# Universidad Nacional Autónoma de México
# Facultad de Ingeniería

**Asignatura**: Estructura de Datos y Algoritmos I

**Actividad 1**: Acordeón del lenguaje C y de un lenguaje con la inicial de nuestro nombre.

**Alumno**: Miranda González José Francisco

**Fecha**: Viernes 26 de Febrero del 2021

# Acordeón del lenguaje C

## Fundamentos de Lenguaje C

### Comentarios

El comentario por línea inicia cuando se insertan los símbolos '//' y termina con el salto de línea.
El comentario por bloque inicia cuando se insertan los símbolos '/*' y termina cuando se encuentran los símbolos '*/'.

### Bibliotecas

Al iniciar el programa se deben agregar todas las bibliotecas que se van a utilizar en el mismo, es decir, funciones externas necesarias para ejecutar el programa. En lenguaje C la biblioteca estándar de entrada y salida está definida en 'stdio.h' (standard in out) y provee, entre otras, funciones para lectura y escritura de datos que se verán a continuación.

### Declaración de variables

sintaxis:
[modificadores] tipoDeDato identificador [= valor];

También es posible declarar varios identificadores de un mismo tipo de dato e inicializarlos en el mismo renglón, lo único que se tiene que hacer es separar cada identificador por comas.
tipoDeDato identificador1[= valor], identificador2[= valor];

Tipos de datos

Los tipos de datos básicos en C son:
Caracteres: codificación definida por la máquina.
Enteros: números sin punto decimal.
Flotantes: números reales de precisión normal.
Dobles: números reales de doble precisión.

Las variables enteras que existen en lenguaje C son:
Tipo:
signed char
unsigned char
signed short
unsigned short
signed int
unsigned int

signed long
unsigned long
enum

Si se omite el clasificador por defecto se considera 'signed'.

Las variables reales que existen en lenguaje C son:
Tipo :
float
double
long double

Las variables reales siempre poseen signo.

Para poder acceder al valor de una variable se requiere especificar el tipo de dato. Los especificadores que tiene lenguaje C para los diferentes tipos de datos son:

| Tipo de dato | Especificador de formato |
|---|---|
| Entero | %d, %i, %ld, %li, %o, %x |
| Flotante | %f, %lf, %e, %g |
| Carácter | %c, %d, %i, %o, %x |
| Cadena de caracteres | %s |

**Identificador**

Un identificador es el nombre con el que se va a almacenar en memoria un tipo de dato.
Debe iniciar con una letra [a-z].
Puede contener letras [A-Z, a-z], números [0-9] y el carácter guión bajo (_).

**printf** es una función para imprimir con formato, es decir, se tiene que especificar entre comillas el tipo de dato que se desea imprimir, también se puede combinar la impresión de un texto predeterminado:

printf("El valor de la variable real es: %lf", varReal);

**scanf** es una función que sirve para leer datos de la entrada estándar (teclado), para ello únicamente se especifica el tipo de dato que se desea leer entre comillas y en qué variable se quiere almacenar. Al nombre de la variable le antecede un ampersand (&), esto indica que el dato recibido se guardará en la localidad de memoria asignada a esa variable.

scanf ("%i", &varEntera);

Para imprimir con formato también se utilizan algunas secuencias de caracteres de escape, C maneja los siguientes:

\a carácter de alarma
\b retroceso
\f avance de hoja
\n salto de línea
\r regreso de carro
\t tabulador horizontal
\v tabulador vertical
'\0' carácter nulo

**Modificadores de alcance**

Los modificadores que se pueden agregar al inicio de la declaración de variables son const y static.
El modificador const impide que una variable cambie su valor durante la ejecución del programa, es decir, permite para crear constantes. Por convención, las constantes se escriben con mayúsculas y se deben inicializar al momento de declararse.
El modificador static indica que la variable permanece en memoria desde su creación y durante toda la ejecución del programa, es decir, permanece estática en la memoria.
Ejemplo

**Operadores**

Los operadores aritméticos que maneja lenguaje C son:

| Operador | Operación |
|---|---|
| + | Suma |
| - | Resta |
| * | Multiplicación |
| / | División |
| % | Módulo |

Los operadores lógicos a nivel de bits que maneja el lenguaje C son:

| Operador | Operación |
|---|---|
| >> | Corrimiento a la derecha |
| << | Corrimiento a la izquierda |
| & | Operador AND |
| \| | Operador OR |
| ~ | Complemento ar-1 |

**Expresiones lógicas**

Los operadores de relación permiten comparar elementos numéricos,

alfanuméricos, constantes o variables.

| Operador | Operación |
|---|---|
| == | Igual que |
| != | Diferente a |
| < | Menor que |
| > | Mayor que |
| < = | Menor o igual |
| > = | Mayor o igual |

Los operadores lógicos permiten formular condiciones complejas a partir de condiciones simples

| Operador | Operación |
|---|---|
| ! | No |
| && | Y |
| \|\| | O |

Lenguaje C posee operadores para realizar incrementos y decrementos de un número.
El operador ++ agrega una unidad (1) a su operando.
El operador -- resta una unidad (1) a su operando.

## Estructuras de selección

### Estructura de control selectiva if

Sintaxis:

```
if (expresión_lógica) {
   // bloque de código a ejecutar
}
```

En esta estructura se evalúa la expresión lógica y, si se cumple (si la condición es verdadera), se ejecutan las instrucciones del bloque que se encuentra entre las llaves de la estructura. Si no se cumple la condición, se continúa con el flujo normal del programa.

### Estructura de control selectiva if-else

Sintaxis:

```
if (expresión_lógica) {
    // bloque de código a ejecutar
    // si la condición es verdadera
```

```
} else {
    // bloque de código a ejecutar
    // si la condición es falsa
}
```

Esta estructura evalúa la expresión lógica y si la condición es verdadera se ejecutan las instrucciones del bloque que se encuentra entre las primeras llaves, si la condición es falsa se ejecuta el bloque de código que está entre las llaves después de la palabra reservada 'else'. Al final de que se ejecute uno u otro código, se continúa con el flujo normal del programa.
Es posible anidar varias estructuras if-else, es decir, dentro de una estructura if-else tener una o varias estructuras if-else.

**Estructura de control selectiva switch-case**

Sintaxis:

```
switch (opcion_a_evaluar){
   case valor1:
            /* Código a ejecutar*/
   break;
   case valor2:
            /* Código a ejecutar*/
   break;
    …
   case valorN:
           /* Código a ejecutar*/
   break;
   default:
          /* Código a ejecutar*/
}
```

La estructura switch-case evalúa la variable que se encuentra entre paréntesis después de la palabra reservada switch y la compara con los valores constantes que posee cada caso (case). Los tipos de datos que puede evaluar esta estructura son enteros, caracteres y enumeraciones. Al final de cada caso se ejecuta la instrucción break, si se omite esta palabra reservada se ejecutaría el siguiente caso, es decir, se utiliza para indicar que el bloque de código a ejecutar ya terminó y poder así salir de la estructura.
Si la opción a evaluar no coincide dentro de algún caso, entonces se ejecuta el bloque por defecto (default). El bloque por defecto normalmente se escribe al final de la estructura, pero se puede escribir en cualquier otra parte. Si se escribe en alguna otra parte el bloque debe terminar con la palabra reservada break.

**Enumeración**

Existe otro tipo de dato constante conocido como enumeración. Una variable

enumerador se puede crear de la siguiente manera:
enum identificador {VALOR1, VALOR2, ... , VALORN};

Para crear una enumeración se utiliza la palabra reservada enum, seguida de un identificador (nombre) y, entre llaves se ingresan los nombres de los valores que puede tomar dicha enumeración, separando los valores por coma. Los valores son elementos enteros y constantes (por lo tanto se escriben con mayúsculas).

### Estructura de control selectiva condicional

La estructura condicional (también llamado operador ternario) permite realizar una comparación rápida. Su sintaxis es la siguiente:

Condición ? SiSeCumple : SiNoSeCumple

Consta de tres partes, una condición y dos acciones a seguir con base en la expresión condicional. Si la condición se cumple (es verdadera) se ejecuta la instrucción que se encuentra después del símbolo '?'; si la condición no se cumple (es falsa) se ejecuta la instrucción que se encuentra después del símbolo ':'.

## Estructuras de repetición

### Estructura de control repetitiva while

La estructura repetitiva (o iterativa) while primero valida la expresión lógica y si ésta se cumple (es verdadera) procede a ejecutar el bloque de instrucciones de la estructura, el cual está delimitado por las llaves {}. Si la condición no se cumple se continúa el flujo normal del programa sin ejecutar el bloque de la estructura, es decir, el bloque se puede ejecutar de cero a ene veces.

Sintaxis:

```
while (expresión_lógica) {
    // Bloque de código a repetir
    // mientras que la expresión
    // lógica sea verdadera.
}
```

### Estructura de control repetitiva do-while

do-while es una estructura cíclica que ejecuta el bloque de código que se encuentra dentro de las llaves y después valida la condición, es decir, el bloque de código se ejecuta de una a ene veces.

Sintaxis:

```
do {
      /*     Bloque de código que se ejecuta
      por lo menos una vez y se repite
      mientras la expresión lógica sea
      verdadera.
      */
} while (expresión_lógica);
```

**Estructura de control de repetición for**

Lenguaje C posee la estructura de repetición for la cual permite realizar repeticiones cuando se conoce el número de elementos que se quiere recorrer.

Sintaxis:

```
for (inicialización ; expresión_lógica ; operaciones por iteración) {
       /*
            Bloque de código
            a ejecutar
      */
}
```

La estructura for ejecuta 3 acciones básicas antes o después de ejecutar el bloque de código. La primera acción es la inicialización, en la cual se pueden definir variables e inicializar sus valores; esta parte solo se ejecuta una vez cuando se ingresa al ciclo y es opcional. La segunda acción consta de una expresión lógica, la cual se evalúa y, si ésta es verdadera, ejecuta el bloque de código, si no se cumple se continúa la ejecución del programa; esta parte es opcional. La tercera parte consta de un conjunto de operaciones que se realizan cada vez que termina de ejecutarse el bloque de código y antes de volver a validar la expresión lógica; esta parte también es opcional.

**Define**

define permite definir constantes o literales; se les nombra también como constantes simbólicas.

Sintaxis:

```
#define <nombre> <valor>
```

Al definir la constante simbólica con #define, se emplea un nombre y un valor. Cada vez que aparezca el nombre en el programa se cambiará por el valor definido. El valor puede ser numérico o puede ser texto.

**Break**

La proposición break proporciona una salida anticipada dentro de una estructura de repetición, tal como lo hace en un switch. Un break provoca que el ciclo que lo encierra termine inmediatamente.

**Continue**

La proposición continue provoca que inicie la siguiente iteración del ciclo de repetición que la contiene.

## Depuración de programas

La depuración de un programa es útil cuando:
Se desea optimizar el programa
El programa tiene algún fallo
El programa tiene un error de ejecución o defecto

Algunas funciones básicas que tienen en común la mayoría de los depuradores son:
Ejecutar el programa
Mostrar el código fuente del programa
Punto de ruptura
Continuar
Ejecutar la siguiente instrucción
Ejecutar la siguiente línea
Ejecutar la instrucción o línea anterior
Visualizar el valor de las variables

**Depuración de programas escritos en C con GCC y GDB**

Para depurar un programa usando las herramientas desarrolladas por GNU, éste debe compilarse con información para depuración por medio del compilador GCC.

Para compilar, por ejemplo, un programa llamado calculadora.c con GCC con información de depuración, debe realizarse en una terminal con el siguiente comando:
gcc -g -o calculadora calculadora.c

El parámetro -g es quien indica que el ejecutable debe producirse con información de depuración.

Una vez hecho el paso anterior, debe usarse la herramienta GDB, la cual, es el depurador para cualquier programa ejecutable realizado por GCC.

Para depurar un ejecutable debe invocarse a GDB en la terminal indicando cuál es el programa ejecutable a depurar, por ejemplo, para depurar calculadora:
gdb ./calculadora

Al correr GDB se entra a una línea de comandos. De acuerdo al comando es posible realizar distintas funciones de depuración:

list o l: Permite listar diez líneas del código fuente del programa, si se desea visualizar todo el código fuente debe invocarse varias veces este comando para mostrar de diez en diez líneas.

b: Establece un punto de ruptura para lo cual debe indicarse en qué línea se desea establecer o bien también acepta el nombre de la función donde se desea realizar dicho paso.

d o delete: Elimina un punto de ruptura, indicando cuál es el que debe eliminarse usando el número de línea.

clear: Elimina todos los puntos de ruptura.

info line: Permite mostrar información relativa a la línea que se indique después del comando.

run o r: Ejecuta el programa en cuestión.

c: Continúa con la ejecución del programa después de un punto de ruptura.

s: Continúa con la siguiente instrucción después de un punto de ruptura.

n: Salta hasta la siguiente línea de código después de un punto de ruptura.

p o print: Muestra el valor de una variable, para ello debe escribirse el comando y el nombre de la variable separados por un espacio.

ignore: Ignora un determinado punto de ruptura indicándolo con el número de línea de código.

q o quit: Termina la ejecución de GDB.

**Depuración de programas escritos en C con Dev-C++ 5.0.3.4**

Antes de iniciar la depuración debe tenerse a la mano el archivo con el programa escrito en C o proceder a escribirlo en la misma IDE. Para ello debe usarse el menú Archivo → Nuevo → Código Fuente si se piensa usar la IDE para escribirlo o en su lugar Archivo → Abrir Proyecto o Archivo si ya existía el código fuente.

Una vez que se tiene el programa, debe activarse la opción de compilación generando información para el depurador. Para activar esta opción debe abrirse el menú Herramientas → Opciones del Compilador y acceder a la pestaña Generación/Optimización de Código y finalmente, en la subpestaña Enlazador (linker), activar la opción Generar Información de Depuración:

Después de realizar lo anterior, el programa realizado puede compilarse y ejecutarse con lo que ofrece el menú Ejecutar.

Para agregar puntos de ruptura, debe hacerse clic en la línea de código donde se desea colocar y ésta se volverá en color rojo. Para retirarlo se hace clic de nuevo en la línea y volverá a su color normal.

Para depurar el programa, primero se debe compilar con el menú Ejecutar → Compilar y luego depurar con Depurar → Depurar. El programa se abrirá y se ejecutará hasta el primer punto de ruptura seleccionado.

Cuando se llega al punto de ruptura, se tienen diversas opciones, Siguiente Paso ejecuta la siguiente línea, Avanzar Paso a Paso ejecuta instrucción por instrucción, Saltar Paso ejecuta hasta el siguiente punto de ruptura.

Para detener la depuración puede seleccionarse la opción Parar ejecución. La opción Ver ventana del CPU permite ver a detalle las instrucciones enviadas al procesador, registros de memoria involucrados y valor de cada una de las banderas en el procesador.

Finamente, para estudiar el valor de cada variable, se puede recurrir a la función Añadir Watch y escribir el nombre de la variable.

**Depuración de programas escritos en C con Code::Blocks 13.12**

Para poder depurar, es necesario crear un nuevo proyecto desde el menú File → New → Project y elegir en el cuadro que aparece Console Application. Seguir el asistente para crear el proyecto eligiendo que se usará el lenguaje C, luego seleccionar un título adecuado para el proyecto, la ruta donde se creará y el título del archivo asociado al proyecto. Posteriormente, en el mismo asistente seleccionar Create “Debug” Configuration y Create “Release” Configuration en el mismo asistente con compilador GNU GCC Compiler. Nótese que existen dos carpetas asociadas que son /bin/debug y /bin/release que son donde se crearán los ejecutables de depuración y el final respectivamente. Se debe finalizar el asistente.

En la parte izquierda, se encontrará el nombre del archivo fuente del proyecto. Navegar y dar clic en main.c:

Dicho archivo debe editarse para formar el programa deseado. Cuando se está desarrollando es importante que esté seleccionado el modo de depuración

hay un menú despegable que tiene la opción Debug y Release, debe estar siempre seleccionada la primera opción hasta no haber terminado el programa a desarrollar.

Cuando todo esté listo, se cambiará a la segunda opción y se usará el archivo ejecutable que se localiza en la /bin/reléase, jamás el localizado en /bin/debug que solo tiene efectos de desarrollo.

La opción, Debug/Continue, permite ejecutar el programa en modo de depuración y reanudar la ejecución después de un punto de ruptura. La opción Stop debugger, detiene la depuración y permite continuar editando.

Para visualizar una variable, debe hacerse clic sobre ella con el depurador corriendo, y dar clic en Watch 'variable'.

Finalmente, para agregar un punto de ruptura, se tiene que hacer clic derecho sobre el número de línea de código y agregarlo, aparecerá un punto rojo en la línea, para quitarlo se tiene que hacer el mismo procedimiento.

La parte del código que se está ejecutando por el depurador, se indica por medio de una flecha amarilla en el número de línea correspondiente.

**Depuración de programas escritos en C con Xcode de Mac**

Abrir la aplicación Xcode, crear un nuevo proyecto seleccionando Create a new Xcode project; a lo cual aparecerá la siguiente pantalla donde se deberá seleccionar macOS y Command Line Tool, y dar Next.

Posteriormente dar las características del proyecto, seguido de Next.

Se presentará una ventana donde hay que indicar dónde se grabará el proyecto. Después dar Create.

Seleccionando main.c en la jerarquía de archivos, mostrado en el lado izquierdo, podremos observar que nos presenta un "esqueleto" de un programa en C, el cual se deberá sustituir por el que se vaya a depurar. Como Xcode es un IDE, podemos aquí mismo editar y compilar antes de iniciar las actividades de depuración. Para ello nos podemos auxiliar de la barra superior de menús de Xcode

Una vez que se tiene el programa a depurar editado, primeramente se recomienda compilarlo antes de realizar su depuración; esto se realiza usando las teclas Cmd +B. Posteriormente para ejecutarlo, en la barra de trabajo utilizar (flecha).

Para iniciar con algunas actividades de depuración, vamos a indicar puntos de ruptura. Esto se realiza haciendo clic, con el botón derecho del mouse, sobre el número que indica la línea de código donde se desea detener y aparecerá una

flecha azul

Una vez definidos los puntos de ruptura, al momento de ejecutar el programa se detendrá al encontrar el primer punto de ruptura, indicando automáticamente el valor de todas las variables definidas en el programa hasta ese punto.

La continuidad y control de la ejecución del programa, se realiza a través de los íconos de la barra de resultados:


Desactiva todos los puntos de ruptura


Continúa la ejecución del programa hasta la siguiente línea de ruptura


Ejecuta sólo la siguiente instrucción y se detiene


Sigue ejecutano el programa hasta la siguiente línea de ruptura

## Arreglos unidimensionales y multidimensionales

### Arreglos unidimensionales

La sintaxis para definir un arreglo en lenguaje C es la siguiente:
tipoDeDato nombre[tamaño]

Donde nombre se refiere al identificador del arreglo, tamaño es un número entero y define el número máximo de elementos que puede contener el arreglo. Un arreglo puede ser de los tipos de dato entero, real, carácter o estructura.

### Apuntadores

La sintaxis para declarar un apuntador y para asignarle la dirección de memoria de otra variable es, respectivamente:
TipoDeDato *apuntador, variable;
apuntador = &variable;

La declaración de una variable apuntador inicia con el carácter *. Cuando a una variable le antecede un ampersand, lo que se hace es acceder a la dirección de memoria de la misma.

Los apuntadores solo pueden apuntar a direcciones de memoria del mismo tipo de dato con el que fueron declarados; para acceder al contenido de dicha dirección, a

la variable apuntador se le antepone *.

Un apuntador almacena la dirección de memoria de la variable a la que apunta.

### Arreglos multidimensionales

Lenguaje C permite crear arreglos de varias dimensiones con la siguiente sintaxis: tipoDato nombre[ tamaño ][ tamaño ]…[tamaño];

Donde nombre se refiere al identificador del arreglo, tamaño es un número entero y define el número máximo de elementos que puede contener el arreglo por dimensión. Los tipos de dato que puede tolerar un arreglo multidimensional son: entero, real, carácter o estructura.

De manera práctica se puede considerar que la primera dimensión corresponde a los renglones, la segunda a las columnas, la tercera al plano, y así sucesivamente. Sin embargo, en la memoria cada elemento del arreglo se guarda de forma contigua, por lo tanto, se puede recorrer un arreglo multidimensional con apuntadores.

## Funciones

La sintaxis básica para definir una función es la siguiente:

```
valorRetorno nombre (parámetros){
      // bloque de código de la función
}
```

Una función puede recibir parámetros de entrada, los cuales son datos de entrada con los que trabajará la función, dichos parámetros se deben definir dentro de los paréntesis de la función, separados por comas e indicando su tipo de dato, de la siguiente forma:

(tipoDato nom1, tipoDato nom2, tipoDato nom3…)

El tipo de dato puede ser cualquiera de los vistos hasta el momento y el nombre debe seguir la notación de camello.

El valor de retorno de una función indica el tipo de dato que va a regresar la función al terminar el bloque de código de la misma. El valor de retorno puede ser cualquiera de los tipos de datos vistos hasta el momento (entero, real, carácter o arreglo), aunque también se puede regresar el elemento vacío (void).

Una declaración, prototipo o firma de una función tiene la siguiente sintaxis:

valorRetorno nombre (parámetros);

La firma de una función está compuesta por tres elementos: el nombre de la función, los parámetros que recibe la función y el valor de retorno de la función; finaliza con punto y coma (;).

**Ámbito o alcance de las variables**

Las variables que se declaren dentro de cada función se conocen como variables locales (a cada función). Estas variables existen al momento de que la función es llamada y desaparecen cuando la función llega a su fin.

Las variables que se declaran fuera de cualquier función se llaman variables globales. Las variables globales existen durante la ejecución de todo el programa y pueden ser utilizadas por cualquier función.

**Argumentos para la función main**

La función main también puede recibir parámetros. Debido a que la función main es la primera que se ejecuta en un programa, los parámetros de la función hay que enviarlos al ejecutar el programa. La firma completa de la función main es:

int main (int argc, char ** argv);

La función main puede recibir como parámetro de entrada un arreglo de cadenas al ejecutar el programa. La longitud del arreglo se guarda en el primer parámetro (argument counter) y el arreglo de cadenas se guarda en el segundo parámetro (argument vector). Para enviar parámetros, el programa se debe ejecutar de la siguiente manera:

En plataforma Linux/Unix
./nombrePrograma arg1 arg2 arg3 ...

En plataforma Windows
nombrePrograma.exe arg1 arg2 arg3 ...

Esto es, el nombre del programa seguido de los argumentos de entrada. Estos argumentos son leídos como cadenas de caracteres dentro del argument vector, donde en la posición 0 se encuentra el nombre del programa, en la posición 1 el primer argumento, en la posición 2 el segundo argumento y así sucesivamente.

**Estático**

La sintaxis para declarar elementos estáticos es la siguiente:

static tipoDato nombre;
static valorRetorno nombre(parámetros);

Es decir, tanto a la declaración de una variable como a la firma de una función solo se le agrega la palabra reservada static al inicio de las mismas.

El atributo static en una variable hace que ésta permanezca en memoria desde su creación y durante toda la ejecución del programa, lo que quiere decir que su valor se mantendrá hasta que el programa llegue a su fin.

El atributo static en una función hace que esa función sea accesible solo dentro del mismo archivo, lo que impide que fuera de la unidad de compilación se pueda acceder a la función.

## Lectura y escritura de datos

### Apuntador a archivo

Los apuntadores a un archivo se manejan en lenguaje C como variables apuntador de tipo FILE que se define en la cabecera stdio.h. La sintaxis para obtener una variable apuntador de archivo es la siguiente:

FILE *F;

### Abrir archivo

La función fopen() abre una secuencia para que pueda ser utilizada y la asocia a un archivo. Su estructura es la siguiente:

*FILE fopen(char *nombre_archivo, char *modo);

Donde nombre_archivo es un puntero a una cadena de caracteres que representan un nombre válido del archivo y puede incluir una especificación del directorio. La cadena a la que apunta modo determina cómo se abre el archivo.

Existen diferentes modos de apertura de archivos:

r: Abre un archivo de texto para lectura.
w: Crea un archivo de texto para escritura.
a: Abre un archivo de texto para añadir.
r+: Abre un archivo de texto para lectura / escritura.
w+: Crea un archivo de texto para lectura / escritura.
a+: Añade o crea un archivo de texto para lectura / escritura.
rb: Abre un archivo en modo lectura y binario.
wb: Crea un archivo en modo escritura y binario.

**Cerrar archivo**

La función fclose() cierra una secuencia que fue abierta mediante una llamada a fopen(). Escribe la información que se encuentre en el buffer al disco y realiza un cierre formal del archivo a nivel del sistema operativo.

La firma de esta función es:
int fclose(FILE *apArch);

Donde apArch es el apuntador al archivo devuelto por la llamada a fopen(). Si se devuelve un valor cero significa que la operación de cierre ha tenido éxito.

**Funciones fgets y fputs**

Las funciones fgets() y fputs() pueden leer y escribir, respectivamente, cadenas sobre los archivos. Las firmas de estas funciones son, respectivamente:

char *fgets(char *buffer, int tamaño, FILE *apArch);
char *fputs(char *buffer, FILE *apArch);

La función fputs() permite escribir una cadena en un archivo especifico. La función fgets() permite leer una cadena desde el archivo especificado. Esta función lee un renglón a la vez.

**Funciones fscanf y fprintf**

Las funciones fprintf() y fscanf() se comportan exactamente como printf() (imprimir) y scanf() (leer), excepto que operan sobre archivo. Sus estructuras son:
int fprintf(FILE *apArch, char *formato, ...);
int fscanf(FILE *apArch, char *formato, ...);

Donde apArch es un apuntador al archivo devuelto por una llamada a la función fopen(), es decir, fprintf() y fscanf() dirigen sus operaciones de E/S al archivo al que apunta apArch. formato es una cadena que puede incluir texto o especificadores de impresión de variables. En los puntos suspensivos se agregan las variables (si es que existen) cuyos valores se quieren escribir en el archivo.

**Funciones fread y fwrite**

fread y fwrite son funciones que permiten trabajar con elementos de longitud conocida. fread permite leer uno o varios elementos de la misma longitud a partir de una dirección de memoria determinada (apuntador).

El valor de retorno es el número de elementos (bytes) leídos. Su sintaxis es la siguiente:

int fread(void *ap, size_t tam, size_t nelem, FILE *archivo)

fwrite permite escribir hacia un archivo uno o varios elementos de la misma longitud almacenados a partir de una dirección de memoria determinada.

El valor de retorno es el número de elementos escritos. Su sintaxis es la siguiente:

int fwrite(void *ap, size_t tam, size_t nelem, FILE *archivo).

# Acordeón de lenguaje con la inicial de mi nombre

# José – Java

## Conceptos Básicos en Java

**Objeto**:
Es un elemento de software que intenta representar un objeto del mundo real. De esta forma un objeto tendrá sus propiedades y acciones a realizar con el objeto. Estas propiedades y acciones están encapsuladas dentro del objeto, cumpliendo así los principios de encapsulamiento.

**Clase**:
Las clases representan los prototipos de los objetos que tenemos en el mundo real. Es decir, es una generalización de un conjunto de objetos. A su vez los objetos serán instancias de una determinada clase.
En la clase es dónde realmente definimos las propiedades y métodos que podrán contener cada una de las instancias de los objetos.

**Paquete**:
Un paquete es una forma de organizar elementos de software mediante un espacio de nombres. Así podremos afrontar desarrollos grandes de software facilitando la forma de encontrar o referirnos a un elemento.
Podríamos entender el sistema de paquetes como si fuese un sistema de carpetas. De tal manera que colocaremos cada una de las clases (o ficheros) en un paquete (o directorio).
Los paquetes se definen mediante el modficiador package seguido del nombre del paquete. El paquete lo definiremos en la primera línea de cada una de las clases.

**Interface**:
Un interface es una forma de establecer un contrato entre dos elementos. Un interface indica qué acciones son las que una determinada clase nos va a ofrecer cuando vayamos a utilizarla.
Cuando implementemos un interface deberemos de implementar todas las acciones que este contenga.

**Herencia**:
La herencia es una forma de estructurar el software. Mediante la herencia podemos indicar que una clase hereda de otra. Es decir la clase extiende las capacidades (propiedades y métodos) que tenga y añade nuevas propiedades y acciones.
Digamos que las nuevas clases especializan más aquellas clases de las que heredan al añadir nueva funcionalidad. Aunque también pueden reescribir el funcionamiento de dichos elementos.

## Variables en Java

Las variables Java son un espacio de memoria en el que guardamos un determinado valor (o dato). Para definir una variable seguiremos la estructura:

```
[privacidad] tipo_variable identificador;
```

Ejemplos de variables serían…

```
int numero = 2;
String cadena = "Hola";
long decimal = 2.4;
boolean flag = true;
```

Las variables son utilizadas como propiedades dentro de los objetos.

```
class Triangulo {
    private long base;
    private long altura;
}
```

## Tipos de variables en Java

**Variables de instancia (campos no estáticos)**, son las variables que están definidas dentro de un objeto pero que no tienen un modificador de estáticas (static). Suelen llevar un modificador de visibilidad (public, private, protected) definiéndose.

**Variables de clase (campos estáticos),** son aquellas variables que están precedidas del modificador static. Esto indica que solo hay una instancia de dicha variable.

**Variables locales**, son variables temporales cuyo ámbito de visibilidad es el método sobre el que están definidas. No pueden ser accedidas desde otra parte del código. Se las distingue de las variables de instancia ya que estas no llevan modificadores de visibilidad delante.

**Parámetros**, son las variables recibidas como parámetros de los métodos. Su visibilidad será el código que contenga dicho método.

## Tipos de Datos Primitivos en Java

El lenguaje Java da de base una serie de tipos de datos primitivos.

**byte**
Representa un tipo de dato de 8 bits con signo. De tal manera que puede almacenar los valores numéricos de -128 a 127 (ambos inclusive).

**short**
Representa un tipo de dato de 16 bits con signo. De esta manera almacena valores numéricos de -32.768 a 32.767.

**int**
Es un tipo de dato de 32 bits con signo para almacenar valores numéricos. Cuyo valor mínimo es -231 y el valor máximo 231-1.

**long**
Es un tipo de dato de 64 bits con signo que almacena valores numéricos entre -263 a 263-1

**float**
Es un tipo dato para almacenar números en coma flotante con precisión simple de 32 bits.

**double**
Es un tipo de dato para almacenar números en coma flotante con doble precisión de 64 bits.

**boolean**
Sirve para definir tipos de datos booleanos. Es decir, aquellos que tienen un valor de true o false. Ocupa 1 bit de información.

**char**
Es un tipo de datos que representa a un carácter Unicode sencillo de 16 bits.

## Literales en Java

Los valores literales son aquellos que podemos asignar a las variables. Dependiendo del tipo de variable podremos asignar unos valores u otros.

**Literales de enteros:**
Los enteros que podemos utilizar serán byte, short, int y long. Los literales que les asignemos siempre será un número entero.

**Literales de decimales:**

Los dos tipos de datos de decimales que podemos manejar son float y double. Para estos casos la representación del literal de decimales serán con separación de un punto entre la parte entera y la parte decimal.

**Literales de caracteres y cadenas:**
Tanto los caracteres del tipo de dato char, como las cadenas del tipo de datos String contienen caracteres Unicode UTF-16.
Los caracteres UTF-16 se pueden escribir directamente en la cadena o si nuestro editor de textos no nos permite el manejo de esa codificación los podemos poner escapados en el formato.

Además en las cadenas podemos utilizar una serie de secuencias de escape, las cuales empiezan por una barra invertida y siguen con un modificador:

| Secuencia | Significado |
|---|---|
| b | retroceso |
| t | tabular la cadena |
| n | salto de línea |
| f | form feed |
| r | retorno de carro |
| ' | comilla simple |
| " | comilla doble |
| \ | barra invertida |

**Literales subrayados:**
A partir de la versión 1.7 de Java se puede utilizar el subrayado para realizar separaciones entre números para una mejor visualización.
A todos los efectos el valor del número es como si no existiese el carácter de subrayado.

## Expresiones, sentencias y bloques en Java

**Expresiones**:
Una expresión es un conjunto de variables, operadores e invocaciones de métodos que se construyen para poder ser evaluadas retornando un resultado.

Ejemplos de expresiones son:

```
int valor = 1;
if (valor 1 > valor2) { ... }
```

**Sentencias**:
Una sentencia es la unidad mínima de ejecución de un programa. Un programa se compone de conjunto de sentencias que acaban resolviendo un problema. Al final de cada una de las sentencias encontraremos un punto y coma (;).

Tenemos los siguientes tipos de sentencias.

**Sentencias de declaración**
int valor = 2;

**Sentencias de asignación**
valor = 2;

**Sentencias de incremento o decremento**
valor++;

**Invocaciones a métodos**
System.out.println("Hola Mundo");

**Creaciones de objetos**
Circulo miCirculo = new Circulo(2,3);

**Sentencias de control de flujo**
if (valor>1) { … }

**Bloques:**
Un bloque es un conjunto de sentencias los cuales están delimitados por llaves

```
if (expresion) {
    // Bloque 1
} else {
    // Bloque 2
}
```

## Operadores de Asignación y Aritméticos Java

**Operador de Asignación:**
El operador Java más sencillo es el operador de asignación. Mediante este operador

se asigna un valor a una variable. El operador de asignación es el símbolo igual.

La estructura del operador de asignación es:

```
variable = valor;
```

Así podemos asignar valores a variables de tipo entero, cadena,…

**Operadores Aritméticos:**
Los operadores aritméticos en Java son los operadores que nos permiten realizar operaciones matemáticas: suma, resta, multiplicación, división y resto.

Los operadores aritméticos en Java son:

| Operador | Descripción |
|---|---|
| + | Operador de Suma. Concatena cadenas para la suma de String |
| - | Operador de Resta |
| * | Operador de Multiplicación |
| / | Operador de División |
| % | Operador de Resto |

## Operadores Unarios en Java

Los operadores unarios en Java son aquellos que solo requieren un operando para funcionar.

Los operadores unitarios que tenemos en Java son:

| Operador | Descripción |
|---|---|
| + | Operador unario suma. Indica un número positivo. |
| - | Operador unario resta. Niega una expresión. |
| ++ | Operador de incremento. Incrementa el valor en 1. |
| – | Operador de decremento. Decrementa el valor en 1. |
| ! | Operador de complemento lógico. Invierte el valor de un booleano |

**Operadores unarios suma o resta:**
Los operadores unitarios de suma o resta son muy sencillos de utilizar. En el caso del operador unitario suma su uso es redundante. Con el operador unitario resta podemos invertir un valor.
Operadores de incremento y decremento:
Los operadores de incremento se pueden aplicar como prefijo o como sufijo.

**Operador de complemento lógico:**
El operador de complemento lógico sirve para negar un valor lógico. Se suele utilizar delante de una operación de evaluación booleana. Normalmente en sentencias de decisión o bucles.

## Operadores Igualdad y Relacionales en Java

Los operadores de igualdad y relacionales en Java son aquellos que nos permiten comparar el contenido de una variable contra otra atendiendo a si son variables con un valor igual o distinto o bien si los valores son mayores o menores.

El listado de operadores de igualdad y relacionales en Java es:

| Operador | Descripción |
|---|---|
| == | igual a |
| != | no igual a |
| > | mayor que |
| >= | mayor o igual que |
| < | menor que |
| <= | menor o igual que |

**Operadores de Igualdad**
Mediante los operadores de igualdad podemos comprobar si dos valores son iguales (operador ==) o diferentes (operador !=).

**Operadores relacionales**
Permiten comprobar si un valor es mayor que (operador >), menor que (operador <), mayor o igual que (>=) y menor o igual que (<=).

## Operadores Condicionales Java

Los operadores condicionales en Java son aquellos que evalúan dos expresiones booleanas.

Dentro de los operadores condicionales en Java tenemos:

| Operador | Descripción |
|---|---|
| && | Operador condicional AND |
| \|\| | Operador condicional OR |
| ?: | Operador Ternario |
| instanceof | Operador instanceof |

**Operadores Condicionales:**
La estructura de los operadores condicionales en Java es:

```
(expresion_booleana1 && expresion_booleana2)
(expresion_booleana1 || expresion_booleana2)
```

En el caso del operador condicional AND el resultado será true siempre y cuando las dos expresiones evaluadas sean true. Si una de las expresiones es false el resultado de la expresión condicional AND será false.

Para el operador condicional OR el resultado será true siempre que alguna de las dos expresiones sea true.

**Operador Ternario:**
El operador ternario es otro de los operadores condicionales. Es una forma reducida de escribir un if-then-else. El operador ternario es representado mediante el símbolo ?:

La estructura del operador ternario es:

```
(expresion)?valor_true:valor_false;
```

En el caso de que la expresión tenga un valor de true se retorna el valor indicado después del cierre de interrogación (?) Y si la expresión tiene un valor de false se retorna el valor que esté después de los dos puntos (:).

**Operador instanceof:**
El operador instanceof es un operador especial para los objetos. Mediante el

operador instanceof podemos comprobar si un objeto es de una clase concreta.

La estructura del operador instanceof es:

```
objeto instanceof clase
```

El operador instanceof devolverá true siempre y cuando el objeto sea del tipo clase o de alguna de las clases de las que herede.

## Operadores de Bit Java

Estos nos permiten manejar números binarios. Los número binarios podremos operarlos de dos formas. Por un lado de forma que el cálculo se haga bit a bit, bien sea realizando operaciones de AND, OR,… Y por otro desplazamiento de bits.

| Operador | Descripción |
|---|---|
| & | Operador bits AND |
| ^ | Operador bits OR exclusivo (XOR) |
| \| | Operador bits OR inclusivo |
| ~ | Operador Negación Bits (NOT) |
| « | Operador desplazamiento izquierda |
| » | Operador desplazamiento derecha |
| »> | Operador desplazamiento derecha sin signo |

**Operadores de Bit a Bit**
La estructura de los operadores de bit a bit es la siguiente

```
(valor_binario1) & (valor_binario2)
(valor_binario1) ^ (valor_binario2)
(valor_binario1) | (valor_binario2)
```

**Operador AND**
El operador de bit AND lo que hace es multiplicar los bits de las dos cadenas. Las multiplicaciones de bits dan como resultado que 1 x 1 siempre es 1 y que 1 x 0 y 0 x 0 siempre da 0. El tamaño de la cadena resultado siempre es el mismo al

tamaño de las cadenas.

En el caso del **OR inclusivo (operador )** lo que se hace es multiplicar los bits asumiendo que 1 multiplicado por 1 o por 0, siempre es 1. Mientras que la única multiplicación que da 0 es 0 x 0.

**Operador XOR**
Para el caso del OR exclusivo (operador ^) conocido como XOR. La multiplicación de bits será 1 para los casos 1 x 0 y 0 x 1. Las multiplicaciones 1 x 1 y 0 x 0 siempre darán como resultado 0.

**Operador NOT**
El operador de negación de bits nos permite invertir el contenido de bits de una cadena. De tal manera que invierte los bits convirtiendo los 1 en 0 y los 0 en 1.

## Operadores de Desplazamiento de Bits

Los operadores de desplazamiento de bits permiten mover los bits dentro de la cadena. La estructura de estos operadores es la siguiente:

```
(valor_binario1) >> (valor_binario2)
(valor_binario1) >> (valor_binario2)
(valor_binario1) >>> (valor_binario2)
```

**Operador Desplazamiento a Izquierdas**
En el caso del desplazamiento a izquierdas, desplaza el valor_binario1 a izquierdas tantas veces como indique el valor_binario2. Los bits se mueven a la izquierda y se añaden tantos ceros como indique el valor_binario 2.

**Operador Desplazamiento a Derechas**
Es exactamente igual al operador de desplazamiento a izquierdas, pero con la diferencia que mete dígitos por la izquierda y va eliminado el último dígito.

Operador Desplazamiento a Derechas sin signo
Es como el operador desplazamiento a derechas pero no tiene en cuenta el signo. Por lo cual siempre mete ceros por la izquierda.

## Sentencias Control en Java

Un programa en Java se ejecuta en orden desde la primera sentencia hasta la última.
Dentro de las sentencias de control de flujo tenemos las siguientes:

**Sentencias de Decisión:**
Son sentencias que nos permiten tomar una decisión para poder ejecutar un bloque de sentencias u otro.

Las sentencias de decisión son: if-then-else y switch.

Mediante if-then-else podremos evaluar una decisión y elegir por un bloque u otro.

```
if (expresion) {
  // Bloque then
} else {
  // Bloque else
}
```

Mientras que con switch podremos evaluar múltiples decisiones y ejecutar un bloque asociado a cada una de ellas.

```
switch (expresion) {
  case valor1:
    bloque1;
    break;
  case valor2:
    bloque2;
    break;
  case valor3:
    bloque3;
    break;
  ...
  default:
      bloque_por_defecto;
}
```

**Sentencias de Bucle**
Las sentencias de bucle nos van a permitir ejecutar un bloque de sentencias tantas veces como queramos, o tantas veces como se cumpla una condición.

En el momento que se cumpla esta condición será cuando salgamos del bucle.

Las sentencias de bucle en Java son: while, do-while y for.

En el caso de la sentencia while tenemos un bucle que se ejecuta mientas se cumple la condición, pero puede que no se llegue a ejecutar nunca, si no se cumple la condición la primera vez.

```
while (expresion) {
  bloque_sentencias;
}
```

Por otro lado, si utilizamos do-while, lo que vamos a conseguir es que el bloque de sentencias se ejecute, al menos, una vez.

```
do {
  bloque_sentencias;
} while (expresion)
```

La sentencia for nos permite escribir toda la estructura del bucle de una forma más acotada. Si bien, su cometido es el mismo.

```
for (sentencias_inicio;expresion;incremento) {
  bloque_sentencias;
}
```

**Sentencias de ramificación**
Las sentencias de ramificación son aquellas que nos permiten romper con la ejecución lineal de un programa.

El programa se va ejecutando de forma lineal, sentencia a sentencia. Si queremos romper esta linealidad tenemos las sentencias de ramificación.

Las sentencias de ramificación en Java son: break y continue.

En el caso de break nos sirve para salir de bloque de sentencias, mientras que continue sirve para ir directamente al siguiente bloque.

## Clase String: Representando una cadena

Una cadena de texto no deja de ser más que la sucesión de un conjunto de

caracteres alfanuméricos, signos de puntuación y espacios en blanco con más o menos sentido.
Todas ellas serán representadas en java con la clase String y StringBuffer.
Para encontrar la clase String dentro de las librerías de Java tendremos que ir a java.lang.String.

Para crear una cadena tenemos dos opciones:

Instanciamos la clase String. Que sería una creación explicita de la clase

```
String sMiCadena = new String("Cadena de Texto");
```

Crear implícitamente la cadena de texto. Es decir, simplemente le asignamos el valor al objeto.

```
String sMiCadena = "Cadena de Texto";
```

En este caso, Java, creará un objeto String para tratar esta cadena.

**Crear una cadena vacía:**

Para poder crear la cadena vacía nos bastará con asignarle el valor de “”, o bien, utilizar el constructor vacío.

```
String sMiCadena = "";
String sMiCadena = new String();
```

**Constructores String**

Tenemos dos tipos de constructores principales de la clase String:

String(), que construirá un objeto String sin inicializar.
String(String original), construye una clase String con otra clase String que recibirá como argumento.

**Volcando una cadena de texto a la consola**

Esto lo haremos con la clase System.out.println que recibirá como parámetro el objeto String.
Por ejemplo:

```
System.out.println("Mi Cadena de Texto");
```

## Funciones Básicas con Cadenas

**Información básica de la cadena**
.length() Nos devuelve el tamaño que tiene la cadena.

char charAt(int index) Devuelve el carácter indicado como índice.

**Comparación de Cadenas**
Los métodos de comparación nos sirven para comparar si dos cadenas de texto son iguales o no. Dentro de los métodos de comparación tenemos los siguientes:

boolean equals(Object anObject) Nos permite comparar si dos cadenas de texto son iguales. En el caso de que sean iguales devolverá como valor “true”. En caso contrario devolverá “false”.

boolean equalsIgnoreCase(String anotherString) Compara dos cadenas de caracteres omitiendo si los caracteres están en mayúsculas o en minúsculas.

int compareTo(String anotherString) Este método es un poco más avanzado que el anterior, el cual, solo nos indicaba si las cadenas eran iguales o diferentes En este caso compara a las cadenas léxicamente. Para ello se basa en el valor Unicode de los caracteres.

**Búsqueda de caracteres**
Tenemos un conjunto de métodos que nos permiten buscar caracteres dentro de cadenas de texto. Y es que no nos debemos de olvidar que la cadena de caracteres no es más que eso: una suma de caracteres.

int indexOf(int ch) Nos devuelve la posición de un carácter dentro de la cadena de texto. En el caso de que el carácter buscado no exista nos devolverá un -1. Si lo encuentra nos devuelve un número entero con la posición que ocupa en la cadena.

int indexOf(int ch, int fromIndex) Realiza la misma operación que el anterior método, pero en vez de hacerlo a lo largo de toda la cadena lo hace desde el índice (fromIndex) que le indiquemos.

int lastIndexOf(int ch) Nos indica cual es la última posición que ocupa un carácter dentro de una cadena. Si el carácter no está en la cadena devuelve un -1. int lastIndexOf(int ch, int fromIndex) Lo mismo que el anterior, pero a partir de una posición indicada como argumento.

**Búsqueda de subcadenas**
Este conjunto de métodos son, probablemente, los más utilizados para el manejo de cadenas de caracteres. Ya que nos permiten buscar cadenas dentro de cadenas, así como saber la posición donde se encuentran en la cadena origen para poder acceder a la subcadena. Dentro de este conjunto encontramos:

int indexOf(String str) Busca una cadena dentro de la cadena origen. Devuelve un entero con el índice a partir del cual está la cadena localizada. Si no encuentra la cadena devuelve un -1.

int indexOf(String str, int fromIndex) Misma funcionalidad que indexOf(String str), pero a partir de un índice indicado como argumento del método.

int lastIndexOf(String str) Si la cadena que buscamos se repite varias veces en la cadena origen podemos utilizar este método que nos indicará el índice donde empieza la última repetición de la cadena buscada.

lastIndexOf(String str, int fromIndex) Lo mismo que el anterior, pero a partir de un índice pasado como argumento.

boolean startsWith(String prefix) Probablemente mucha gente se haya encontrado con este problema. El de saber si una cadena de texto empieza con un texto específico. La verdad es que este método podía obviarse y utilizarse el indexOf(), con el cual, en el caso de que nos devolviese un 0, sabríamos que es el inicio de la cadena.

boolean startsWith(String prefix, int toffset) Más elaborado que el anterior, y quizás, y a mi entender con un poco menos de significado que el anterior.

boolean endsWith(String suffix) Y si alguien se ha visto con la necesidad de saber si una cadena empieza por un determinado texto, no va a ser menos el que se haya preguntado si la cadena de texto acaba con otra. De igual manera que sucedía con el método .startsWith() podríamos utilizar una mezcla entre los métodos .indexOf() y .length() para reproducir el comportamiento de .endsWith(). Pero las cosas, cuanto más sencillas, doblemente mejores.

**Métodos con subcadenas**

String substring(int beginIndex) Este método nos devolverá la cadena que se encuentra entre el índice pasado como argumento (beginIndex) hasta el final de la cadena origen.

String substring(int beginIndex, int endIndex) Si se da el caso que la cadena que queramos recuperar no llega hasta el final de la cadena origen, que será lo normal, podemos utilizar este método indicando el índice inicial y final del cual queremos obtener la cadena.

**Manejo de caracteres**
Otro conjunto de métodos que nos permite jugar con los caracteres de la cadena de texto. Para ponerles en mayúsculas, minúsculas, quitarles los espacios en blanco, reemplazar caracteres,….

String toLowerCase(); Convierte todos los caracteres en minúsculas.

String toUpperCase(); Convierte todos los caracteres a mayúsculas.

String trim(); Elimina los espacios en blanco de la cadena.

String replace(char oldChar, char newChar) Este método lo utilizaremos cuando lo que queramos hacer sea el remplazar un carácter por otro. Se reemplazarán todos los caracteres encontrados.

## Arrays Java

Un array Java es una estructura de datos que nos permite almacenar una ristra de datos de un mismo tipo. El tamaño de los arrays se declara en un primer momento y no puede cambiar en tiempo de ejecución como puede producirse en otros lenguajes. La declaración de un array en Java y su inicialización se realiza de la siguiente manera:

```
tipo_dato nombre_array[];
nombre_array = new tipo_dato[tamanio];
```

**Tamaño del array: .length**
Este atributo nos devuelve el número de elementos que posee el array.

**Matrices o Arrays de varios subindices**
Podremos declarar arrays Java de varios subíndices, pudiendo tener arrays Java de dos niveles, que serían similares a las matrices, arrays Java de tres niveles, que serían como cubos y así sucesivamente, si bien a partir del tercer nivel se pierde la perspectiva geométrica. Para declarar e inicializar un array de varios subíndices lo haremos de la siguiente manera:

```
tipo_dato nombre_array[][];
nombre_array = new tipo_dato[tamanio][tamanio];
```

De esta forma podemos declarar una matriz Java de 2x2 de la siguiente forma:

```
int matriz[][];
matriz = new int[2][2];
```

**Incialización de Arrays en Java**
Existe una forma de inicializar un array en Java con el contenido, amoldándose su tamaño al número de elementos a los que le inicialicemos. Para inicializar un array Java utilizaremos las llaves de la siguiente forma:

```
tipo_dato array[] = {elemento1,elemento2,...,elementoN};
```

Así, por ejemplo, podríamos inicializar un array Java o una matriz Java:

```
// Tenemos un array de 5 elementos.
char array[] = {'a','b','c','d','e'};


// Tenemos un array de 4x4 elementos.
int array[][] = { {1,2,3,4}, {5,6,7,8}};
```

**Bibliografía**: http://www.manualweb.net/java/applets-java/