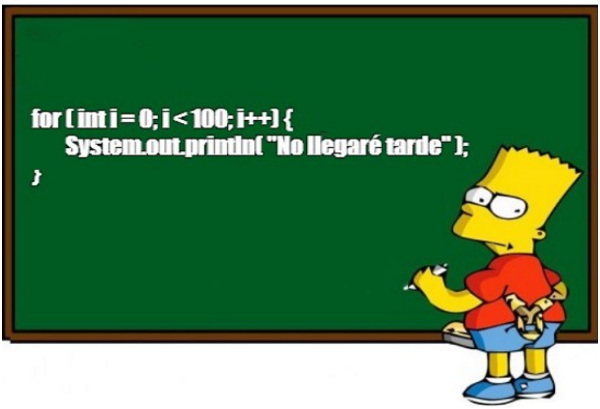
UNIDAD 3

EXCEPCIONES, BUCLES Y ARRAYS.

BIBLIOGRAFÍA:
- Java, Cómo Programar (10ªEd) Pearson. Paul y Harvey Deite(2016)
-Introduction to Programming Java (7ª Ed) David J. Eck (2014)



Nuestros programas van a ir incorporando la suficiente complejidad como para que sea conveniente desarrollarlos con ayuda de un IDE. Antes de comenzar con el contenido de la unidad, sería interesante familiarizarse con el manejo básico de por ejemplo Eclipse. Tenéis varios apéndices para diferentes IDES. En clase se explicarán algunos elementos interesantes de estas herramientas.

EJERCICIO 0. Imagina que quieres usar Java JDK SE 17 (que nace con vocación de LTS (Long-Time-Support)) y tu versión de Eclipse no le da soporte, visita markeplacet y busca "Java 17" para instalar el soporte de este JDK en el IDE Eclipse.

3.1. EXCEPCIONES.

Java tiene una forma de manejar las situaciones de error en tiempo de ejecución (excepciones) que implica cambiar el flujo de ejecución normal del programa.

Hay una clase llamada **Error** que contiene errores tan graves que nuestros programas no serían capaces de recuperarse de ellos. También existe la clase **Exception** para cuando ocurre un error durante la ejecución del programa. Ambas clases descienden de **Throwable**. El comportamiento normal es que el programa termina y se imprime un mensaje de error. Pero Java también permite atrapar (**catch**) estos errores y tener un comportamiento distinto a que el programa se rompa y termine. Se hace con la sentencia **try-catch**.

EXCEPCIONES

En Java son objetos de tipo **Exception**. Las excepciones se definen como subclases de esta clase. En este apartado vamos a usar dos tipos de excepciones: `NumberFormatException` y `IllegalArgumentException` aunque

la manera de trabajar con el resto de posibles errores es idéntica.

- **NumberFormatException:** ocurre al intentar convertir un String en un número con las funciones `Integer.parseInt()` y `Double.parseDouble()`. Por ejemplo, al llamar a Integer.parseInt(s) donde s es una variable de tipo String cuyo valor es "frodo", la función falla porque la conversión a entero es imposible.
- **IllegalArgumentException**: ocurre si se pasa un valor ilegal como parámetro a una subrutina. Por ejemplo si una subrutina necesita que el valor de un parámetro sea mayor o igual que cero y pasamos un nº negativo, la función podría generar esta excepción.

Se pueden usar las excepciones ya credas o un programador puede definirse las suyas propias. Basta con crear una clase que herede de Exception, pero esto no lo vemos por ahora.

A veces tendrás que importar las definiciones de las clases de ciertas excepciones según el código que necesitas utilizar.

3.1.1. ATRAPARLAS CON TRY-CATCH.

Si ocurre una excepción, se dice que es lanzada. Para atraparla se usa la sentencia cuya sintaxis es:

```
try {
    sentencias_que_pueden_provocar_excepciones;
} catch ( clase_de_excepción error ) {
    sentencias_a_ejecutar_si_hay_excepción;
}
```

La variable error contiene una referencia al objeto excepción que se atrapa. Este objeto excepción contiene información sobre la causa que la lanza. Incluye un mensaje de error `getMessage()` y varios métodos que

podemos utilizar. Cuando las sentencias que tratan la excepción se han ejecutado, el programa continúa en la siguiente sentencia del try-catch sin colgarse, como si nada hubiese ocurrido.

> ***Nota**: Las llaves son obligatorias aunque solamente haya una sentencia.*

EJEMPLO 1: fragmento de código que usa **try-catch**

```
double x;  // La declaramos fuera para usarla en varios bloques
try {
   x = Double.parseDouble(str);
   System.out.println( "El nº es " + x );
} catch ( NumberFormatException e ) {
   System.out.println( "No es un nº legal." );
   x = Double.NaN; // NaN es Not a Number, un valor especial.
}
```

Las setencias que se ponen en la zona de tratamiento varían. A veces es incluso mejor hacer que el programa se cuelgue. Otras veces puedes recuperar al programa del error, otras veces solamente informar del problema, otras veces la lógica de tu programa se aprovecha de que ocurran excepciones.

EJEMPLO 2: Aprovechar excepciones para definir el comportamiento de un programa. Calcular la media de nºs tecleados hasta que el valor leído sea vacío. Se ignoran valores ilegales.

```
import java.util.Scanner;

public class media {
   public static void main(String[] args) {
      Scanner teclado = new Scanner();
      String str;    // Lo teclea el usuario
      double num;    // La conversión del string
      double total;  // Suma de nºs tecleados (hasta línea blanco)
      double avg;    // la media
      int contador;  // Cantidad de nºs tecleados
```

```
            total = 0;
            contador = 0;
            System.out.println("Teclea tus nºs, return para acabar.");
            while (true) {
               System.out.print("? ");
               str = teclado.next();
               if (str.equals(""))  break; // Sale del bucle
               try {
                  num = Double.parseDouble(str);
                  /* Si hay error no se ejecutan las siguientes líneas,
                     es un comportamiento perfecto!! */
                  total = total + num;
                  contador = contador + 1;
               } catch (NumberFormatException e) {
                  System.out.println("No es legal, repite.");
               }
            } // fin while
            avg = total / contador;
            System.out.printf("La media de los %d nºs es %1.6g%n",
                              contador, avg);
        } // fin main
    }
```

En un mismo bloque try puedes atrapar diferentes excepciones y hacer tratamientos especializados para cada tipo de excepción.

```
Scanner teclado = new Scanner(System.in);
int numerador, denominador, resultado;
try {
    System.out.print("Numerador: ");
    numerador = input.nextInt();
    System.out.print ("Denominador: ");
    denominador = input.nextInt();
    resultado = numerador / denominador;
    System.out.println(numerador + " / " + denominador + " = " +
                       resultado);
} catch(ArithmeticException e) {
    System.out.println(e.getMessage());
} catch(InputMismatchException e) {
    System.out.println("Tipo de dato incorrecto");
}
```

3.1.2. LAS SENTENCIAS THROW Y THROWS.

Nosotros como programadores podemos lanzar una excepción si en cierto momento nos interesa hacerlo. Las excepciones son objetos y todas descienden de un padre común llamado **Exception**. El resto de excepciones son clases hija de esta clase (la clase madre contiene todos los elementos comunes de sus hijas) y cada hija contiene solo los elementos que la diferencian del resto.

Para lanzar una excepción debemos usar la sentencia **throw**. Antes o en ese mismo momento debemos construir un nuevo objeto **Exception que lanzar**. Las excepciones tienen cuatro constructores sobrecargados (cuatro versiones).

- Una vacía: `new Exception();`
- Otra con un mensaje: `new Exception( "Mi mensaje" )`
- Otra con el mensaje y la causa: `new Exception(String message, Throwable cause)`
- Otra con la causa: `new Exception(Throwable cause)`

EJEMPLO 3: lanzar una excepción si es menor de edad. La excepción la creamos antes.

```
int edad = escaner.nextInt();
if( edad < 18 ) {
    Exception e = new Exception("Debe ser mayor de edad");
    throw e;
}
```

EJEMPLO 4: lanzar una excepción si es menor de edad pero creando la excepción directamente.

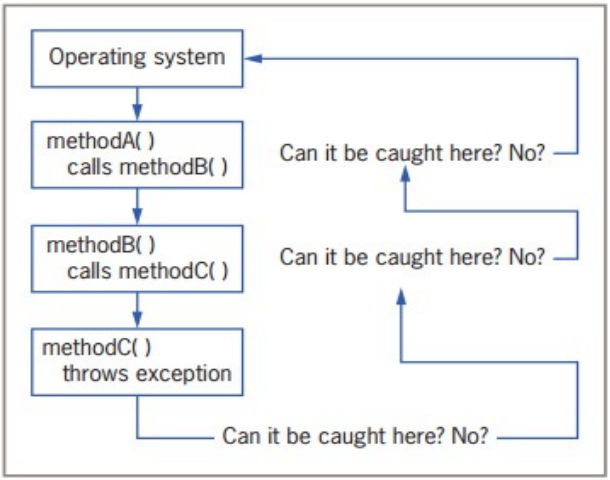
```
int edad = escaner.nextInt();
if( edad < 18 )
    throw new Exception("Debe ser mayor de edad");
```

Cuando un bloque de código utiliza sentencias que pueden generar

excepciones, es necesario:

- Protegerlo atrapándolas.
- Al menos avisar al código que lo utilice, de que podrían aparecer esos errores en tiempo de ejecución (de esta forma vamos a pasar la obligación de tratar las excepciones a otro lugar).

Para pasar esa responsabilidad del tratamiento a otro lugar se usa la sentencia **throws** seguida de una lista de excepciones que pueden aparecer en el código.
Podemos ver por cuántos métodos se ha propagado el error usando `printStackTrace()`.



Suelen ponerse en la declaración de los métodos (antes de abrir su bloque de código. Sintaxis:

```
// El bloque puede lanzar cualquiera de las excepciones indicadas
... throws excepcion1 [, excepción2 [, ...] ]  { .... }
```

EJEMPLO 5: lanzar una excepción y delegar su tratamiento.

```
public static void main(String[] args) throws Exception {
    int edad = escaner.nextInt();
    if( edad < 18 )
        throw new Exception("Debe ser mayor de edad");
}
```

3.1.3. LA SENTENCIA VACÍA Y EL BLOQUE VACÍO.

La sentencia vacía consiste en un punto y coma pero que no lleva sentencia. Es el equivalente a no hacer nada.

EJEMPLO 6: Ejemplo de sentencia vacía.

```
if (x < 0) {
  x = -x;
};
```

El punto y coma detrás de la llave que cierra el if no es necesario, por tanto es una sentencia vacía (no hace nada). Otro ejemplo:

```
if ( realizado )
   ; // sentencia vacía
else
   System.out.println( "No se ha realizado aún.");
```

A veces pueden provocar errores de interpretación. Por ejemplo la siguientes sentencias imprimen "Hola" una sola vez, porque el bucle **for** se repite para una sentencia vacía (por el ; se acaba el for):

```
for (int i = 0; i < 10; i++);
   System.out.println("Hola");
```

De la misma manera, un bloque vacío es aquel que no tiene sentencias en su interior.

EJEMPLO 7: Silenciar una excepción con un bloque vacío.

```
int x = escaner.nextInt();
try { int inverso = 1 / x; } catch(Exception e) {}
/* en caso de que x sea 0 el programa continúa como si nada
   aunque no toma ninguna medida (ni avisa, ni soluciona) solo
   silencia el error */
```

3.1.3. LIBERAR RECURSOS CON FINALLY.

El uso de la cláusula **finally** en la sentencia try-catch permite asegurar

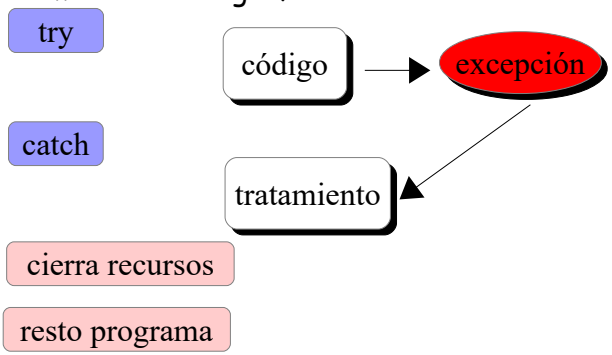
al programador que se cierran recursos (se libera memoria que están usando los objetos de nuestro código incluso en caso de error).

Vamos a ver un ejemplo sencillo con la división de dos números enteros donde el denominador es 0. Recordemos que es una operación que no se puede realizar y la máquina virtual de Java lanzará una excepción que podemos capturar con un bloque try-catch (en realidad en este caso hubiera sido mejor comprobar simplemente el denominador, pero nos puede servir de ejemplo).

```
public static void main(String[] args) {
    int a = 5;
    int b = 0;
    try {
       int resultado=a/b;
       System.out.println(resultado);
    } catch (Exception e) {
       System.out.println("La aplicacion falla.");
    }
    System.out.println("Se cierran los recursos.");
    System.out.println("La aplicacion finaliza.");
 } // main
```

JAVA SIN FINALLY

El resultado lo podemos ver en el gráfico:



La figura representa cuando la excepción salta y es capturada, se emite un mensaje de error y fuera del try-catch se cierran los recursos y la aplicación finaliza:

Bueno, pues lo que hay fuera de la sentencia try que cierra recursos, podemos indicarlo en una cláusula **finally** de la sentencia try-catch.

```
package p1;

public class Principal2 {

    public static void main(String[] args) {
        int a =5;
        int b=0;
        try {
            int resultado=a/b;
            System.out.println(resultado);
        } catch (Exception e) {
            System.out.println("la aplicacion falla");
        } finally {
            System.out.println("se cierran los recursos");
        }
        System.out.println("la aplicacion finaliza");
    } // fin main
} // fin clase
```

Todo parece quedar un poco más ordenado, sin embargo ¿El resultado es el mismo?

ES MEJOR USAR FINALLY

Podemos preguntarnos para qué es necesario usar **finally** si podemos conseguir lo mismo sin utilizarlo. La respuesta es que finally asegura siempre el cierre de los recursos y se ejecuta ocurra lo que ocurra en el programa (funcione o falle). Por ejemplo imagina que se produce una excepción dentro de la clausula catch.

```
package p1;
```

```
public class Principal1 {

    public static void main(String[] args) {
        int a = 5;
        int b = 0;
        try {
            int resultado=a/b;
            System.out.println(resultado);
        } catch (Exception e) {
            System.out.println("La aplicacion falla.");
            throw new NullPointerException();  // Lo forzamos
        }
        System.out.println("Se cierran los recursos.");
        System.out.println("La aplicación finaliza.");
    }
}
```

En este caso el programa finaliza. pero no llega a cerrar los recursos **y eso supone un problema**. Muchas veces es una situación difícil de prevenir, son cosas que ocurren. Si se lanza una excepción dentro de la cláusula catch los **recursos no se cerrarán**. Ahora bien, si hemos usado finally la situación cambia: haya o no errores en cualquier parte de la sentencia try-catch, lo último que se ejecuta es lo que haya en finally.

3.1.5. LA SENTENCIA TRY-CATCH CON RECURSOS.

Una de las novedades que incorporó Java 7 es esta variación de la sentencia **try-catch** con el objetivo de cerrar los recursos de forma automática en la sentencia **try-catch-finally** y hacer más simple el código.

Aquellas variables cuyas clases implementan la interfaz **AutoCloseable** pueden declararse en el bloque de inicialización de la sentencia **try-con recursos** y sus métodos **close()** se llaman después del bloque **finally** como si su código estuviese de forma explícita dentro.

Un ejemplo de código que usa un Objeto para leer una línea de un fichero de texto y luego lo libera porque ya no lo va a usar más. Como se observa no es necesario llamar de forma explícita al método close() para liberar los recursos del objeto **BufferedReader**.

```
public static String leeLinea(String ruta) throws IOException {
    try (BufferedReader br = new BufferedReader(
                                    new FileReader(ruta)
                                )
        ) {
        return br.readLine();
    }
}
```

Anteriormente a Java 7 esto se debía hacer de la siguiente manera con unas pocas lineas más de código y algo menos legible.

```
public static String leeLinea(String ruta) throws IOException {
    BufferedReader br = new BufferedReader(
                                            new FileReader(ruta)
                                           );
    try {
        return br.readLine();
    } finally {
        if (br != null) br.close();
    }
}
```

El código es similar pero no es equivalente. Observa que requiere declarar la variable br fuera de la sentencia try-catch-finally donde se usa. Además, si se produce una excepción en el bloque try y posteriormente en el bloque finally en Java 6 la excepción del bloque try se enmascara y la que se lanza es la del bloque finally, lo que provoca que sea más difícil de solucionar al poder confundirla.

La excepción que se lanza en el bloque try puede averiguarse usando el método `Throwable.addSuppressed()` que se añadió a la API en Java 7 junto con el método `Throwable.getSuppressed()` se obtienen las excepciones

enmascaradas o suprimidas en la sentencia try-con recursos. El orden de ejecución de los bloques de una sentencia try-con recursos es el indicado en los números emitidos con el método println en el ejemplo.

```
import java.io.BufferedReader;
import java.io.FileReader;

public class Main {

    public static void main(String[] args) {
        try {
            leeLinea();
        } catch (Exception e) {
            e.printStackTrace();
            for (Throwable t : e.getSuppressed()) {
                t.printStackTrace();
            }
        }
    }

    private static String leeLinea() throws Exception {
        System.out.println("0");
        String linea = null;
        Exception excepcion = null;
        try (BufferedReader br = new BufferedReader(
                                        new FileReader("file.txt")
                                    )
            ) {
            System.out.println("1");
            lanzaExcepcion(null);
            linea = br.readLine();
        } catch (Exception e) {
            System.out.println("2");
            excepcion = e;
        } finally {
            System.out.println("3");
            tlanzaExcepcion(excepcion);
        }
        System.out.println("4");
        return linea;
    }
```

```
        private static void lanzaException(Exception x)
        throws Exception {
            Exception e = new Exception();
            if (x != null) {
                e.addSuppressed(x);
            }
            throw e;
        }
    }
```

La mayoría de clases relacionadas con entrada y salida de datos y conexión a bases de datos mediante JDBC con Connection implementan la interfaz ***AutoCloseable***. También las relacionadas con el sistema de ficheros y flujos de red como InputStream.

3.2. SENTENCIAS DE REPETICIÓN.

Las sentencias que nos permiten repetir varias veces la ejecución de una o más sentencias se denominan bucles o lazos. Son un elemento fundamental para poder resolver problemas más complejos que los visto hasta ahora y dominarlas es fundamental para programar. Veamos las que Java nos ofrece.

3.2.1. BUCLE CONDICIONAL MIENTRAS (while).

La instrucción **while** permite crear bucles controlados por una condición. Se repite la ejecución de una sentencia o un grupo de sentencias, mientras se cumpla una determinada condición. Antes de ejecutarse las sentencias, la condición se evalúa. Si es true, se ejecutan. Cuando se han ejecutado, se vuelve a evaluar la condición. Y así hasta que se evalúa a false, momento en que el flujo pasa a la siguiente sentencia del bucle. Por tanto cabe la posibilidad de que las sentencias a repetir nunca se ejecuten (el bucle nunca entra).

Sintaxis:

```
while ( expresión_lógica ) {
   repite_estas_sentencias;  // Si es una, puede ir sin llaves
}
```

En forma de pseudocódigo, por pasos numerados:

(1) Se evalúa la expresión lógica

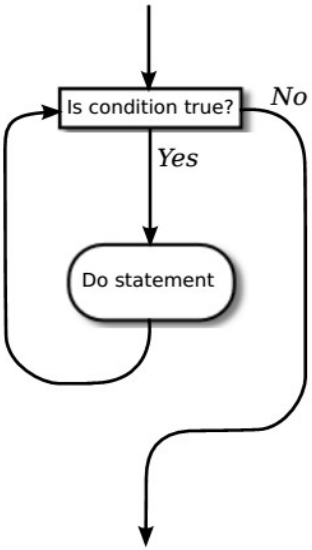
(2) Si es verdadera ejecuta la(s) sentencia(s), sino, continúa por la siguiente instrucción del while.

(3) Volver al paso 1

EJEMPLO 7: escribir números enteros del 1 al 100 ambos incluidos:

```
int i = 1;
while ( i <= 100 ) {
   System.out.println( i );
   i++;  // Si no cambias i, la condición siempre será cierta!!
}
```

Expresado en forma de diagrama de flujo:



BUCLES WHILE CONTROLADOS POR CONTADOR

Se llaman así a los bucles que se repiten un determinado número de veces y una variable (llamada **variable contador** o simplemente **contador**) almacena ese número de veces que se debe repetir. Sirve para contar las repeticiones que faltan o las que se han realizado.

Cada vez que se ejecuten las sentencias del bucle (una **iteración**) la variable debe cambiar de valor para reflejar correctamente las ejecuciones realizadas. La expresión lógica del bucle incluye esa variable. En todos los bucles de contador necesitamos saber:

(1) Lo que vale la variable contador al principio, antes de entrar en el bucle (**valor inicial** o límite inicial).
(2) Lo que varía (lo que se incrementa o decrementa) el contador en cada iteración.
(3) Las acciones a realizar en cada iteración del bucle.
(4) El **valor final** del contador o límite final. En cuanto se alcance o rebase, el bucle termina.

Si usamos el contador no para contar las ejecuciones que se han realizado sino las que quedan por realizar, el valor se pone como condición del bucle, pero a la inversa, es decir, si contamos las ejecuciones realizadas la condición mide el valor que tiene que tener el contador para que el bucle se repita y si contamos las que faltan por hacer la condición usa el valor para que termine. Ejemplo:

```
int i= 10; /* Valor inicial del contador, empieza valiendo 10 */
while ( i <= 200 ){ /* condición del bucle */
  System.out.printf("%d\n", i); /* acciones */
  i += 10; /* Variación del contador */
}
```

¿Serías capaz de decir cuál es el primer valor del contador cuando se

ejecutan las sentencias? ¿Y el valor la última vez que se ejecutan? ¿Podrías predecir el resultado de ejecutarlo?

BUCLES WHILE CONTROLADOS POR CENTINELA

Una condición lógica (llamada **centinela**), que puede ser desde una simple variable booleana hasta una expresión lógica más compleja, sirve para decidir si el bucle se repite o no. De modo que cuando la condición lógica se incumpla, el bucle termina. Esa expresión lógica a cumplir es lo que se conoce como centinela y normalmente la suele realizar una variable booleana. Ejemplo:

```
/* En este caso el centinela es una variable booleana que
inicialmente vale falso */
boolean salir = false;
int n;
/* Condición de repetición: que salir siga siendo falso. Ese es el
   centinela. También se podía haber escrito: while(!salir)
*/
while ( salir == false) {
   n= (int)Math.floor(Math.random() * 500 + 1; // Lo que se repite
   System.out.println(i);
   salir= (i % 7 == 0); /* centinela true si n múltiplo de 7 */
}
```

Comparando los bucles de centinela con los de contador, podemos señalar estos puntos:

- Solo puedes usar bucles por contador si cuando aparece el bucle, el programa puede conocer el número concreto de veces que hay que repetir las sentencias. En los bucles por centinela no es obligatorio saber de antemano ese número de veces. Por tanto es ideal usarlos en esas situaciones.
- En general, en un bucle por contador es más sencillo saber cuando finaliza, el de centinela es más propenso a errores.
- Un bucle por contador está relacionado con la programación de

algoritmos basados en series (Hacer algo para i=1,2,3...n).

BUCLES MIXTOS: POR CENTINELA Y POR CONTADOR

Un bucle podría ser incluso mixto: controlado a la vez por centinela y por contador. Por ejemplo, imagina un programa que escriba números aleatorios entre 1 y 500 y se repita hasta que encuentre el primer múltiplo de 7, pero que como mucho, imprima 5 números (se repita cinco veces como máximo). Sería:

```
boolean salir = false; // centinela
int n;
int i = 1; // contador
while (salir == false && i < = 5) {
   n = (int) Math.floor( Math.random() * 500 + 1);
   System.out.println(n);
   i++;   // actualizar el contador
   salir = (i % 7 == 0);  // actualizar el centinela
}
```

***Nota**: en un bucle condicional while, podría ocurrir que las sentencias nunca se ejecuten, si la condición no se cumple la primera vez.*

Nota: *debes asegurarte de no escribir bucles infinitos (salvo que sea lo que necesitas), porque la ejecución del programa quedará atrapada en el bucle. Las consecuencias dependerán de lo que hagan las sentencias: el programa deja de responder y sube el consumo de CPU de la máquina, si las sentencias consumen memoria, sube el consumo de memoria (hasta agotarla) y puede hacer caer hasta un servidor o al propio sistema operativo.*

3.2.2. BUCLE CONDICIONAL HAZ MIENTRAS (do-while).

La única diferencia respecto a la sentencia anterior está en que la expresión lógica se evalúa después de haber ejecutado las sentencias. Es decir, este bucle te asegura que se ejecutan al menos una vez las sentencias del cuerpo. Sintaxis:

```
do {
   instrucciones_a_repetir;
} while ( expresión_lógica );
```

Expresada la lógica en pseudocódigo, por pasos:

(1) Ejecutar sentencias
(2) Evaluar expresión lógica
(3) Si la expresión es verdadera volver al paso 1, sino continuar fuera del while.

EJEMPLO 8: contar de 1 a 1000 usando un bucle do-while.

```
int i = 0;
do {
   i++;
   System.out.println(i);
} while ( i < 1000 );  // Observa: i debe definirse fuera
```

EJERCICIO 1: Repite el ejemplo 8 usando un bucle while.

Se utiliza cuando sabemos que las sentencias del bucle se van a ejecutar como mínimo una vez (en un bucle while puede que incluso no se ejecuten las sentencias que hay dentro del bucle si la condición fuera falsa desde un inicio). De hecho cualquier sentencia **do-while** se puede convertir en **while** y viceversa. El ejemplo anterior usando la instrucción while:

```
int i = 0;
i++;  // Sacando una copia de las sentencias antes del bucle
System.out.println(i);
```

```
while ( i < 1000 ) {
   i++;
   System.out.println(i);
}
```

EJERCICIO 2. En este ejemplo, ¿El bucle es controlado por centinela o contador?

3.2.3. BUCLE CONTADOR (for).

Es un bucle más complejo, especialmente pensado para situaciones donde se necesita ejecutar instrucciones repetidamente controladas por contador. Una vez más se ejecutan una serie de instrucciones en el caso de que se cumpla una determinada condición. Pero la condición suele usar un contador. Sintaxis:

```
for(inicialización; condición; incremento){
   sentencias_a_repetir;
}
```

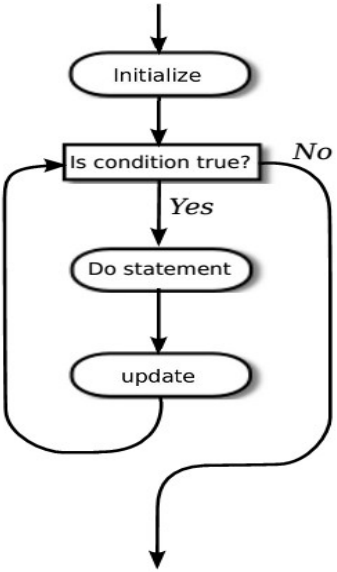
Las sentencias se ejecutan mientras la condición sea verdadera. Además antes de entrar en el bucle se ejecuta la instrucción de inicialización y en cada iteración se ejecuta el incremento. Es decir el funcionamiento expresado en psudocódigo por pasos es:

(1) Se ejecuta la instrucción de inicialización
(2) Se comprueba la condición
(3) Si es cierta, entonces se ejecutan las sentencias. Si la condición es falsa, abandonamos el bloque for
(4) Tras ejecutar las sentencias, se ejecuta la instrucción de incremento y se vuelve al paso 2.

EJEMPLO 8: contar números del 1 al 1000 con un bucle for.

```
for(int i= 1; i <= 1000; i++){
   System.out.println( i );
}
```

La ventaja que tiene con respecto a usar una sentencia while es que el código se reduce. La desventaja es que el código es menos comprensible. En diagrama de flujo:



El bucle anterior es equivalente al siguiente bucle while:

```
int i= 1 ;           // sentencia de inicialización
while( i <= 1000) { // condición
   System.out.println(i);
   i++;              // cambio del contador, en for aparece arriba
}
```

Es posible declarar la variable contador usada en el bucle tanto fuera como dentro del propio bucle for. Hacerlo dentro es la forma habitual de declarar un contador. De esta manera se crea una variable que muere en cuanto el bucle acaba.

Debes prestar atención siempre al primer y último valor que toma la variable. Por ejemplo, intenta averiguar el primer y último valor usados en los bucles:

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

```
      System.out.println( N );
// comienza en 1 y acaba en 10
for ( N = 0 ; N < 10 ; N++ )
   System.out.println( N );
// comienza en cero y acaba en 9
for ( N = 10 ; N >= 1 ; N-- )
   System.out.println( N );
// comienza en 10 y acaba en 1
```

Es posible usar en la parte de inicialización y actualización, varias expresiones separadas por comas. Ejemplo:

```
for ( i=1, j=10; i <= 10; i++, j-- ) {
   System.out.printf("%5d", i); // muestra i en 5 columnas
   System.out.printf("%5d", j); // muestra j en 5 columnas
   System.out.println();        // salto de línea
}
```

EJEMPLO 9: Hacer un bucle for que muestre los números pares entre 2 y 20. Mostramos varias soluciones posibles:

```
// (1) Hay 10 nºs a mostrar. Generamos 1,2...10
// para imprimir 2*1, 2*2, ... 2*10.
for (N = 1; N <= 10; N++) {
   System.out.println( 2 * N );
}
// (2) Generar directamente la salida: 2, 4, ...20
for (N = 2; N <= 20; N = N + 2) {
   System.out.println( N );
}
// (3) Generar todos (1,2,...,19,20) y solo imprimir
// los que sean pares
for (N = 2; N <= 20; N++) {
   if ( N % 2 == 0 ) // si N es par
      System.out.println( N );
}
```

EJEMPLO 10: Imprimir las letras mayúsculas ente la 'A' y la 'Z'.

```
char ch; // No solo pueden utilizarse enteros, un char es un nº
for ( ch = 'A'; ch <= 'Z'; ch++ )
   System.out.print(ch);
```

```
System.out.println();
```

EJEMPLO 11: contar cuantos divisores tiene un nº llamado n (nºs entre 1 y n que lo dividen dejando un resto cero).

```
int numeroDivisores = 0;
for (testDivisor = 1; testDivisor <= N; testDivisor++) {
   if ( N % testDivisor == 0 )
     numeroDivisores++;
}
```

3.2.4. SENTENCIAS DE RUPTURA DE FLUJO

Solamente afectan a bucles (while, do-while, for) salvo break que también se usan en la sentencia switch. No es aconsejable su uso intensivo ya que son instrucciones que rompen el paradigma de la programación estructurada. Se comentan porque usados de forma adecuada y no generalizada pueden ser útiles: ahorran trabajo o mejoran la eficiencia en ciertas situaciones.

SENTENCIA BREAK

Se trata de una sentencia que hace que el flujo de ejecución del programa abandone el bloque del bucle en el que se encuentra y continúe fuera. Es como romper el bucle, hacer que se acabe. Ejemplo:

```
for(int i=1; i <= 1000; i++){
   System.out.println(i);
   if( i == 300 ) break;
}
```

En el listado anterior, el contador no llega a 1000, en cuanto llega a 300 sale del for. Recuerda que aunque aparezca dentro de un if, afecta al bucle dentro del que se encuentre.

SENTENCIA CONTINUE

Es parecida a la anterior, sólo que en este caso en lugar de abandonar

el bucle, lo que ocurre es que no se ejecutan el resto de sentencias del bucle en la iteración actual y se vuelve directamente a la condición del mismo: abandona la repetición o iteración actual y continúa con la siguiente iteración:

```
for(int i=1; i <= 1000; i++){
   if( i % 3 == 0 ) continue;
   System.out.println(i);
}
```

En ese listado aparecen los números del 1 al 1000, menos los múltiplos de 3 (en los múltiplos de 3 se ejecuta la instrucción **continue** que salta el resto de instrucciones del bucle y vuelve a la siguiente iteración. El uso de esta sentencia genera malos hábitos, siempre es mejor resolver los problemas sin usar **continue**. El ejemplo anterior sin usar esta instrucción quedaría:

```
for(int i=1; i <= 1000; i++){
   if( i % 3 != 0) System.out.println(i);
}
```

BREAK ETIQUETADO

Se usa muy muy poco. Consiste en poner a un bucle un nombre (una etiqueta) y al usar el break indicas a que nombre debe saltar. Sintaxis:

```
//Poner etiqueta                    // Saltar a etiqueta:
etiqueta: bucle                     break etiqueta;
```

EJEMPLO 12: Averiguar si dos String tienen caracteres en común.

```
boolean disjuntos = true; // No tienen en común
int i, j; // Recorrer letras de los dos strings por posición
i = 0;
buclePincipal: while ( i < s1.length() ) {
   j = 0;
   while ( j < s2.length() ) {
      if ( s1.charAt(i) == s2.charAt(j) ) { // s1 y s2 en común
         disjuntas = false;
```

```
                break buclePrincipal; // rompe los dos bucles
            }
            j++; // Siguiente letra de s2
        }
        i++; // Siguiente letra de s1.
    }
```

Podrías evitar usarla si añades un centinela en las expresiones lógicas de los bucles del tipo: .. && disjuntas

3.2.5. BUCLES ANIDADOS.

Las sentencias de control pueden contener otras sentencias de control, así que es normal que un bucle contenga otro bucle en su interior. Puedes tener varios niveles de anidamiento, Java no fija ningún límite.

EJEMPLO 13: Contruir la siguiente tabla de multiplicación:

```
 1  2  3  4  5  6  7  8   9  10  11  12
 2  4  6  8 10 12 14 16  18  20  22  24
 :  :  :
 9 18 27 36 45 54 63 72  81  90  99 108
10 20 30 40 50 60 70 80  90 100 110 120
```

Los datos de la tabla están formados por 10 filas y 12 columnas.

```
for (int fila = 1; fila <= 10; fila++ ) {
   for (int N = 1; N <= 12; N++ ) {
      // imprime las columnas en 4 caracteres
      System.out.printf( "%4d", N * fila );
   }
   System.out.println(); // Al acabar una fila salta línea
}
```

EJEMPLO 14: Contar las letras de un String que son diferentes siguiendo este algoritmo.

Paso 1: Leer cadena
Paso 2: contador <- 0

Paso 3: PARA cada letra del alfabeto:
Paso 4: SI la letra del alfabeto está en cadena ENTONCES
Paso 5: Imprimir la letra
Paso 6: contador <- contador + 1
Paso 7: Escribir contador

```
/**
* Lee cadena e imprime las letras no repetidas que tiene
* y cuantas letras diferentes tiene
*/
import java.util.Scanner;

public class ListarLetras {

public static void main(String[] args) {
   Scanner teclado = new Scanner( System.in );
   String cadena;     // Linea de texto del usuario
   int contador = 0;  // Nº de letras distintas
   char letra;        // Una letra
   System.out.println("Teclee un texto.");
   cadena = teclado.nextln();
   cadena = cadena.toUpperCase();
   System.out.println("Las letras distitnas son:");
   System.out.println();
     System.out.print(" ");
     for ( letra = 'A'; lettra <= 'Z'; letra++ ) {
        int i; // Position of a character in str.
        for ( i = 0; i < cadena.length(); i++ ) {
           if ( letra == cadena.charAt(i) ) {
              System.out.print(letra);
              System.out.print(' ');
              contador++;
              break;
           }
        }
     }
     System.out.println();
     System.out.println();
     System.out.println("Hay " + contador + " letras distintas.");
   } //  main()
} // fin class
```

EJEMPLO 15: Usar bucles y excepciones para blindar la lectura de datos frente a entradas incorrectas del usuario. Vamos a pedir un entero al usuario y vamos a evitar que el programa se rompa si el usuario teclea otro valor distinto.

```
Scanner sc = new Scanner(System.in);
int edad = 0;
boolean errorDeLectura;
do {
    errorDeLectura = false;
    try {
        System.out.print("Teclee su edad (un entero): ");
        edad = sc.nextInt();
    }catch(InputMismatchException ime) {
        errorDeLectura = true;
        sc.nextLine();    // Quitamos el salto de línea de sc
    }
} while( errorDeLectura ); // Es como poner errorDeLectura == true
```

Se usa un centinela que se activa (a true) cuando se lanza la excepción si el usuario teclea cualquier cosa que no sea un entero. Además se elimina el salto de línea que queda en sc antes de hacer la siguiente lectura. El `do-while` usa el centinela para repetir la lectura si hay error.

EJEMPLO 16: Repetimos el caso anterior pero sin usar excepciones.

```
Scanner sc = new Scanner(System.in);
int edad = 0;
boolean errorDeLectura;
do {
    errorDeLectura = false;
    System.out.print("Teclee su edad (un entero): ");
    if( sc.hasNextInt() )
        edad = sc.nextInt();
    else {
       errorDeLectura = true;
       sc.nextLine();    // Quitamos el salto de línea de sc
    }
} while( errorDeLectura ); // Es como poner errorDeLectura == true
```

3.3. ARRAYS.

3.3.1 LOS OBJETOS ARRAY.

Un array es un tipo de dato (un objeto en Java) que puede almacenar más de un valor, es decir, puede almacenar una lista de valores. A estos tipos de datos se les llama **colecciones**, porque guardan una colección de valores.

Cada valor del array ocupa una posición llamada índice (es como una dirección) dentro del array. **Las posiciones se indican mediante un número entero**. **La primera posición de un array es la cero**. Se usa el operador de array para indicar esta posición (**operador array: []**). **En Java, todos los elementos que guarda un array deben ser del mismo tipo**. Los elementos de un array pueden ser tipos primitivos o referencias a objetos (incluyendo otros arrays).

Para trabajar con arrays hay que familiarizarse con varios términos que se usan habitualmente:

- **Longitud del array**: máximo número de elementos que puede guardar.
- **Tipo base:** el tipo de dato de los elementos que almacena.
- **Índice de un elemento**: la posición donde se almacena el elemento dentro del array (de 0 hasta longitud del array - 1).

¿Qué utilidad tienen? ¿Qué nos aportan? Imagina que tienes que hacer un programa que sea capaz de trabajar con los nombres de 1000 personas. Si tienes que leer los datos, necesitas 1000 sentencias de lectura, si necesitas impimir los nombres, necesitas 1000 sentencias de escritura, y claro, 1000 variables donde almacenarlos. Lógicamente programar de esta forma es muy ineficaz, por ejemplo, hay declarar 1000 variables al estilo:

```
String nombre1, nombre2, ...., nombre1000;
```

Los arrays nos proporcionan la herramienta perfecta para solucionar de forma eficiente esta necesidad. Podemos tener en nuestro programa un array de mil elementos que ocupen posiciones de la 0 a la 999. Cada posición del array actúa y es en realidad una variable. El tipo base del array (el tipo de los elementos) es String.

Así que tener una variable array de 1000 cadenas es como tener 1000 variables de tipo cadena, cada una en una posición del array, en la que podemos almacenar un valor, cambiarlo, leerlo, etc.

En un programa te puede interesar trabajar con todo el array en su conjunto, y a veces puede interesarte trabajar solamente con uno de sus elementos. Si quieres trabajar con todo el array, usas el nombre de su variable, si solamente quieres trabajar con uno de sus elementos, debes indicar la posición que ocupa. Sintaxis:

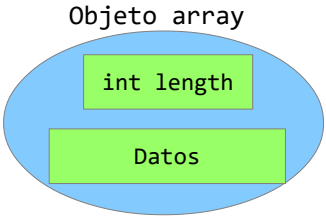
```
/* Elemento del array variableArray en la posición expresión */
variableArray[ expresión_de_tipo_int ]
```

Por ejemplo, si tenemos nuestro array de nombres de personas que se llama `listaPersonas` y queremos utilizar el nombre almacenado en la posición 7, se escribe `listaPersonas[7]` para usarlo en una expresión:

```
"Hola Sr. " + listaPersonas[7] +  ", espere un momento...";
```

Cada array además de los datos también tiene una variable con su longitud. Para usar la longitud de un array se utiliza el operador contiene (.):

```
variableArray.length
```



En la figura 2 se muestra una representación lógica de un array de enteros, llamado c que contiene 12 elementos. Un programa puede hacer referencia a cualquiera de estos elementos mediante una **expresión de acceso a un array** que contiene el nombre del array, seguido por el índice del elemento encerrado entre corchetes (**[ ]**). **El primer elemento en cualquier array tiene el índice cero**, y algunas veces se le denomina elemento cero. Por lo tanto, los elementos del array c son c[0], c[1], c[2] y así sucesivamente. El mayor índice en el array c es 11 (1 menos que 12, donde 12 es la longitud del array), es decir, el número anterior al número de elementos en el array.

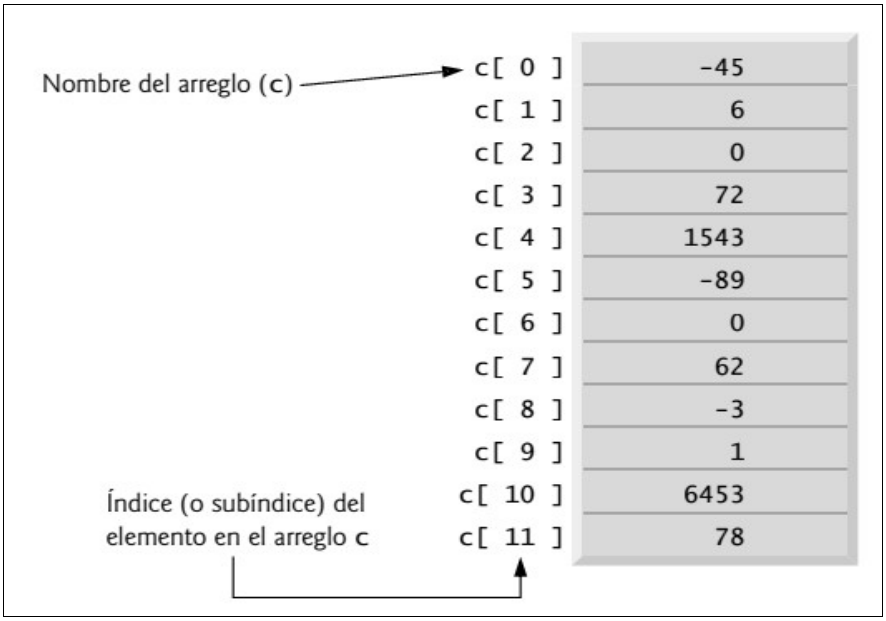


Figura 2: Representación lógica de un array de enteros.

Los nombres de los arrays siguen las mismas convenciones que los demás nombres de variables.

Un índice debe ser un entero no negativo. Un programa puede utilizar una expresión como índice. Por ejemplo, si suponemos que la variable **a** es 5 y que **b** es 6, entonces la instrucción **c[a + b] += 2;** suma 2 al elemento c[11] del array.

El nombre de un array con subíndice es una expresión de acceso al array, que puede utilizarse:

- En el lado izquierdo de una asignación, para cambiar el valor en un elemento del array.
- En el lado derecho para usar el valor/objeto almacenado en una posición del array.

> ***Nota**: **Un índice debe ser un valor int**, o un valor de un tipo que pueda promoverse a int; por ejemplo, byte, short o char, **pero no long**. De lo contrario, ocurre un error de compilación.*

En la figura 2, el nombre del array es c. Cada array conoce su propia longitud y mantiene esta información en su su variable de instancia **length**.

La expresión **c.length** devuelve la longitud del array c. Aun cuando la variable de instancia **length** de un array es public, no puede cambiarse, ya que es una variable **final**. La manera en que se hace referencia a los 12 elementos de este array: c[0], c[1], c[2], …,c[11] . El valor de c[0] es -45, el valor de c[1] es 6, el de c[2] es 0, el de c[7] es 62 y el de c[11] es 78. Para calcular la suma de los valores contenidos en los primeros tres elementos del array c y almacenar el resultado en la variable suma, escribiríamos lo siguiente:

```
suma = c[0] + c[1] + c[2];
```

Para dividir el valor de c[6] entre 2 y asignar el resultado a la variable x, escribiríamos lo siguiente:

```
x = c[6] / 2;
```

DECLARAR UN ARRAY

Antes de que puedas usar una variable en el programa, debes declararla, esto se mantiene con los arrays. La declaración consiste en indicar el tipo base, el operador array (los corchetes) y el nombre de la variable. Sintaxis:

```
tipoBase[] variable; // Declara variable como array de tipoBase
```

EJEMPLO 17: declaramos tres arrays, uno de cadenas, otro de enteros y otro de valores double.

```
String[] listaNombres;      // Lista de cadenas
int[] a;                    // Lista de enteros
double[] precios;           // Lista de números double
```

En Java, los arrays son objetos. Por tanto, declarar una variable no hace que exista el objeto. Para que exista hay que instanciarlo (crearlo).

CREAR/INSTANCIAR UN ARRAY

Si solamente declaras una variable array, aún no puedes usarla. Antes de usarla debe tener un valor. Para darle valor hay que crear un objeto array con el operador **new**. La sintaxis para asignar un valor array a una variable array sería:

```
variableArray = new tipoBase[ expresión_de_tipo_int ];
```

Donde `expresión` indica cuantos elementos puede guardar el array (el tamaño o la longitud del array). Ejemplos:

```
listaNombres = new String[1000];
A = new int[5];
precios = new double[100];
```

Observa que en la declaración no se indica la cantidad de elementos que almacena, pero si en la creación. Si visualizas mentalmente el array llamado A, al que se da valor en la segunda asignación, será algo similar a la figura siguiente.

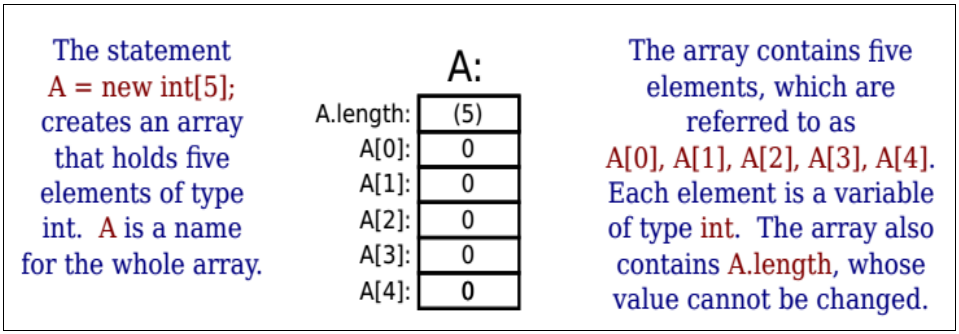


Figura 1: Un array A declarado y asignado un array de 5 enteros.

***Nota**: si creas un array de int, … float o double, los elementos se ponen a 0. Si lo creas de boolean a false. Si lo creas de char, al carácter UNICODE 0 y si lo creas de String o de cualquier otra clase, a **null**.*

Al crear un array, cada uno de sus elementos recibe un valor predeterminado (0 para los elementos numéricos de tipos primitivos, **false** para los elementos boolean y **null** para las referencias). Podemos proporcionar también valores iniciales no predeterminados al crear un array para sus elementos (más adelante).

La creación del array también puede realizarse en dos pasos, como se muestra a continuación:

```
int[] c;           // declara la variable array
c = new int[12] ;  // crea el array
```

En la declaración, los corchetes que van después del tipo indican que c

es una variable que hará referencia a un array (es decir, la variable almacenará una referencia a un array). En la instrucción de asignación, la variable c recibe la referencia a un nuevo objeto array de 12 elementos **int**.

> **Nota**: *En la declaración de un array, si se especifica el número de elementos en los corchetes de la declaración (por ejemplo, int[12] c;), se produce un error de sintaxis.*

Un programa puede crear varios arrays en una sola declaración. La siguiente declaración reserva 100 elementos para b y 27 para x:

```
String[] b = new String[100], x = new String[27];
```

Cuando el tipo del array y los corchetes se combinan al principio de la declaración, todos los identificadores en ésta son variables tipo array. En este caso, las variables b y x hacen referencia a arrays String. Por cuestión de legibilidad, es preferible declarar sólo una variable en cada declaración. La declaración anterior es equivalente a:

```
String[] b = new String[100] ; // declara y crea el array b
String[] x = new String[27] ;  // declara y crea el array x
```

Cuando sólo se declara una variable en cada declaración, los corchetes se pueden colocar después del tipo o del nombre de la variable del array, como aquí:

```
String b[] = new String[100]; // crea el array b, mejor no
String x[] = new String[27] ; // crea el array x, mejor no
```

Pero es preferible acostumbrarse a colocar los corchetes después del tipo base.

> **Nota**: *Declarar múltiples variables tipo array en una sola declaración puede provocar errores sutiles. Considera la*

*declaración **int[] a, b, c;** Si a, b y c deben declararse como variables tipo array, entonces esta declaración es correcta, ya que al colocar corchetes directamente después del tipo, indicamos que todos los identificadores son de tipo array. No obstante, si sólo a debe ser una variable tipo array mientras que b y c deben ser variables int individuales, entonces esta declaración es incorrecta; la declaración **int a[], b, c;** lograría el resultado deseado.*

Un programa puede declarar arrays de cualquier tipo. Cada elemento de un array de tipo primitivo contiene un valor del tipo del elemento declarado del array. De manera similar, en un array de un tipo de referencia, cada elemento es una referencia a un objeto del tipo del elemento declarado del array. Por ejemplo, cada elemento de un array **int** es un valor **int**, y cada elemento de un array **String** es una referencia a un objeto String.

ARRAYS Y BUCLES

La potencia de los arrays viene del hecho de que el índice para acceder a un elemento puede ser una expresión entera. Esto permite que sentencias relativamente pequeñas, puedan manipular arrays con muchos datos y accediendo de forma muy flexible a los elementos. Vamos a comenzar a ver problemas que necesitan acceder a todos los elementos de un array.

EJEMPLO 18: Procesar/recorrer todos los elementos de un array de enteros para ... (imprimirlos por ejemplo).

```
int[] lista;
lista = new int[20];  // Puedes juntar declaración y asignación
int i;                // el índice del array
for(i = 0; i < lista.length; i++) {
   System.out.println( lista[i] );
```

```
}
```

EJEMPLO 19: Supongamos que A es un array de doubles y queremos calcular la media aritmética de todos los elementos (suma de todos entre el número de ellos):

```
double[] A = new double[2 * n];
...
double suma = 0;     // La suma
double media;        // La media
for(int i = 0; i < A.length; i++ ) {
   suma += A[i];     // Sumar al total el valor del elemento
}
media = suma / A.length; // A.length es el nº de elementos
```

EJEMPLO 20: Encontrar el mayor elemento de un array de doubles A.

```
double max; // Aquí guardamos el mayor
max = A[0]; // Suponemos que el primer elemento es el mayor
for(int i = 1; i < A.length; i++ ) { // comprobamos a partir del 1
   if ( A[i] > max ) {  // Si otro es mayor, cambia el mayor
      max = A[i];
   }
}
```

Otras veces, necesitas recorrer todos los elementos del array, aunque solamente quieres trabajar con algunos de ellos (no todos) y para saber si cada i-ésimo elemento es uno de los que necesitas, debes comprobarlo. En estos casos, debes anidar un if dentro del bucle que recorre los elementos.

EJEMPLO 21: Calcular la media de los números distintos de 0 del array A (que tiene doubles).

```
double total = 0;    // La suma total
int cuenta = 0;      // Nº de valores sumados (no cero)
double media;        // La media de valores no cero
for(int i = 0; i < A.length; i++ ) {  // Recorre todos
   if ( A[i] != 0 ) {                 // Para cada uno comprueba
      total = total + A[i];           // Suma
```

```
            cuenta++;                    // como cuenta = cuenta + 1
        }
    }
    if (cuenta == 0) {
        System.out.println("No hay valores no cero.");
    }
    else {
        media = total / cuenta;
        System.out.printf("Media de %d elementos es %.5f%n",
                          cuenta, media);
    }
```

Todos los ejemplos vistos hasta ahora han usado **acceso secuencial** (se recorre el array desde la primera posición hasta la última o viceversa). Sin embargo, otra de las ventajas de los arrays es que puedes acceder a cualquier elemento sin tener que recorrer los anteriores (**acceso aleatorio**). Veamos un ejemplo que se aproveche de esto.

EJEMPLO 22: Vamos a simular un problema estadístico conocido como **la paradoja del cumpleaños**: sopongamos que hay N personas en una habitación. ¿Qué opciones hay de que dos de esas personas tengan el mismo día de cumpleaños? Es decir, hayan nacido el mismo mes y dia (no necesariamente el mismo año). Muchas personas subestiman la probabilidad, que en realidad es alta. Vamos a hacer un programa que escoja a dos personas al azar y comprobamos sus cumpleaños. ¿Cuántas personas debemos comprobar antes de que encontremos una coincidencia? Como la respuesta depende de cosas aleatorias, deberás ejecutarlo varias veces para hacerte una idea de cuánta gente necesitas comprobar, de media. Hay 365 posibles días de cumpleaños (ignoramos bisiestos).

```
/**
* Simula escoger personas aleatorias y comprobar su cumpleaños
* Si es el mismo de otra persona, para la ejecución
* mostrando cuantas personas se comprueban
*/
```

```
public class ProblemaDelCumple {

public static void main(String[] args) {
   boolean[] usado;            // Almacena posibles cumpleaños
   int contador;               // Nº de personas comprobadas
   usado = new boolean[365];   // Todos a false
   contador = 0;
   while (true) {
         // Selecciona al azar un dia (0 a 364)
         // Si ya se ha elegido antes, salir
         int cumple;      // El dia escogido
         cumple = (int)(Math.random() * 365);
         contador++;
         System.out.printf("Persona %d nace %d", cuenta, cumple);
         System.out.println();
         if ( usada[cumple] ) {    // Hay coincidencia
            break;
         }
         usada[cumple] = true;
      } // fin while
      System.out.println();
      System.out.println("Encontrada en " + contador + " veces.");
   }
} // fin clase
```

Piensa en una aplicación que deba leer una cantidad máxima de datos que no se conozca de antemano. Como mucho 100 números, pero no sabes cuántos cada vez que ejecutes el programa. ¿Cómo sabes los datos que has leído y guardado en el array? ¿Qué longitud le das? (ten en cuenta que no la puedes cambiar) ¿En qué posiciones guardas los datos?

EJEMPLO 23: Haz una aplicación que lea un máximo de 100 números int que debe almacenar en un array y luego mostrarlos en orden inverso al que se añadieron.

Como sabemos la cantidad máxima a almacenar, el array tendrá una longitud del máximo número de datos, los datos se almacenarán en las

posiciones 0 a máximo-1, y necesitamos una variable adicional que nos va a indicar dos cosas: contar los datos que hemos leído y almacenado en el array y la siguiente posición libre para guardar datos en el array (en el código variable contador).

```
import java.util.Scanner;

// Lee, almacena y muestra valores en orden inverso a como se leen
public class LeAlMuIn {

   public static void main(String[] args) {
      final int MAXIMO = 100;
      int[] numeros;    // array
      int contador;     // Nºs leídos
      int num;          // Uno de los números
      Scanner teclado = new Scanner( System.in );
      numeros = new int[MAXIMO]; // Espacio para MAXIMO ints
      contador = 0;     // No se han leído todavía
      System.out.printf("Teclee hasta %d enteros (<0 acaba).\n",
                        MAXIMO);
      while ( num >= 0 && contador < MAXIMO) {
         System.out.print("? ");
         num = teclado.nextInt();
         if (num >= 0 && contador < MAXIMO) {
            numeros[contador] = num; // Almacenar
            contador++;              // Actualizar contador
         }
      }
      System.out.println("\nLos números en orden inverso:\n");
      for (int i = contador - 1; i >= 0; i--) {
         System.out.println( numeros[i] );
      }
   } // fin main();
} // fin clase
```

Lo resaltado en amarillo es una medida de seguridad para evitar intentar guardar datos en una posición del array que no existe, lo que generaría una excepción de tipo **`ArrayIndexOutOfBoundsException`**.

VALORES INICIALES DE ARRAYS

Puedes indicar un **inicializador de arrays**, que es una lista de expresiones separadas por comas (se conoce también como **lista inicializadora**) y que está encerrada entre llaves. En este caso, la longitud del array se determina contando el número de elementos en la lista inicializadora. Por ejemplo, la declaración

```
int[] n = { 10, 20, 30, 40, 50 };
```

crea un array de 5 elementos con los valores de índices 0 a 4. El elemento n[0] se inicializa con 10, n[1] se inicializa con 20, etc. Cuando el compilador encuentra la declaración de un array que incluye una lista inicializadora, *cuenta* el número de elementos y aplica la operación new apropiada.

EJEMPLO 24: Inicialización de los elementos de un array con valores predeterminados a cero.

```
public class IniArray {
   public static void main( String[] args ) {
      int[] array = new int[10] ; // declara y crea objeto array
      // encabezados de columnas
      System.out.printf("%7s%8s%n", "Indice", "Valor");
      // imprime el valor de cada elemento del array
      for(int c = 0; c < array.length; c++)
         System.out.printf("%7d%8d%n", c, array[c] );
   }
} // fin de la clase IniArray
```

EJEMPLO 25: Inicialización de los elementos de un array con un inicializador de arrays.

```
public class Iniarray {
   public static void main( String[] args ) {
      // la lista especifica el valor de cada elemento
      int[] array = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
      // encabezados de columnas
```

```
        System.out.printf("%7s%8s%n", "Indice", "Valor");
        // imprime el valor de cada elemento del array
        for(int c = 0; c < array.length; c++)
           System.out.printf("%7d%8d%n", c, array[c]);
     }
} // fin de la clase IniArray
```

EJEMPLO 26: Rellenar los datos de un array con uno de los enteros pares del 2 al 20 (2, 4, 6, …, 20).

```
public class IniArray {
   public static void main(String[] args) {
      final int LONGITUD_ARRAY = 10; // declara la constante
      int[] a = new int[LONGITUD_ARRAY] ; // crea el array
      // calcula el valor para cada elemento del array
      for(int c = 0; c < array.length; c++)  a[c] = 2 + 2 * c;
      // encabezados de columnas
      System.out.printf("%7s%8s%n", "Indice", "Valor");
      // imprime el valor de cada elemento del array
      for(int c = 0; c < a.length; c++)
         System.out.printf("%7d%8d%n", c, a[c] );
   }
} // fin de la clase IniArray
```

***Nota**: Las variables constantes también se conocen como **constantes con nombre**. Mejoran la legibilidad de un programa, en comparación con los programas que utilizan valores literales (por ejemplo, 10); una constante con nombre como LONGITUD_ARRAY indica sin duda su propósito, mientras que un valor literal podría tener distintos significados, según su contexto.*

***Nota**: Las constantes con nombres compuestos por varias palabras deben tener cada palabra separada, una de la otra, por un guion bajo (_), como en LONGITUD_ARRAY.*

***Nota**: asignar un valor a una variable final después de inicializarla es un error de compilación. De igual forma, al tratar*

de acceder al valor de una variable final antes de inicializarla se produce un error de compilación como: "variable nombre_var might not have been initialized".

EJEMPLO 27: Suma de los elementos de un array. Por ejemplo, si los elementos del array representan las calificaciones de un examen, es probable que el profesor desee sumar el total de los elementos del array y utilizar esa suma para calcular la nota promedio.

```
public class SumaArray {
   public static void main(String[] args) {
      int[] a = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
      int total = 0;
      // suma el valor de cada elemento al total
      for(int c = 0; c < a.length; c++)
         total += a[c];
      System.out.printf("Total: %d%n", total);
    }
} // fin de la clase SumaArray
```

EJEMPLO 28: Crear el gráfico de barras de los valores de un array. Imagina que las puntuaciones de un examen fueron 87, 68, 94, 100, 83, 78, 85, 91, 76 y 87. Queremos ver como son los resultados agrupando los valores en intervalos (una calificación de 100, dos calificaciones en el rango de 90 a 99, cuatro calificaciones en el rango de 80 a 89, dos en el rango de 70 a 79, una en el rango de 60 a 69 y ninguna por debajo de 60). Tendremos el array de notas llamado a y otro array llamado d donde anotaremos las distribuciones.

```
public class GraficoBarras {
   public static void main(String[] args) {
      int[] a = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
      // Serías capaz de escribir el código que genera d?
      int[] d = { 0, 0, 0, 0, 0, 0, 1, 2, 4, 2, 1 };
      System.out.println("Distribucion de calificaciones:");
      // para cada elemento de a, imprime una barra del gráfico
      for(int c = 0; c < d.length; c++) {
```

```
            // Etiqueta de la barra ("00-09: ",.., "90-99: ","100: ")
            if (c == 10)
              System.out.printf("%5d: ", 100);
            else
              System.out.printf("%02d-%02d: ", c * 10, c * 10 + 9);
            // imprime barra de asteriscos
            for (int e = 0; e < d[c]; e++)
              System.out.print("*");
            System.out.println();
          }
      }
} // fin de la clase GraficoBarras
```

EJERCICIO 3: escribe un trozo de código que cree y rellene el array d a partir de los datos que haya en el array a del ejemplo anterior.

EJEMPLO 29: En ocasiones, los programas utilizan variables tipo contador para sintetizar datos, como los resultados de una encuesta. Si queremos tirar un dado muchas veces y contar cuantas veces aparece cada valor, podemos usar contadores o como en este ejemplo, un array.

```
import java.security.SecureRandom;

public class TirarDado {
   public static void main(String[] args)  {
      SecureRandom na = new SecureRandom();
      int[] f = new int[7]; // array de contadores de frecuencia
      // tira el dado 6,000,000 veces
      for(int tiro = 1; tiro <= 6000000; tiro++)
         ++f[ 1 + na.nextInt(6) ];
      System.out.printf( "%s%10s%n", "Cara ", "Frecuencia");
      // imprime el valor de cada elemento del array
      for(int i = 1; i < f.length; i++)
          System.out.printf("%4d %10d%n", i, f[i] );
    }
} // fin de la clase TirarDado
```

EJEMPLO 30: Contar las respuestas a una encuesta que son números entre 1 y 5 y en caso de respuestas incorrectas usamos la posición 0 del array de frecuencias f. *El programa también utiliza un* mecanismo de *manejo de excepciones* de Java, para detectar y manejar una excepción **ArrayIndexOutOfBoundsException**.

```
public class Encuesta {
   public static void main(String[] args) {
     // array respuestas (lo normal es rellenar en t.ejecución)
     int[] r = {1, 2, 5, 4, 3, 5, 2, 1, 3, 3, 1, 4, 3, 3, 3, 14 };
     int[] f = new int[6]; // array de contadores de frecuencia
     for(int res = 0; res < r.length; res++) {
       try {
          ++f[ r[res] ];
       } catch(ArrayIndexOutOfBoundsException e) {
          System.out.println(e.getMessage());
          System.out.printf("respuestas[%d] = %d%n%n",res,r[res]);
       }
     }
   System.out.printf("%s%10s%n", "Nota", "Frecuencia");
   // imprime el valor de cada elemento del array
   for (int k = 1; k < f.length; k++)
      System.out.printf("%6d%10d%n", k, f[k]);
   }
} // fin de la clase Encuesta
```

ARRAYS DE DOS DIMENSIONES

Los arrays que hemos usado hasta ahora se dice que tienen una dimensión (son una secuencia de valores que pueden ponerse uno debajo de otro en una columna). También es posible tener arrays bidimensionales (una tabla de valores). Por ejemplo la siguiente figura representa un array de 5 filas y 7 columnas. Ahora para acceder a cada valor, es necesario usar dos índices, uno para cada dimensión, el primero de los índices indica la fila (en vertical) y el otro indica la columna (en horizontal).

Este array de dimensiones 5x7 (una tabla), contiene 35 valores. Es

como un array de 5 elementos (filas de la 0 a la 4), donde cada elemento es otro array de 7 elementos (columnas de la 0 a la 6).

	0	1	2	3	4	5	6
0	13	7	33	54	-5	-1	92
1	-3	0	8	42	18	0	67
2	44	78	90	79	-5	72	22
3	43	-6	17	100	1	-12	12
4	2	0	58	58	36	21	87

Figura 3: Un array de dos dimensiones (fila y columna)

En Java, la sintaxis para declarar el array 2D es similar al de una dimensión. Se indica el tipo base y a continuación el operador array una vez por cada dimensión:

```
int[][] tabla;
```

Y al instanciar se indica de la longitud de cada dimensión del array:

```
tabla = new int[filas][columnas];
```

EJEMPLO 31: Mostrar el contenido de una tabla de 5 filas y 7 columnas:

```
int fila, colu;  // variables indices
for ( fila = 0; fila < 5; fila++ ) {
   for ( colu = 0; colu < 7; colu++ ) {
      System.out.printf( "%7d", A[fila][colu] );
   }  // fin bucle interno
   System.out.println();
}     // fin bucle externo
```

Igual que antes, el tipo base puede ser cualquiera, incluyendo otro array. Lo que nos daría arrays tridimensionales. En Java, el límite en

las dimensiones de un array es 8.

EJEMPLO 32: Considera una empresa que tiene 5 tiendas (numeradas de 0 a 4). Imagina que de cada tienda se almacena el beneficio de cada mes del año 2019 (0 a 11). Usamos un array 2D para almacenar estos datos:

```
double[][] beneficio;
beneficio = new double[25][12];
```

Imagina que el array ya tiene elementos rellenados.

a) Un programa que calcule el beneficio total de toda la empresa:

```
double total = 0;
int tienda, mes; // variables índice
for( tienda = 0; tienda < 5; tienda++ ) {
    for( mes = 0; mes < 12; mes++ )
        total+= beneficio[tienda][mes];
}
```

b) Beneficios de todas las tiendas en diciembre:

```
double total = 0.0;
int tienda;
for( tienda = 0; tienda < 5; tienda++ ) {
    total+= beneficios[tienda][11];
}
```

EJERCICIO 4: Haz el fragmento de código que calcule el beneficio de la tienda 3 durante todo el año 2019.

3.3.2 ARRAYS MULTIDIMENSIONALES.

Los arrays multidimensionales son los que utilizan más de un índice para acceder a los elementos. Los de dos dimensiones se utilizan a menudo para representar *tablas* de valores, con datos organizados en *filas* y *columnas*. Para identificar un elemento específico de una tabla,

debemos especificar *dos* índices.

Por convención, **el primero identifica la fila** del elemento y **el segundo su columna**. Los arrays que necesitan dos índices para identificar un elemento se llaman **arrays bidimensionales** (los arrays multidimensionales pueden tener más de dos dimensiones).

Java no soporta los arrays multidimensionales directamente, pero permite al programador especificar arrays unidimensionales, cuyos elementos sean también arrays unidimensionales, con lo cual se obtiene algo equivalente.

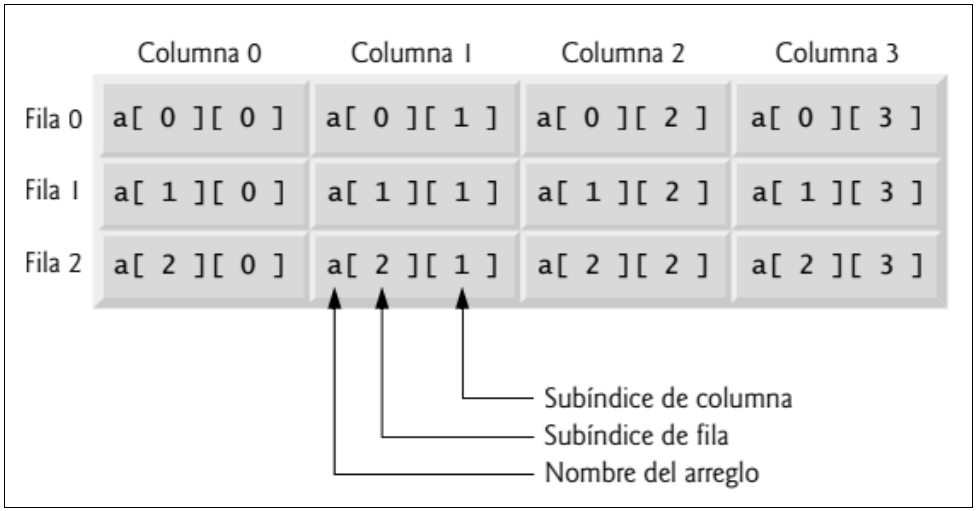


Figura 4: Array bidimensional.

La figura 4 ilustra un array bidimensional llamado a, que contiene 3 filas y 4 columnas (es decir, un array de tres por cuatro). En general, a un array con *m* filas y *n* columnas se le llama **array de *m* por *n***.

Cada elemento en el array a se identifica mediante una *expresión de acceso a un array* de la forma **a[*fila*][*columna*];** a es el nombre del

array, *fila* y *columna* son los índices que identifican de forma única a cada elemento por índice de fila y columna. Todos los nombres de los elementos en la *fila* 0 tienen un *primer* índice de 0, y los nombres de los elementos en la *columna* 3 tienen un segundo índice de 3.

Al igual que los arrays unidimensionales, los multidimensionales pueden inicializarse mediante inicializadores en las declaraciones. Un array bidimensional b con dos filas y dos columnas podría declararse e inicializarse con **inicializadores de arrays anidados**, como se muestra a continuación:

```
int[][] b = { {1, 2, 3}, {3, 4, 5} };
```

Los valores iniciales se *agrupan por fila* entre llaves. Por lo tanto, 1, 2 y 3 inicializan a b[0][0], b[0][1] y b[0][2], respectivamente; 3, 4 y 5 inicializan a b[1][0], b[1][1] y b[1][2], respectivamente. El compilador cuenta el número de inicializadores de arrays anidados (representados por conjuntos de llaves dentro de las llaves externas) para determinar el número de *filas* en el array b. El compilador cuenta los valores inicializadores en el inicializador de arrays anidado de una fila, para determinar el número de *columnas* en esa fila. Como veremos en unos momentos, esto significa que *las filas pueden tener arrays de distintas longitudes o columnas*.

Los arrays multidimensionales son *arrays de arrays unidimensionales*. Por lo tanto, el array b en la declaración anterior en realidad está compuesto de dos arrays unidimensionales independientes: uno que contiene los valores en la primera lista inicializadora anidada {1, 2, 3} y otro que contiene los valores en la segunda lista inicializadora anidada {3, 4, 5}. Así, el array b en sí es un array de dos elementos, cada uno de los cuales es un array unidimensional de valores int.

La forma en que se representan los arrays multidimensionales los hace bastante flexibles. De hecho, **las longitudes de las filas en el array b *no* tienen que ser iguales**. Por ejemplo,

```
int[][] b = { {1, 2}, {3, 4, 5} };
```

crea el array entero b con dos elementos (los cuales se determinan según el número de inicializadores de arrays anidados) que representan las filas del array bidimensional. Cada elemento de b es una *referencia* a un array unidimensional de variables **int**. El array int de la fila 0 es un array unidimensional con *dos* elementos (1 y 2), y el array int de la fila 1 es un array unidimensional con *tres* elementos (3, 4 y 5).

Un array multidimensional con el *mismo* número de columnas en cada fila puede formarse mediante una expresión de creación de arrays. Por ejemplo, en las siguientes líneas se declara el array b y se le asigna una referencia a un array de tres por cuatro:

```
int[][] b = new int[3][4];
```

En este caso, utilizamos los valores literales 3 y 4 para especificar el número de filas y columnas, respectivamente, pero esto *no* es obligatorio. Los programas también pueden utilizar variables para especificar las dimensiones de los arrays, ya que ***new*** *crea arrays en tiempo de ejecución, no en tiempo de compilación.*

Los elementos de un array multidimensional se inicializan cuando se crea el objeto array.

ARRAYS ASIMÉTRICOS

Un array multidimensional, en el que cada fila tiene un número *distinto* de columnas, puede crearse de la siguiente manera: indicas la cantidad

de filas pero no el de columnas y en cada fila instancias otro array.

```
int[][] b = new int[2][]; // crea array de 2 filas, no dices nºcol
b[0] = new int[5];        // crea array:5 columnas para la fila 0
b[1] = new int[3];        // crea array:3 columnas para la fila 1
```

Estas instrucciones crean un array bidimensional con dos filas. La fila 0 tiene *cinco* columnas (un array de 5 columnas) y la fila 1 tiene *tres columnas*.

Observa que al crear arrays bidimensionales asimétricos, no debes indicar tamaño en la dimensión que puede cambiar de tamaño.

Cuando se procesan arrays simétricos o asimétricos, hay que ser cuidadoso en que se accede a elementos con los índices correctos.

Pero si son unidimensionales y almacenan objetos, o son multidimensionales hay que asegurarse además de que los índices sean correctos, de que exista el elemento, pues podría no haber ningún objeto almacenado, o ningún array definido si el array es asimétrico. Para ello podemos aprovecharnos de la **evaluación en cortocircuito** de las expresiones lógicas.

EJEMPLO 33: Crear un array bidimensional de forma aleatoria e imprimir el total de elementos que almacena. Este programa podría tener errores de ejecución algunas veces y otras veces podría funcionar:

```
public class FalloProbable {

  public static void main( String[] args ) {
      int[][] a = new int[5][]; // crea array de 5 arrays a null
      for(int i= 0; i < 5; i++) {
          double azar = Math.random();
          if( azar < 0.5 ) a[i]= new int[ (int)(azar * 10 + 1) ];
      }
      System.out.println("El array puede almacenar...");
```

```
        int suma = 0;
        for(int i= 0; i < 5; i++)
            suma += a[i].length; // Que pasa si a[i] tiene null?
        System.out.println(suma + " elementos.");
    }
```

EJEMPLO 34: Crear un array bidimensional de forma aleatoria e imprimir el total de elementos que almacena.

```
public class FalloControlado {

  public static void main( String[] args ) {
      int[][] a = new int[5][]; // crea array de 5 arrays a null
      for(int i= 0; i < 5; i++) {
          double azar = Math.random();
          if( azar < 0.5 ) a[i]= new int[ (int)(azar * 10 + 1) ];
      }
      System.out.println("El array puede almacenar...");
      int suma = 0;
      for(int i= 0; i < 5; i++)
          if(a[i] != null) suma+= a[i].length; // a[i] tiene array
      System.out.println(suma + " elementos.");
  }
```

EJEMPLO 35: Inicializar arrays bidimensionales con inicializadores de arrays, y utilizar bucles for anidados para **recorrer** los arrays (es decir, manipular *cada uno* de los elementos de cada array).

```
public class IniArray {

  public static void main( String[] args ) {
     int[][] a1 = { {1, 2, 3}, {4, 5, 6} };
     int[][] a2 = { {1, 2}, {3}, {4, 5, 6} };
     System.out.println("Los valores en a1 por filas son");
     imprimirArray(a1); // muestra array1 por filas
     System.out.println("%nLos valores en a2 por filas son%n");
     imprimirArray(a2) ; // muestra array2 por filas
  }

  // imprime filas y columnas de un array bidimensional
  public static void imprimirArray(int[][] a) {
     // itera a través de las filas del array
```

```
        for(int f = 0; f < a.length; f++) {
           // itera a través de las columnas de la fila actual
           for(int c = 0; c < a[f].length; c++)
              System.out.printf("%d ", a[f][c] );
           System.out.println();
        }
     }
} // fin de la clase IniArray
```

EJEMPLO 36: Utiliza un array bidimensional llamado **cali**, para almacenar las calificaciones de *un número* de estudiantes en *varios* exámenes. Cada *fila* del array representa las calificaciones de un *solo* estudiante durante todo el curso, y cada *columna* representa las calificaciones de *todos* los estudiantes que presentaron un examen específico.

```
public class LibroCali {
   private String nombreCurso; // nombre del curso
   private int[][] cali; // array bidimensional de calificacion
   // el constructor de 2 argumentos inicializa nombre y califica.
   public LibroCali(String curso, int[][] cal) {
      this.nombreCurso = curso;
      this.cali = cali;
   }

   // método para establecer el nombre del curso
   public void setNombreCurso(String nombre) {
      this.nombreCurso = nombre;
   }

   // método para obtener el nombre del curso
   public String getNombreCurso() {
      return nombreCurso;
   }

   // realiza varias operaciones sobre los datos
   public void procesarCalificaciones() {
      // imprime el array de calificaciones
      imprimirCalificaciones();
      // llama a los métodos obtenerMinima y obtenerMaxima
      System.out.printf("%n%s %d%n%s %d%n%n",
```

```
                " La calificacion más baja es", getMinima(),
                " La calificacion más alta es", getMaxima() );
        // imprime gráfico de distribución
        imprimirGraficoBarras();
    }

    // busca la calificación más baja
    public int getMinima() {
        // Supone que el primer elemento es el más bajo
        int baja = cali[0][0];
        for(int[] cali: c) { // itera filas
            for(int c: ce) {  // itera columnas
                // si calificación menor que baja, es la menor
                if (ce < baja)
                  baja = ce;
            }
        }
        return baja;
    }

    // busca la calificación más alta
    public int getMaxima() {
      // asume que el primer elemento es el más alto
      int alta = cali[0][0];
      for(int[] ce: cali) { // itera filas
            for(int ca: ce) {  // itera columnas de la fila actual
               // si calificación es mayor que alta, cambia alta
               if (ca > alta)
                  alta = ca;
            }
      }
      return alta;
    }

    // determina promedio para un conjunto específico
    public double getPromedio(int[] cjto) {
        int total = 0;
        // suma las calificaciones para un estudiante
        for(int ca : cjto)
            total += ca;
        // devuelve el promedio de calificaciones
        return (double) total / cjto.length;
    }
```

```
// imprime gráfico de barras
public void imprimirGraficoBarras() { // visto antes...
```

3.3.3 LA CLASE ARRAYS.

Gracias a la clase **Arrays** no tenemos que reinventar la rueda, ya que proporciona **métodos static** para las operaciones comunes de arrays. Estos métodos incluyen `sort(array)` para *ordenar* un array (es decir, acomodar los elementos en orden ascendente), `binarySearch(array, valor)` para *buscar* en un array *ordenado* (determinar si un array contiene un valor específico y, de ser así, en dónde se encuentra este valor), `equals(array1, array2)` para *comparar* arrays y **fill(array,valor)** para *colocar valores en un array o* ***`fill(array, desde, hasta, valor)`***, `copyOf(array, nuevaLongitud)`, `copyRange(array, desde, hasta)` y *sus variantes* ***deepX*** *(para cuando son multidimensionales, aplicar la operación a los elementos interiores)*. Estos métodos están sobrecargados para los arrays de tipo primitivo y los arrays de objetos.

EJEMPLO 37: usar los métodos sort, binarySearch, equals y fill de la clase Arrays, para copiar arrays con el método **arraycopy()** de la clase System.

```
import java.util.Arrays;

public class ManiArrays {
   public static void main( String[] args ) {
      // ordena aDouble en forma ascendente
      double[] aDouble = { 8.4, 9.3, 0.2, 7.9, 3.4 };
      Arrays.sort(aDouble);
      System.out.printf("%naDouble: ");
      for(double valor : aDouble)
         System.out.printf("%.1f ", valor);
      // llena array de 10 elementos con 7
      int[] aInt = new int[10] ;
```

```
        Arrays.fill(aInt, 7);
        mostrararray(aInt, "aInt");
        // copia el array aInt en el array copiaInt
        int[] aInt = { 1, 2, 3, 4, 5, 6 };
        int[] copiaInt = new int[aInt.length];
        System.arraycopy(aInt, 0, copiaInt, 0, aInt.length);
        mostrararray(aInt, "aInt");
        mostrararray(copiaInt, "copiaInt");
        // compara si aInt y copiaInt son iguales
        boolean b = Arrays.equals(aInt, copiaInt);
        System.out.printf("%n%naInt %s copiaInt%n",
                          (b ? "==" : "!="));
        // compara si arrayInt y arrayIntLleno son iguales
        b = Arrays.equals(aInt, aInt);
        System.out.printf("aInt %s aInt%n", (b ? "==" : "!="));
        // busca en arrayInt el valor 5
        int ubicacion = Arrays.binarySearch(aInt, 5);
        if (ubicacion >= 0)
           System.out.printf(
             "Se encontro 5 en el elemento %d de aInt%n",
             ubicacion);
        else
           System.out.println("No se encontro el 5 en aInt");
        // busca en arrayInt el valor 8763
        ubicacion = Arrays.binarySearch(aInt, 8763);
        if (ubicacion >= 0)
            System.out.printf("8763 en elemento %d de aInt%n",
                              ubicacion);
        else
           System.out.println("No se encontro 8763");
     }

     // imprime los valores en cada array
     public static void mostrarArray(int[] a, String descrip) {
        System.out.printf("%n%s: ", descrip);
        for(int valor : a)
           System.out.printf("%d ", valor);
     }
  }
```

***Nota**: en Java SE 8 el método **parallelSort()** la clase Arrays tiene varios métodos "paralelos" que sacan provecho del*

hardware multinúcleo. El método parallelSort() puede ordenar los arrays grandes con más eficiencia en sistemas multinúcleo.

3.4. PROCESAR STRINGS CON BUCLES.

Los bucles también nos facilitan la manipulación de cadenas. Si lo piensas, un String es similar a un array de caracteres, aunque en realidad no son lo mismo.

```
char[] ac = new char[20];    // Array de 20 caracteres
String s = "aeiou";          // String de 5 caracteres
```

Pero tienen una estructura similar (elementos consecutivos) y podemos acceder a los caracteres que lo forman a través de los métodos de la clase String como `charAt(posición)`, por tanto se pueden manipular de forma similar. Pero String tiene muchos más métodos para ayudarnos a manipular cadenas.

EJEMPLO 38: contar cuantos espacios en blanco tiene una cadena.

```
String s = "Buenos dias a todos y todas";
int cantidadEspacios = 0;
for(int i = 0; i < s.length(); i++) {
    if( s.charAt(i) == ' ' )
        cantidadEspacios++;
}
```

EJEMPLO 39: contar la cantidad de vocales que tiene un String.

```
String s = "Buenos dias a todos y todas";
int cantidadVocales = 0;
for(int i = 0; i < s.length(); i++) {
    if( "aeiouAEIOU".contains( "" + s.charAt(i) )
        cantidadVocales++;
}
```

EJEMPLO 40: Generar una nueva cadena eliminando letras repetidas de otra.

```
String s = "Buenos dias a todos y todas";
String sinRepetidos = "";
for(int i = 0; i < s.length(); i++) {
    char letra = s.charAt(i);
    if( sinRepetidos.indexOf( letra ) < 0 )
        sinRepetidos += letra;
}
```

EJEMPLO 41: Averiguar si una cadena solo contiene letras que esté, en otra cadena (un alfabeto) en cuyo caso se da por correcta.

```
String alfabetoBinario = "01";
String s = "01001141100";
boolean correcta = true;
for(int i = 0; i < s.length(); i++) {
    if( alfabeto.indexOf( s.charAt(i) ) < 0 )
        correcta= false;
}
if(correcta)
    System.out.println("Es correcta");
else
    System.out.println("No es correcta");
```

EJEMPLO 42: El ejemplo anterior funciona, pero es mejorable. Si encuentra una letra que no es del alfabeto, continúa recorriendo la cadena comprobando las letras que faltan hasta el final de la cadena, cuando en realidad debería dejar de hacerlo porque ya sabe que no es correcta. Modificar la condición de parada para evitar ese trabajo inútil.

```
String alfabetoBinario = "01";
String s = "0100114";
boolean correcta = true;
for(int i = 0; correcta && i < s.length(); i++) {
    if( alfabeto.indexOf( s.charAt(i) ) < 0 )
        correcta= false;
}
if(correcta)
    System.out.println("Es correcta");
else
```

```
        System.out.println("No es correcta");
```

EJEMPLO 43: El mismo ejercicio de antes pero ahora salimos directamente del bucle con un break.

```
String alfabetoBinario = "01";
String s = "0100114";
boolean correcta = true;
for(int i = 0; i < s.length(); i++) {
    if( alfabeto.indexOf( s.charAt(i) ) < 0 ) {
        correcta= false;
        break;
    }
}
if(correcta)
    System.out.println("Es correcta");
else
    System.out.println("No es correcta");
```

EJEMPLO 44: Invertir las letras de un String en un nuevo String.

```
String s = "La casa de papel ha tenido mucho éxito";
String invertida = "";
for(int i = 0; i < s.length(); i++)
    invertida = s.charAt(i) + invertida;
System.out.println( invertida );
```

EJERCICIO 5: Repite el ejemplo 42 pero ahora recorre la cadena a invertir desde el final hasta el principio.

EJEMPLO 45: Averiguar si una frase sin espacios en blanco es simétrica (palíndroma es equivalente a capicúa para un número, es decir, contiene las mismas letras si comenzamos de izquierda a derecha que de derecha a izquierda).

```
String s = "abccba";
String invertida = "";
for(int i = 0; i < s.length(); i++)
```

```
        invertida = s.charAt(i) + invertida;
    // Comparamos la original con la invertida
    if( s.equals( invertida ) )
        System.out.println("Es palíndroma");
    else
        System.out.println("No es palíndroma");
```

EJEMPLO 46: Averiguar si una frase a la que quitamos los espacios en blanco es palíndroma sin necesidad de invertirla.

```
String s = "dabale arroz a la zorra el abad";
// Generar su versión sin espacios
String sinEspacios = "";
for(int i = 0; i < s.length(); i++)
    if( s.charAt(i) != ' ' ) sinEspacios += s.charAt(i);
// La primera y la última iguales, la segunda y penúltima...
int izq = 0;                            // Letra a la izquierda
int der = sinEspacios.length()-1;       // Letra a la derecha
boolean palindroma = true;
while(palindroma && izq < der) {
    if( sinEspacios.charAt(izq) != sinEspacios.charAt(der) )
         palindroma = false;
    else {
        izq++;
        der--;
    }
}
// Comparamos el centinela
if( palindroma )
    System.out.println("Es palíndroma");
else
    System.out.println("No es palíndroma");
```

EJEMPLO 47: Repite el ejercicio anterior pero sin usar la variable centinela.

```
String s = "dabale arroz a la zorra el abad";
// Generar su versión sin espacios
String sinEspacios = "";
for(int i = 0; i < s.length(); i++)
    if( s.charAt(i) != ' ' ) sinEspacios += s.charAt(i);
// La primera y la última iguales, la segunda y penúltima...
```

```
int izq = 0;                            // Letra a la izquierda
int der = sinEspacios.length()-1;       // Letra a la derecha
while( izq < der ) {
    if( sinEspacios.charAt(izq) != sinEspacios.charAt(der) )
         break;
    else {
        izq++;
        der--;
    }
}
// Usamos la condición de parada del bucle
if( izq >= der )
    System.out.println("Es palíndroma");
else
    System.out.println("No es palíndroma");
```

3.5. EJERCICIOS.

E1. TEST

1. ¿Qué es un bloque de sentencias? ¿Cómo se hacen en Java?

2. ¿Cuál es la principal diferencia entre un bucle **while** y un **do-while**?

3. ¿Qué precauciones hay que tener al acceder a un elemento de un array unidimensional que almacena tipos primitivos? ¿Y si almacena objetos o es multidimensional asimétrico?

4. Escribe un bucle que muestre todos los múltiplos del número 3 desde el 3 hasta el 36.

5. Rellena la subrutina main() para que pregunte al usuario por un entero, lo lea e indique si es par o impar (recuerda que un número es par si al dividirlo por 2 su resto es 0, es decir, se puede dividir por 2)

6. Escribe un fragmento de código que genere dos números aleatorios entre 1 y 10 (todos los valores con la misma probabilidad, pero diferentes).

7. Imagina que s1 y s2 son variables de tipo String, cuyos valores se espera que contengan representación de números enteros de tipo int. Escribe un fragmento de código que calcule e imprima su suma o un mensaje de error si los valores no pueden convertirse de forma correcta a enteros (Usa la sentencia **try-catch**).

8. Muestra la salida que genera esta subrutina main():

```
public static void main(String[] args) {
   int N;
   N = 1;
   while (N <= 32) {
      N = 2 * N;
      System.out.println(N);
   }
}
```

9. Muestra la salida producida por esta subrutina main():

```
public static void main(String[] args) {
   int x, y;
   x = 5;
   y = 1;
   while (x > 0) {
      x = x - 1;
      y = y * x;
      System.out.println(y);
   }
}
```

10. ¿Qué salida genera el siguiente fragmento de código?

```
String name;
int i;
boolean startWord;
name = "Richard M. Nixon";
startWord = true;
for (i = 0; i < name.length(); i++) {
   if (startWord)
      System.out.println( name.charAt(i) );
```

```
        if ( name.charAt(i) == ' ' )
            startWord = true;
        else
            startWord = false;
    }
```

11. Imagina que tienes un array de números enteros (int[]). Escribe un trozo de código que cuente y muestre el número de veces que aparece el número 42 en el array.

12. Se define el rango de un array de números como el máximo valor menos el mínimo valor. Escribe un trozo de código que le calcule el rango del array de doubles llamado raceTimes.

E2. ¿Cuántas veces debes lanzar dos dados hasta que te salga **Snake eyes** (ojos de serpiente son dos unos)? Haz un programa que simule lanzamientos de dos dados y cuente el número de tiradas hasta que te salgan dos unos.

> ***Nota****: en teoría, cada lanzamiento genera resultados de tipo (1,1), (1,2), (1,3) …. (6,5), (6,6). Por tanto tienes 36 resultados posibles, todos igual de probables, así que la probabilidad de sacar (1,1) es 1/36. Por tanto, en teoría necesitas 36 lanzamientos. Pero la realidad al ser aleatoria…*

E3. ¿Qué número entero entre 1 y 10000 tiene el mayor número de divisores y cuántos tiene? Escribe un programa que encuentre la solución y muestre los resultados.

> ***Nota:*** *es posible que haya varios con el mismo número de divisores, así que tu programa solamente imprimirá uno de ellos. La idea básica es ir probando cada entero y para cada uno, contar sus divisores. Quédate con el primer número que encuentres que más tenga.*

E4. Escribe un programa que lea y calcule expresiones simples como por ejemplo `17 + 3` y `3,14159 * 4,7`, es decir, del tipo "numero <operación> número". El programa lee una expresión, calcula y muestra el resultado y lee la siguiente expresión. El programa acaba cuando el primer número leído sea cero.

> ***Nota:*** *Las expresiones son tecleadas por el usuario y siempre deben seguir el formato* ***<nº> <operación> <nº>****. Las operaciones pueden ser +, -, * y /.*

E5. Escribe un programa que lea una frase y la rompa en palabras. Por ejemplo si lee la frase "Esto no es una buena idea", su salida será:

```
Esto
no
es
una
buena
idea
```

a) Usando el método split() de la clase String.
b) Sin usar el método split().

E6. Imagina que un fichero contiene información sobre las ventas de una empresa en varias ciudades. Cada línea del fichero tiene diferentes campos separados por el carácter dos puntos (:). Un fichero de prueba tiene este formato: <ciudad>:<ventas>. Imagina que tenemos este fichero llamado **ventas.dat**:

```
Valencia:1000,11
Alicante:dato no recibido
Castellón de la Plana: 2000,22
```

Escribe un programa que calcule las ventas totales de la empresa, indicando el total de ciudades en el fichero, el número de ciudades de

las que hay datos de ventas y el total.

> ***Nota:*** *tendrás que llamar al programa redirigiendo desde la línea de comandos la entrada estándar (C:\programa < fichero) y leer una línea, romper los dos campos e intentar convertir el valor numérico en un* ***try-catch*** *por si da error de conversión, ignorando el dato en ese caso o totalizarlo en el caso contrario.*

E7. Volver a realizar el programa que encuentra el número que tenga mayor número de divisores (ejercicio E3), pero ahora vas a usar un array de enteros y cada vez que cuentes los divisores de un número, guardas la cuenta en el array. Después de rellenar el array buscas el mayor valor que almacena, imagina que es 64. Cuando tengas el mayor, imprime todos los números que tengan 64 divisores.

> ***Nota****: Las posiciones del array se corresponden con cada número entre 1 y 10000. El valor almacenado en cada posición es el número de divisores que tiene cada número. Deberías obtener esta salida:*
>
> ```
> El máximo número de divisores entre 1 y 10000 son 64
> Los números que tienen 64 divisores son:  7560, 9240
> ```

E8. En matemáticas, la letra griega sigma mayúscula (nuestra letra S), se utiliza para indicar la operación suma de muchos elementos. Cada elemento se representa mediante una expresión que puede incluir un índice (normalmente se utiliza la letra i,j,k). El sigma tiene debajo donde comienza el índice y arriba donde acaba. Por ejemplo:

$$\sum_{i=1}^{10} i \;= 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10$$

Es una forma de representar un bucle que hace una suma de

expresiones que contienen el índice (i en el ejemplo) y que se puede interpretar como: para i comenzando a valer 1 y acabando en 10, se hace la suma de i.

En matemáticas, para representar un bucle que calcule no la suma, si no el producto de muchas expresiones, se utiliza la letra griega pi mayúsculas (nuestra P) de la misma forma que sigma es suma. Ejemplo:

$$\prod_{i=1}^{10} i = 1 * 2 * 3 * 4 * 5 * 6 * 7 * 8 * 9 * 10$$

Haz un programa que pregunte por un número double llamado **x**, que representa un ángulo expresado en radianes y calcule su seno utilizando los primeros 17 términos de la siguiente serie (descomposición de Taylor):

$$\sin x \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

> **Nota**: debes buscar patrones cuyo comportamiento puedas reproducir en un bucle. Observa por ejemplo, que las fracciones tienen exponentes y denominadores impares, que a veces suman y a veces restan...

E9. Escribe un programa que pregunte por un número entero n y cree tres matrices llamadas a, b y c (arrays bidimensionales de doubles) de nxn, pida los datos de a y b al usuario, y en c guarde su suma. Cada elemento i,j de C es:

> Cij = Aij + Bij

E10. Escribe un programa que pregunte por las dimensiones de una matriz a (afilas, acolumnas) la creas y por las de una matriz b (bFilas,

bColumnas). Si se pueden multiplicar, crea la matriz c que almacene la matriz producto de a x b, lee los datos de las matrices a y b y realiza la multiplicación. Si no se pueden sumar, lo indicas. Ten en cuenta que:

- El número de filas de B tiene que ser igual al número de columnas de A.
- La matriz producto (C = A x B) tendrá el número de filas de A y el número de columnas de B.
- Cada elemento de C, Cij = multiplicar la fila i de A por la columna j de B.

Ejemplo:

$$\begin{pmatrix} 1 & 2 \\ -2 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 2 \\ 0 & 2 & 0 \end{pmatrix} =$$
$$= \begin{pmatrix} 1+0 & 0+4 & 2+0 \\ -2+0 & 0+0 & -4+0 \end{pmatrix} =$$
$$= \begin{pmatrix} 1 & 4 & 2 \\ -2 & 0 & -4 \end{pmatrix}$$

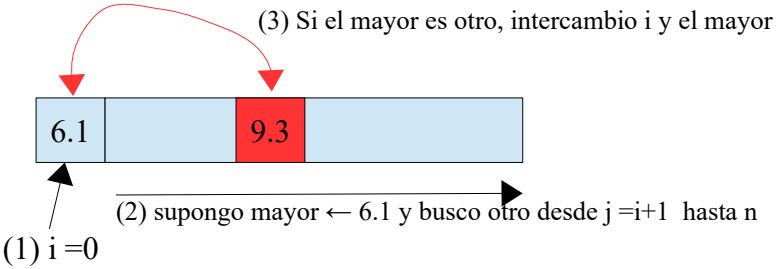
E11: Pregunta por un número entero n (que es el número de alumnos de una clase) y crea dos arrays de ese tamaño, uno llamado nombres que almacene Strings y otro llamado notas que almacene números con decimales. A continuación rellena los arrays con datos:

```
Nombre del alumno 0: Manolo
Nota: 6,1
   :
Nombre del alumno 4: Santi
Nota: 9,3
```

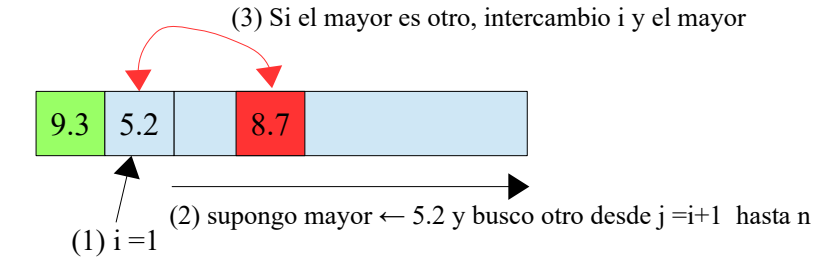
Ahora, ordena de mayor a menor nota los datos de sus alumnos de la siguiente manera:

Recorre todas las posiciones desde la primara a la última-1 y recuerda en una variable i en qué posición estás. Suponiendo que la mayor nota es la de la posición i, busca si hay otra mayor en las posiciones i+1, i+2 ..

hasta el final y si la encuentras recuerda donde está (digamos pmayor). Si la posición del mayor es distinta de i, intercambia los datos de los dos arrays (nombres[i] <-> nombres[pmayor]; y notas[i] <-> notas[pmayor]). Incrementas i y vuelves a repetir el proceso con la siguiente.



En un paso ya tengo ordenado el primer elemento, voy a por el segundo, i <- i+1



Si repites el proceso n-1 veces, para las posiciones de la 0 (el principio) hasta la n-2 (la anterior al final) acabarás ordenando la lista de mayor a menor. Cuando tengas los arrays ordenados los imprimes:

Santi (9) Manolo (7)

E12: Crea una tabla de 10x10 enteros y la rellenas con valores aleatorios de 0 hasta 10. Encuentra el valor que más se repite e indica

el porcentaje de celdas de la tabla donde está almacenado con respecto del total de la tabla. Da prioridad a la rapidez sobre el consumo de memoria.

E13: Crea un array de 10 enteros y lo rellenas con valores aleatorios entre 0 y 10. Busca el número 5 indicando la posición donde se encuentre o una posición imposible si no lo encuentra y para ello debes adaptar el método de búsqueda dicotómico expresado en el siguiente algoritmo en pseudocódigo.

```
ALGORITMO DE BÚSQUEDA DICOTÓMICA
PREREQUISITOS: a debe estar ordenado de menor a mayor
ENTRADAS:
    •   ARRAY a
    •   elemento a buscar e
INICIO
    Definir primero, ultimo, mitad como enteros;
    definir buscando como logico;
    buscando <- verdadero;
    primero <- 0;
    ultimo <- longitud del array
    mientras primero <= ultimo Y buscando hacer
        mitad <- (primero + ultimo) / 2;
        si e = a[mitad] entonces
            buscando <- FALSO:
        sino
            si e < a[mitad] entonces
                ulltimo <- mitad – 1;
            sino
                si e > a[mitad] entonces
                    primero <- mitad + 1;
                finsi
            finsi
       finsi
    finmientras
    si buscando entonces
        escribir e no encontrado;
    sino
        escribir e encontrado;
    finsi
FIN
```

```
El array a=[0, 0, 0, 3, 3, 6, 6, 7, 8, 10]
5 no encontrado
```

```
El array a=[0, 2, 3, 4, 5, 6, 7, 7, 9, 9]
5 encontrado en posición 4
```