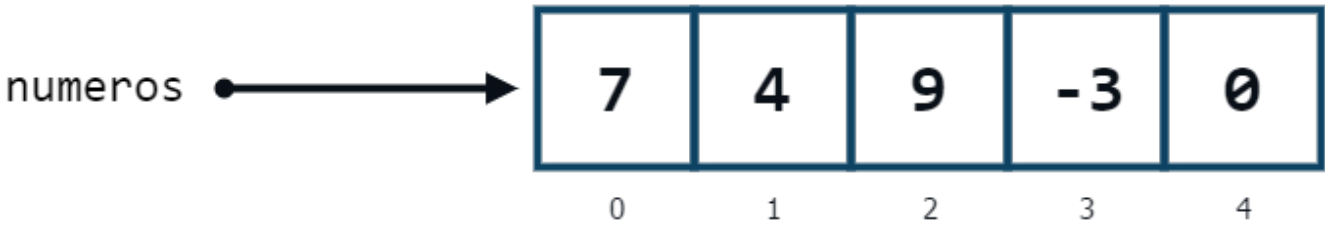

Ilustración 3: Arreglo inicializado con valores definidos

Obtener la longitud de un arreglo

Todo arreglo cuenta con el atributo `length` que permite conocer el número de elementos de los que consta.

```
public class JavaApplication1 {
      public static void main(String[] args) {
            int[] numeros = {7, 4, 9, -3, 0};
            System.out.println("La longitud del arreglo es " + numeros.length);
      }
}
```

Código 22: Obtener la longitud de un arreglo

```
run:
La longitud del arreglo es 5
BUILD SUCCESSFUL (total time: 0 seconds)
```

Salida 12: Obtener la longitud de un arreglo

Obtener un valor de un arreglo

Es posible obtener cualquier valor de un arreglo.

Para ello, se ingresa el nombre del arreglo en cuestión seguido de corchetes de apertura y cierre `[]`. Dentro de los corchetes ingresamos el número de índice donde se encuentra el dato que nos interesa obtener.

```java
public class JavaApplication1 {
      public static void main(String[] args) {
            int[] numeros = {7, 4, 9, -3, 0};
            int x = numeros[1];
            System.out.println("El valor en la 2da posición es " + x);
            System.out.println("El valor en la 4ta posición es " + numeros[3]);
      }
}
```

Código 23: Obtener un valor de un arreglo

```
run:
El valor en la 2da posición es 4
El valor en la 4ta posición es -3
BUILD SUCCESSFUL (total time: 0 seconds)
```

Salida 13: Obtener un valor de un arreglo

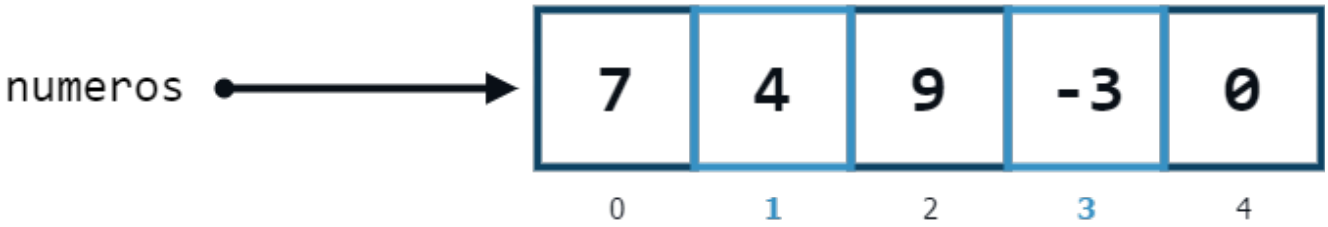

Ilustración 4: Obtener un valor de un arreglo

Reemplazar un valor de un arreglo

Es posible reemplazar cualquier valor de un arreglo.

Para ello, se ingresa el nombre del arreglo en cuestión seguido de corchetes de apertura y cierre `[]`. Dentro de los corchetes ingresamos el número de índice donde se encuentra el dato que nos interesa reemplazar. Ahora podemos guardar un nuevo valor gracias al operador de asignación `=`.

```java
public class JavaApplication1 {
	public static void main(String[] args) {
		int[] numeros = {7, 4, 9, -3, 0};
		System.out.println("El valor en la 4ta posición es " + numeros[3]);
		numeros[3] = 10;
		System.out.println("El nuevo valor en la 4ta posición es " + numeros[3]);
	}
}
```

Código 24: Reemplazar un valor de un arreglo

```
run:
El valor en la 4ta posición es -3
El nuevo valor en la 4ta posición es 10
BUILD SUCCESSFUL (total time: 0 seconds)
```

Salida 14: Reemplazar un valor de un arreglo

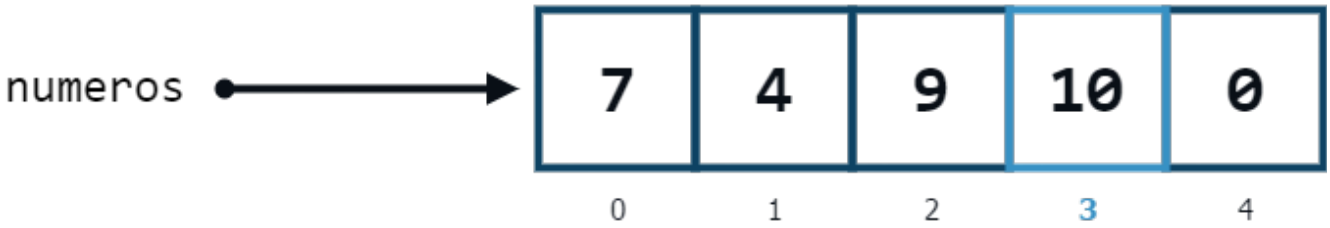

Ilustración 5: Reemplazar un valor de un arreglo

Mostrar todos los valores de un arreglo

Es posible mostrar todos los valores de cualquier arreglo. Para ello, hacemos uso de la estructura de repetición `for`.

```java
public class JavaApplication1 {
	public static void main(String[] args) {
		int[] numeros = {7, 4, 9, -3, 0};
		for (int i = 0; i < numeros.length; i++)
		{
			System.out.println("El " + (i+1) + "º valor es " + numeros[i]);
		}
	}
}
```

Código 25: Mostrar todos los valores de un arreglo

```
run:
El 1º valor es 7
El 2º valor es 4
El 3º valor es 9
El 4º valor es -3
El 5º valor es 0
BUILD SUCCESSFUL (total time: 0 seconds)
```

Salida 15: Mostrar todos los valores de un arreglo

Ordenamiento de arreglos

Debido a que las estructuras de datos son utilizadas para almacenar información, es deseable que ésta esté ordenada para poder recuperarla de manera eficiente. Existen varios métodos para ordenar las diferentes estructuras de datos básicas.

Hay métodos muy simples de implementar que son útiles en los casos en dónde el número de elementos a ordenar no es muy grande (menos de 500 elementos). Por otro lado hay métodos sofisticados, más difíciles de implementar pero que son más eficientes en cuestión de tiempo de ejecución.

Un punto importante a comprender acerca del ordenamiento es que el resultado final (los datos ordenados) será el mismo, sin importar qué algoritmo se utilice para ordenar los datos. La elección del algoritmo sólo afecta al tiempo de ejecución y el uso que haga el programa de la memoria.

Los siguientes algoritmos de ordenamiento que se mostrarán son los más fáciles de implementar.

Tras su ejecución, los valores quedarán ordenados de forma ascendente (de menor a mayor).

Ordenamiento por burbujeo (Bubble Sort)

La idea de este método es ir tomando los elementos de a dos e ir comparándolos e intercambiándolos de ser necesario, hasta que todos los elementos sean comparados.

```
public class JavaApplication1 {
     public static void main(String[] args) {
          int[] arreglo = {1, 5, 8, 0, 6, -3, -7, 9, 14, 6, 1, 0, 4};
          int tam = arreglo.length;
          System.out.print("El arreglo original: ");
          for (int i = 0; i < tam; i++)
          {
               System.out.print(arreglo[i] + " ");
          }
          System.out.println("");
          /*INICIO DEL ALGORITMO DE ORDENAMIENTO POR BURBUJEO*/
          for (int i = 0; i < tam - 1; i++)
          {
               for (int j = i + 1; j < tam; j++)
               {
                    if(arreglo[i] > arreglo[j])
                    {
                         int aux = arreglo[i];
                         arreglo[i] = arreglo[j];
                         arreglo[j] = aux;
                    }
               }
          }
          /*FIN DEL ALGORITMO DE ORDENAMIENTO POR BURBUJEO*/
          System.out.print("El arreglo ordenado: ");
          for (int i = 0; i < tam; i++)
          {
               System.out.print(arreglo[i] + " ");
          }
          System.out.println("");
     }
}
```

Código 26: Ordenamiento por burbujeo

Ordenamiento por inserción (Insertion Sort)

En este procedimiento se recurre a una búsqueda binaria en lugar de una búsqueda secuencial para insertar un elemento en la parte de arriba del arreglo, que ya se encuentra ordenado. El proceso se repite desde el segundo hasta el n-ésimo elemento.

```
public class JavaApplication1 {
      public static void main(String[] args) {
            int[] arreglo = {1, 5, 8, 0, 6, -3, -7, 9, 14, 6, 1, 0, 4};
            int tam = arreglo.length;
            System.out.print("El arreglo original: ");
            for (int i = 0; i < tam; i++)
            {
                  System.out.print(arreglo[i] + " ");
            }
            System.out.println("");
            /*INICIO DEL ALGORITMO DE ORDENAMIENTO POR INSERCION*/
            for(int i = 1; i < tam; i++)
            {
                  int temp = arreglo[i];
                  int izq = 0;
                  int der = i - 1;
                  while(izq <= der)
                  {
                        int medio = (izq + der) / 2;
                        if (temp < arreglo[medio])
                        {
                              der = medio - 1;
                        }
                        else
                        {
                              izq = medio + 1;
                        }
                  }
                  for (int j = i - 1; j >= izq; j--)
                  {
                        arreglo[j+1] = arreglo[j];
                  }
                  arreglo[izq] = temp;
            }
            /*FIN DEL ALGORITMO DE ORDENAMIENTO POR INSERCION*/
            System.out.print("El arreglo ordenado: ");
            for (int i = 0; i < tam; i++)
            {
                  System.out.print(arreglo[i] + " ");
            }
            System.out.println("");
      }
}
```

Código 27: Ordenamiento por inserción

Ordenamiento por selección (Selection Sort)

El método se basa en buscar en cada iteración el mínimo elemento del “subarreglo” situado entre el índice **i** y el final del arreglo e intercambiarlo con el de índice **i**. Tomando la dimensión del arreglo **tam** como tamaño del problema es inmediato que el bucle se repite **tam** veces.

```
public class JavaApplication1 {
      public static void main(String[] args) {
            int[] arreglo = {1, 5, 8, 0, 6, -3, -7, 9, 14, 6, 1, 0, 4};
            int tam = arreglo.length;
            System.out.print("El arreglo original: ");
            for (int i = 0; i < tam; i++)
            {
                  System.out.print(arreglo[i] + " ");
            }
            System.out.println("");
            /*INICIO DEL ALGORITMO DE ORDENAMIENTO POR SELECCION*/
            for (int i = 0; i < tam; i++)
            {
                  int imin=i;
                  for (int j = i + 1; j < tam; j++)
                  {
                        if(arreglo[j] < arreglo[imin])
                        {
                              imin = j;
                        }
                  }
                  int aux = arreglo[i];
                  arreglo[i] = arreglo[imin];
                  arreglo[imin] = aux;
            }
            /*FIN DEL ALGORITMO DE ORDENAMIENTO POR SELECCION*/
            System.out.print("El arreglo ordenado: ");
            for (int i = 0; i < tam; i++)
            {
                  System.out.print(arreglo[i] + " ");
            }
            System.out.println("");
      }
}
```

Código 28: Ordenamiento por selección

Ordenamiento Shell (Shell Sort)

Denominado así por su desarrollador Donald Shell (1959), ordena una estructura de una manera similar a la del ordenamiento por burbujeo, sin embargo no ordena elementos adyacentes sino que utiliza una segmentación entre los datos. Esta segmentación puede ser de cualquier tamaño de acuerdo a una secuencia de valores que empiezan con un valor grande (pero menor al tamaño total de la estructura) y van disminuyendo hasta llegar al 1.

```
public class JavaApplication1 {
      public static void main(String[] args) {
            int[] arreglo = {1, 5, 8, 0, 6, -3, -7, 9, 14, 6, 1, 0, 4};
            int tam = arreglo.length;
            System.out.print("El arreglo original: ");
            for (int i = 0; i < tam; i++)
            {
                  System.out.print(arreglo[i] + " ");
            }
            System.out.println("");
            /*INICIO DEL ALGORITMO DE ORDENAMIENTO POR SHELL*/
            int h = tam / 2;
            while (h > 0)
            {
                  for (int i = h-1; i < tam; i++)
                  {
                        int b = arreglo[i];
                        int j = i;
                        for (j = i; (j >= h) && (arreglo[j - h] > b); j -= h)
                        {
                              arreglo[j] = arreglo[j - h];
                        }
                        arreglo[j] = b;
                  }
                  h = h / 2;
            }
            /*FIN DEL ALGORITMO DE ORDENAMIENTO POR SHELL*/
            System.out.print("El arreglo ordenado: ");
            for (int i = 0; i < tam; i++)
            {
                  System.out.print(arreglo[i] + " ");
            }
            System.out.println("");
      }
}
```

Código 29: Ordenamiento por Shell

Búsqueda en arreglos

Un algoritmo de búsqueda es aquel que está diseñado para localizar un elemento con ciertas propiedades dentro de una estructura de datos. Entre tantos, hay dos algoritmos de búsqueda muy comunes: uno que es fácil de programar, pero relativamente ineficiente, y uno que es relativamente eficiente pero más complejo y difícil de programar.

Búsqueda secuencial

El algoritmo de **búsqueda secuencial busca por cada elemento de un arreglo en forma lineal**, hasta encontrar uno que coincida con la clave de búsqueda. Si se encuentra en el arreglo, devuelve el índice de ese elemento, de lo contrario, informa al usuario que la clave de búsqueda no está presente.

```
import java.util.Scanner;

public class JavaApplication1 {
      public static void main(String[] args) {
            Scanner entrada = new Scanner(System.in);
            int[] arreglo = {1, 5, 8, 0, 6, -3, -7, 9, 14, 6, 1, 0, 4};
            int tam = arreglo.length;
            int datoABuscar;
            int posicionDelDatoABuscar = -1;
            System.out.print("Ingrese dato entero a buscar: ");
            datoABuscar = entrada.nextInt();
            /*INICIO DE BUSQUEDA SECUENCIAL*/
            for ( int i = 0; i < tam; i++ )
            {
                  if (arreglo[i] == datoABuscar)
                  {
                        posicionDelDatoABuscar = i;
                        break;
                  }
            }
            /*FIN DE BUSQUEDA SECUENCIAL*/
            if (posicionDelDatoABuscar == -1)
            {
                  System.out.println("No se encontró al " + datoABuscar);
            }
            else
            {
                  System.out.print("El " + datoABuscar + " se encuentra en ");
                  System.out.println("la posición " + posicionDelDatoABuscar);
            }
      }
}
```

Código 30: Búsqueda secuencial

Búsqueda binaria

El algoritmo de **búsqueda binaria es más eficiente que** el algoritmo de **búsqueda secuencial**, pero **requiere que el arreglo esté ordenado**. La primera iteración de este algoritmo evalúa el elemento medio del arreglo. Si éste coincide con la clave de búsqueda, el algoritmo termina. Suponiendo que el arreglo se ordene en forma ascendente, entonces si la clave de búsqueda es menor que el elemento de en medio, continúa sólo con la primera mitad. En cambio, si la clave de búsqueda es mayor que el elemento de en medio, continúa sólo con la segunda mitad del arreglo. Cada iteración evalúa el valor medio de la porción restante del arreglo. Si la clave de búsqueda no coincide con el elemento, el algoritmo elimina la mitad de los elementos restantes. Para terminar, el algoritmo encuentra un elemento que coincide con la clave de búsqueda o reduce el subarreglo hasta un tamaño de cero.

```
import java.util.Scanner;

public class JavaApplication1 {
        public static void main(String[] args) {
                Scanner entrada = new Scanner(System.in);
                int[] arreglo = {1, 5, 8, 0, 6, -3, -7, 9, 14, 6, 1, 0, 4};
                int tam = arreglo.length;
                int datoABuscar;
                int posicionDelDatoABuscar = -1;
                System.out.print("Ingrese dato entero a buscar: ");
                datoABuscar = entrada.nextInt();
                /*INICIO DE BUSQUEDA BINARIA*/
                int inferior = 0;
                int superior = tam - 1;
                int medio = (inferior + superior) / 2;
                do
                {
                        if (datoABuscar == arreglo[medio])
                        {
                                posicionDelDatoABuscar = medio;
                        }
                        else if (datoABuscar < arreglo[medio])
                        {
                                superior = medio - 1;
                        }
                        else
                        {
                                inferior = medio + 1;
                        }
                        medio = (inferior + superior) / 2;
                } while ( (inferior <= superior) && (posicionDelDatoABuscar == -1) );
                /*FIN DE BUSQUEDA BINARIA*/
                if (posicionDelDatoABuscar == -1)
                {
                        System.out.println("No se encontró al " + datoABuscar);
                }
                else
                {
                        System.out.print("El " + datoABuscar + " se encuentra en ");
                        System.out.println("la posición " + posicionDelDatoABuscar);
                }
        }
}
```

Código 31: Búsqueda binaria

Arreglos multidimensionales

Definición

Un arreglo multidimensional es aquel en donde los datos pueden estar distribuidos en estructuras de más de una dimensión.

Un arreglo bidimensional es aquel que contiene datos distribuidos en dos dimensiones, normalmente separados por filas y por columnas. ¿Se nos ocurre dónde se podría utilizar esta estructura? Por ejemplo, para representar una tabla, un juego de ajedrez o batalla naval, un mapa de coordenadas (latitud y longitud), etc.

Un arreglo tridimensional es aquel que contiene datos distribuidos en tres dimensiones, normalmente separados por filas, columnas y profundidad. ¿Se nos ocurre dónde se podría utilizar esta estructura? Por ejemplo, para representar un cubo mágico.

Los arreglos bidimensionales son los más frecuentemente utilizados, también llamados **matrices**. Para identificar un elemento específico de una matriz, debemos especificar dos subíndices. Por convención, el primero identifica la fila del elemento y el segundo su columna.

Java no soporta los arreglos multidimensionales directamente, pero permite al programador especificar arreglos unidimensionales, cuyos elementos sean también arreglos unidimensionales, con lo cual se obtiene el mismo efecto.

Todos los ejemplos de este libro tratarán sobre matrices (arreglos bidimensionales).

Declaración

Sin inicializar

Para declarar un objeto arreglo de arreglos (matriz), debemos especificar el tipo de dato que alojará junto a dos corchetes de apertura y cierre `[][]` y luego el nombre de la matriz (siguiendo el mismo criterio de nomenclatura explicado en la sección "Nombres de variables").

El siguiente código declara una matriz de enteros llamada `numeros`.

```
public class JavaApplication1 {
     public static void main(String[] args) {
          int[][] numeros;
     }
}
```

Código 32: Declaración de una matriz sin inicializar

Por ahora, `numeros` apunta a una referencia nula. Todavía no tenemos creada la matriz.

Inicializada por defecto

Podemos declarar y crear la matriz en la misma sentencia (mismo criterio que con las variables de tipos primitivos).

Ahora a `numeros` le asignaremos (mediante el operador de asignación `=`) una nueva matriz de **3** x **4** elementos (3 filas y 4 columnas).

Para ello, utilizamos la palabra clave `new`, seguido del tipo de dato del arreglo, y entre los primeros corchetes de apertura y cierre `[]` irá el número de filas de los que consta la matriz. Entre los siguientes corchetes de apertura y cierre `[]` irá el número de columnas de los que consta la matriz.

El siguiente código declara una matriz de enteros llamada `numeros` que consta de **3** x **4** elementos (3 filas y 4 columnas).

```
public class JavaApplication1 {
      public static void main(String[] args) {
            int[][] numeros = new int[3][4];
      }
}
```

Código 33: Declaración de una matriz inicializada con valores por defecto

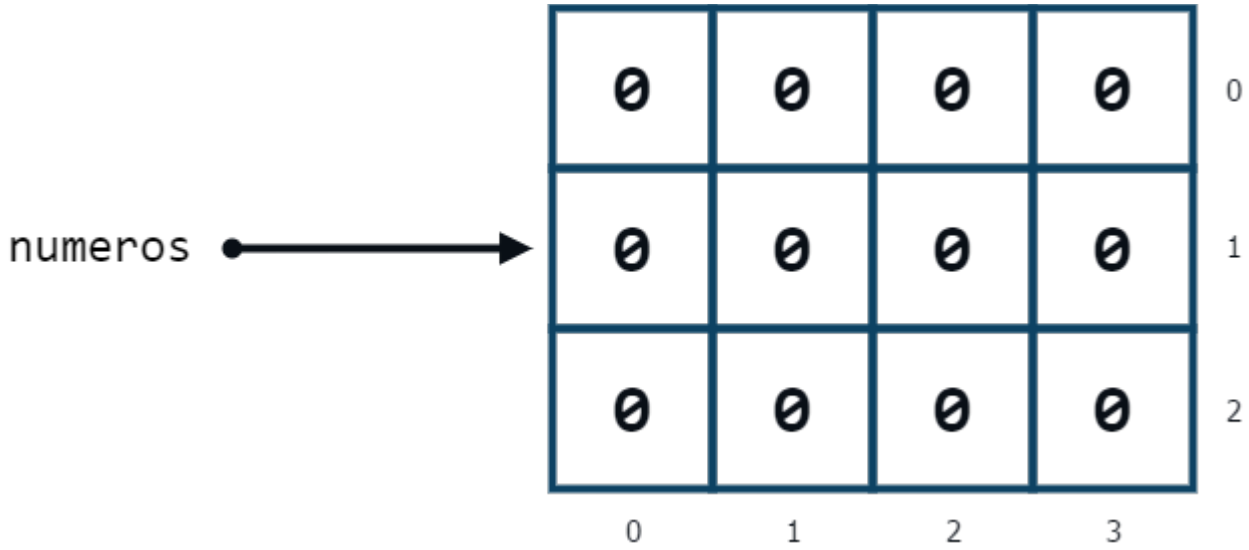

Ilustración 6: Matriz inicializada con valores por defecto

Tenemos entonces una matriz de enteros llamada `numeros` de tamaño **3** x **4**. Pero, ¿qué pasa con cada elemento guardado en el arreglo?

La respuesta depende del tipo de dato guardado. Los valores numéricos (`char`, `byte`, `short`, `int`, `long`, `float` y `double`) se iniciarán con el valor por defecto `0`. Los valores booleanos (`boolean`) se iniciarán con el valor por defecto `false` y cualquier **dato no primitivo** iniciará con valor por defecto `null`.

Inicializada con valores

Podemos declarar e inicializar la matriz con nuestros valores elegidos.

Para ello, encerramos nuestros valores separados por comas `,` entre llaves de apertura y cierre `{}`. Estamos creando una matriz de **3** x **4**, por lo tanto, los valores del arreglo son en realidad 3 arreglos de 4 elementos cada uno. Usamos el mismo criterio.

El siguiente código declara una matriz de enteros llamada `numeros` inicializada con los valores **7**, **4**, **9**, **-3** para la primer fila, el **8, 6, -4, 0** para la segunda fila y el **-1, -3, 9, 6** para la tercera fila.

```
public class JavaApplication1 {
      public static void main(String[] args) {
            int[][] numeros = { {7, 4, 9, -3} , {8, 6, -4, 0} , {-1, -3, 9, 6} };
      }
}
```

Código 34: Declaración de una matriz inicializada con valores definidos

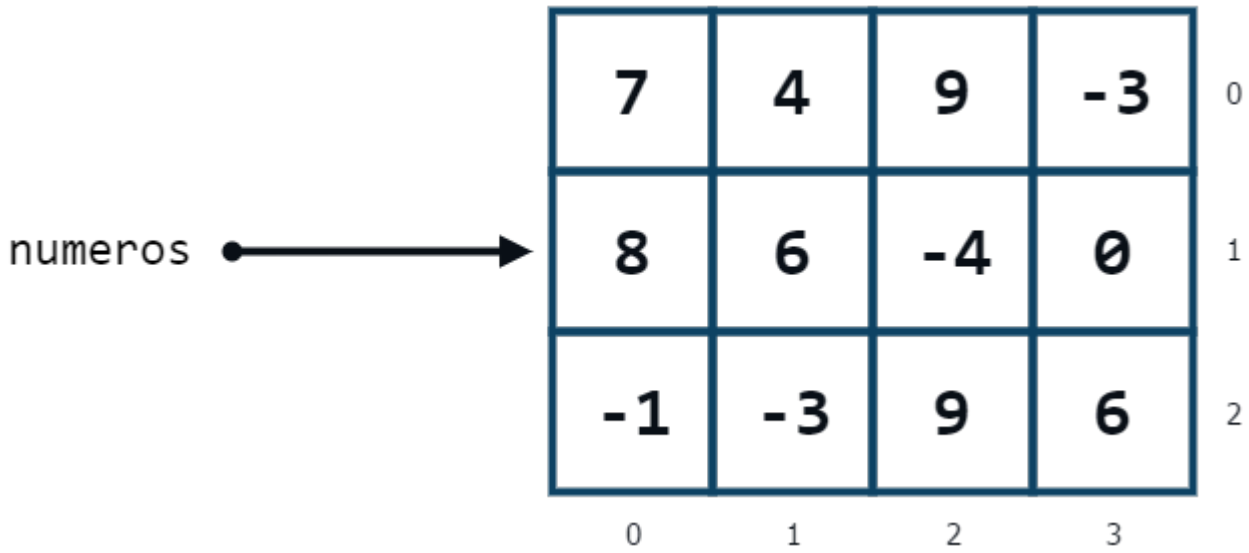

Ilustración 7: Matriz inicializada con valores definidos

Conocer la longitud de una matriz

Todo arreglo cuenta con el atributo `length` que permite conocer el número de elementos de los que consta. En este caso, no es posible conocer el número de elementos de la matriz usando `length`, pues `numeros.length` devolvería **3** (recordar que una matriz es un arreglo de arreglos) y nosotros esperábamos el valor **12** (3 por 4).

Una manera posible sería obtener la longitud del arreglo guardado en la primera posición de la matriz y multiplicarlo por la longitud de la misma.

```
longitudMatriz = numeros.length * numeros[0].length;
```

Obtener un valor de una matriz

Es posible obtener cualquier valor de una matriz.

Para ello, se ingresa el nombre de la matriz en cuestión seguido de dos corchetes de apertura y cierre `[][]`. Dentro de los primeros corchetes ingresamos el número de índice de fila donde se encuentra el dato que nos interesa obtener y dentro de los segundos corchetes ingresamos el número de índice de columna donde se encuentra el dato que nos interesa obtener.

```
public class JavaApplication1 {
	public static void main(String[] args) {
		int[][] numeros = { {7, 4, 9, -3} , {8, 6, -4, 0} , {-1, -3, 9, 6} };
		int x = numeros[0][3];
		System.out.println("El valor de la 1ra fila y 4ta columna es " + x);
		System.out.println("El valor de la 2da fila y 2da columna es " + numeros[1][1]);
	}
}
```

Código 35: Obtener un valor de una matriz

```
run:
El valor de la 1ra fila y 4ta columna es -3
El valor de la 2da fila y 2da columna es 6
BUILD SUCCESSFUL (total time: 0 seconds)
```

Salida 16: Obtener un valor de una matriz

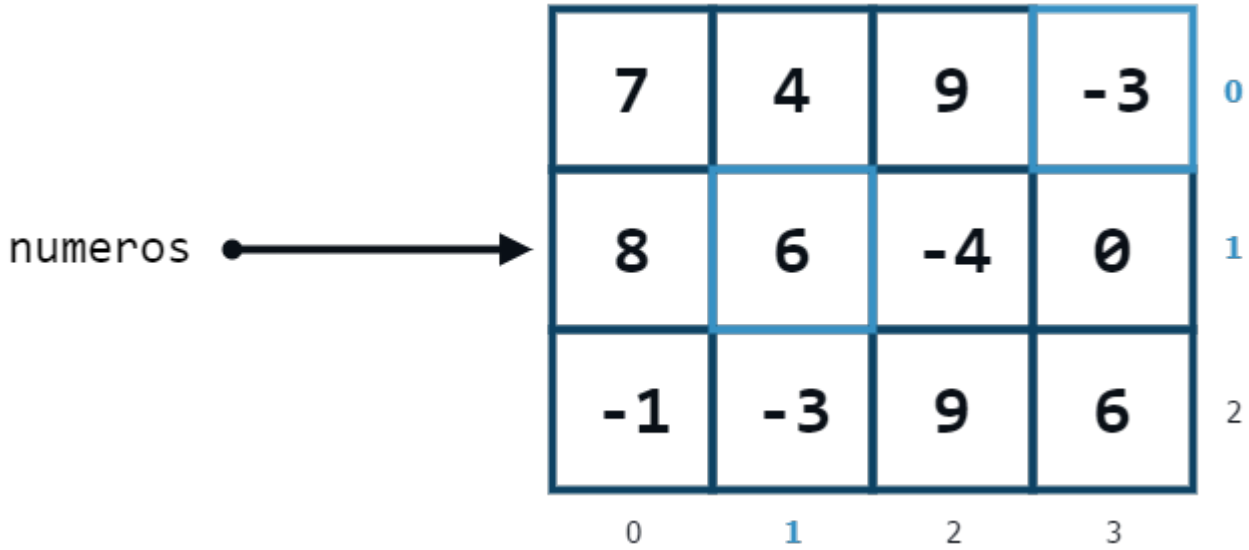

Ilustración 8: Obtener un valor de una matriz

Reemplazar un valor de una matriz

Es posible obtener cualquier valor de una matriz.

Para ello, se ingresa el nombre de la matriz en cuestión seguido de dos corchetes de apertura y cierre `[][]`. Dentro de los primeros corchetes ingresamos el número de índice de fila donde se encuentra el dato que nos interesa reemplazar y dentro de los segundos corchetes ingresamos el número de índice de columna donde se encuentra el dato que nos interesa reemplazar.

Ahora podemos guardar un nuevo valor gracias al operador de asignación `=`.

```
public class JavaApplication1 {
        public static void main(String[] args) {
                int[][] numeros = { {7, 4, 9, -3} , {8, 6, -4, 0} , {-1, -3, 9, 6} };
                System.out.println("El valor en la 2da fila y 2da columna es " + numeros[1][1]);
                numeros[1][1] = 15;
                System.out.println("El nuevo valor en la 2da fila y 2da columna es " + numeros[1][1]);
        }
}
```

Código 36: Reemplazar un valor de una matriz

```
run:
El valor en la 2da fila y 2da columna es 6
El nuevo valor en la 2da fila y 2da columna es 15
BUILD SUCCESSFUL (total time: 0 seconds)
```

Salida 17: Reemplazar un valor de una matriz

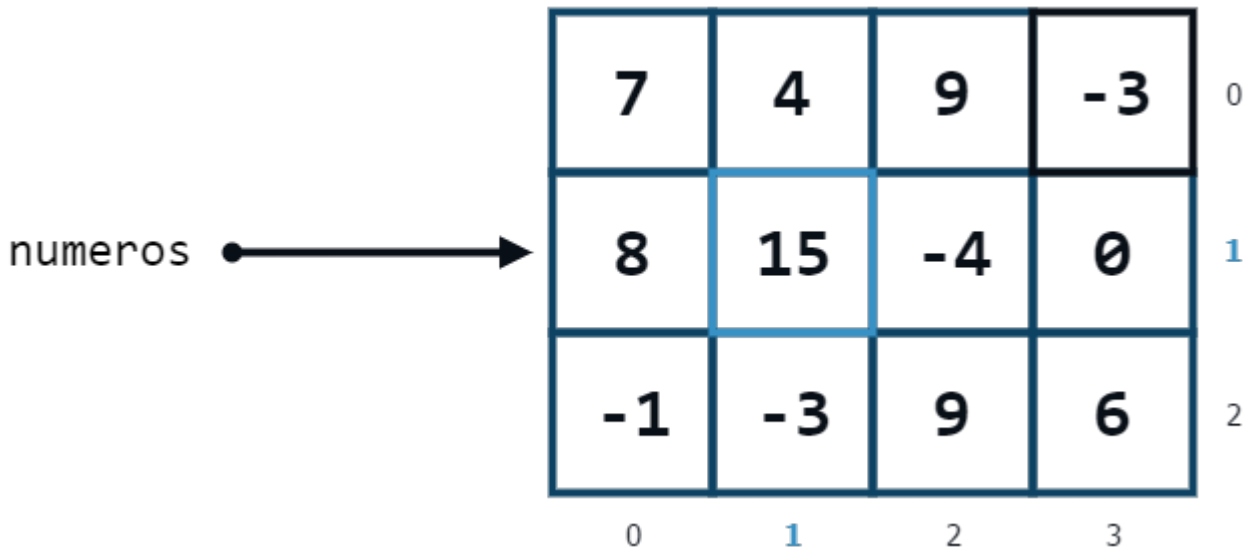

Ilustración 9: Reemplazar valores de una matriz

Mostrar todos los valores de una matriz

Es posible mostrar todos los valores de cualquier matriz.

Para ello, hacemos uso de una estructura de repetición `for` que itere en base a las columnas dentro de otra que itere en base a las filas.

```
public class JavaApplication1 {
	public static void main(String[] args) {
		int[][] numeros = { {7, 4, 9, -3} , {8, 6, -4, 0} , {-1, -3, 9, 6} };
		for (int f = 0; f < numeros.length; f++)
		{
			for (int c = 0; c < numeros[0].length; c++)
			{
				System.out.print(numeros[f][c] + "\t");
			}
			// Espacio en blanco para la siguiente fila
			System.out.println("");
		}
	}
}
```

Código 37: Mostrar todos los valores de una matriz

```
run:
7	4	9	-3
8	6	-4	0
-1	-3	9	6
BUILD SUCCESSFUL (total time: 0 seconds)
```

Salida 18: Mostrar todos los valores de una matriz

Utilidades

Tabla de caracteres ASCII imprimibles

Decimal	Símbolo	Decimal	Símbolo	Decimal	Símbolo
32	(espacio)	64	@	96	`
33	!	65	A	97	a
34	"	66	B	98	b
35	#	67	C	99	c
36	$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	'	71	G	103	g
40	(	72	H	104	h
41	)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[	123	{
60	<	92	\	124	|
61	=	93	]	125	}
62	>	94	^	126	~
63	?	95	_		

Bibliografía y referencias

- http://www.alegsa.com.ar/Dic/variable.php
- https://es.wikibooks.org/wiki/Programaci%C3%B3n_en_Java/Variables
- https://es.wikipedia.org/wiki/CamelCase
- http://tutorialdejava.blogspot.com.ar/2012/09/blog-post.html
- http://puntocomnoesunlenguaje.blogspot.com.ar/2012/08/java-scanner.html
- http://puntocomnoesunlenguaje.blogspot.com.ar/2012/04/operadores.html
- https://www.arkaitzgarro.com/java/capitulo-3.html
- http://iutprogramacion.blogspot.com.ar/2013/02/metodos-de-ordenamiento.html
- https://www.toptal.com/developers/sorting-algorithms