TEMA 2: ALGORITMIA. ELEMENTOS DE PROGRAMA.

1.- INTRODUCCIÓN A LA ALGORITMIA

La noción de algoritmo es básica en programación de ordenadores, ya que para realizar un programa es necesario el diseño previo de un algoritmo. Por tanto, debemos empezar con un cuidadoso análisis de este concepto.

El término proviene del matemático persa Mohamed Al-khowârizmî que alcanzó gran reputación por el enunciado de reglas paso a paso para sumar, restar, multiplicar y dividir números decimales. Euclides, el matemático griego que inventó un método para encontrar el máximo común divisor de dos números, se considera el otro gran padre de los algoritmos.

Un **algoritmo** es un método para resolver un problema. Más explícitamente, es un conjunto finito de reglas que dan una secuencia de operaciones para resolver un tipo específico de problema.

Otra definición que podemos dar de algoritmo es la siguiente: secuencia de pasos, sin ambigüedad, que permite obtener la solución a un problema en un número finito de pasos y en un tiempo finito.

Todo algoritmo debe cumplir las siguientes condiciones:

1.- *Ser finito*, es decir acabar siempre tras un número finito de pasos. Si el algoritmo nunca acaba, no obtendremos ninguna solución y, como se ha señalado, el objetivo principal de un algoritmo es obtener la solución de un problema.
2.- *Ser preciso*. Para ello debe estar compuesto por un *conjunto ordenado* de acciones, especificadas en cada caso rigurosamente y sin ambigüedad.
3.- *Estar definido*. De esta forma se cumplirá que si se sigue el algoritmo dos veces con los mismos datos de entrada, se obtendrán los mismos datos de salida.

Cada uno de los pasos de un algoritmo se denomina ***sentencia*** o ***instrucción***.

¿Cuál es la diferencia entre programa y algoritmo? Para que un ordenador resuelva un problema según un algoritmo éste tiene que ser convertido en un programa traduciéndolo a algún lenguaje de programación concreto. Los algoritmos no son directamente interpretables por el ordenador, pero tienen la ventaja de que son independientes de cualquier lenguaje de programación.

1.1. Ejemplos de algoritmos

El concepto de algoritmo no es particular de la ciencia de la informática, sino que existen algoritmos que describen todo tipo de procesos cotidianos. A continuación veremos algunos ejemplos de algoritmos.

Ejemplo A: Receta o algoritmo para hacer una tortilla de patatas

- Pelar y partir las patatas
- Echar aceite en una sartén.
- Encender el fuego y poner la sartén en el fuego.
- Freír las patatas.
- Batir los huevos.
- Echar los huevos en la sartén y freír todo.

Ejemplo B: Algoritmo que calcula la media aritmética de 3 números usando una calculadora.

- Encender la calculadora.
- Escribir un número.
- Pulsar la tecla '+'.
- Escribir 2º número.
- Pulsar la tecla '+'.

```
- Escribir 3º número.
- Pulsar la tecla '='.
- Pulsar la tecla '/'.
- Escribir el número 3.
- Pulsar la tecla '='.
```

Los dos ejemplos anteriores, aún con una descripción muy similar tienen las siguientes diferencias:

- ✓ En el ejemplo *A*, la repetición del mismo no garantiza el idéntico resultado, en el *B* sí.
- ✓ El resultado del proceso depende del ejecutor en el caso *A* (del cocinero), no así en el caso *B*.
- ✓ La descripción del caso *A* sólo nos serviría para preparar un único plato (tortilla de patatas), mientras que el proceso *B*, descrito de la forma adecuada, permitirá realizar cualquier tipo de media aritmética.

En el campo de la informática a descripciones del tipo del ejemplo *B*, es a lo que denominaremos algoritmo.

1.2. Diseño de algoritmos

La ***algoritmia*** o rama de la ciencia que investiga en los algoritmos, es una ciencia en constante desarrollo y que día a día descubre nuevos algoritmos que mejoran a otros ya existentes o que permiten solucionar nuevos problemas.

Los aspectos más importantes a tener en cuenta en el diseño e nuevos algoritmos son:

1. Análisis de los procesos para los que se desea encontrar un método de resolución único, un algoritmo.
2. Optimización de los algoritmos. Hay que procurar que la utilización de un método no sea más gravosa que el resolver cada problema de forma concreta y única.
3. Determinación de la validez de los algoritmos. Deben servir para resolver cualquier problema del mismo tipo o caso concreto sin excepciones.

El diseño de algoritmos requiere una gran dosis de creatividad, ya que no existe un algoritmo para diseñar algoritmos. Los científicos de la informática pretenden suplir esta falta mediante la creación de unos marcos dentro de los cuales pueda realizarse el proceso de diseño de forma más sencilla, correcta y eficiente. Estos marcos, que usualmente consisten en recomendaciones y consejos basados en la experiencia de los autores, reciben el nombre genérico de "***Metodologías de programación***".

Debido a la complejidad del diseño de algoritmos, es fácil que éstos omitan algún detalle del proceso que describen, no considerado o no previsto por el diseñador. Valga como ejemplo el algoritmo descrito por una persona para indicarle a un amigo cómo encontrar su casa: tuerce por la tercera calle a la derecha, sigue recto hasta el bar, etc. Es normal que por falta de algún detalle el amigo acabe perdiéndose.

Es pues necesario tener mucho cuidado en describir con precisión y sin omitir nada, el proceso para el cual se diseña el algoritmo, eliminando posibles ambigüedades.

Un método que ayuda a esta realización es el conocido como "***Diseño descendente***", "***Top-down***" o de "*Refinamientos sucesivos*", también llamado "*Método de descomposición*" (según Jacopini), "*Diseño Jerárquico*" (según Dijkstra) o "*Por pasos*" ("*Stepwise*" según Niklaus Wirth).

Este método es realmente una variación del famoso método "***divide y vencerás***", y consiste en establecer una serie de niveles, de menor a mayor complejidad, en la descripción del proceso, ordenándolos de forma jerárquica, de manera que la representación de las descripciones forme una especie de árbol genealógico. Cada nivel a su vez estará compuesto por diversos algoritmos más sencillos.

Para comprender mejor el funcionamiento de esta técnica veamos un ejemplo. Supongamos que queremos diseñar un robot para usos domésticos. Uno de los algoritmos a implementar será el de "*preparar una taza de café instantáneo*". Una versión inicial del algoritmo podría ser la siguiente:

```
1) Hervir agua.
2) Poner café en la taza.
3) Agregar agua a la taza.
```

Es posible que los pasos de este algoritmo no sean lo suficientemente claros como para que el robot los interprete (ejecute). Por tanto, cada paso debe refinarse en una secuencia de pasos más simples y detallados, como por ejemplo:

```
1)  Hervir agua.                        2)Poner café en la taza
    1.1) Llenar la cafetera.              2.1) Abrir frasco de café.
    1.2) Encender la cafetera.            2.2) Sacar una cucharada de café.
    1.3) Esperar a que hierva.            2.3) Poner la cucharada en la taza.
    1.4) Apagar la cafetera.              2.4) Tapar el frasco de café.

3) Agregar agua a la taza.
    3.1) Vaciar agua de la cafetera a la taza, hasta llenarla.
```

Obsérvese que el último refinamiento no aumenta el número de pasos, sino que simplemente vuelve a expresar con más detalle el mismo paso.

Lo más probable, es que el robot siga sin entender las diversas instrucciones. Sería necesario realizar entonces otro refinamiento y así sucesivamente hasta llegar a una descripción en la que todos los pasos sean directamente interpretables por el robot.

La versión final de este algoritmo una vez refinado, utilizando primitivas del robot (acciones que el robot es capaz de entender y hacer directamente), podría ser

```
(Hervir agua)
1.1.1. Coger la cafetera del estante.
1.1.2. Colocar la cafetera bajo el grifo de agua.
1.1.3. Abrir el grifo, girando el mando a la izquierda.
1.1.4. Esperar hasta que la cafetera esté llena.
1.1.5. Cerrar el grifo, girando el mando a la derecha.
1.2.1. Conectar enchufe de la cafetera a la red eléctrica.
1.2.2. Pulsar interruptor de encendido de la cafetera.
1.3.1. Esperar hasta que la cafetera silbe.
1.4.1. Pulsar interruptor de apagado de la cafetera.
1.4.2. Desconectar el enchufe de la red eléctrica.

(Poner café en la taza)
2.1.1. Sacar el frasco de café del estante.
2.1.2. Quitar la tapa al frasco, girándola a la izquierda.
2.2.1. Coger una cucharilla del estante.
2.2.2. Meter la cucharilla en el frasco de café.
2.2.3. Recoger una cucharada de café del frasco.
2.3.   Poner la cucharada en la taza.
2.4.1. Tapar frasco de café, girando la tapa a la derecha.
2.4.2. Retornar el frasco de café al estante.

(Agregar agua a la taza)
3.1.   Vaciar agua de la cafetera a la taza, hasta llenarla.
```

Este ejemplo demuestra el hecho de que si el diseñador no conoce la capacidad del procesador que va a ejecutar el algoritmo (las primitivas), el diseño puede avanzar en el vacío y hacerse infinito el refinamiento. Cuando el ejecutor es una persona, se complica el diseño ya que cada persona posee unas capacidades interpretativas diferentes dependientes de muchos factores (edad, nivel cultural, profesión, lugar de origen, etc.). En cambio, estas capacidades están muy bien definidas en un ordenador por lo que la labor de refinamiento resulta más sencilla.

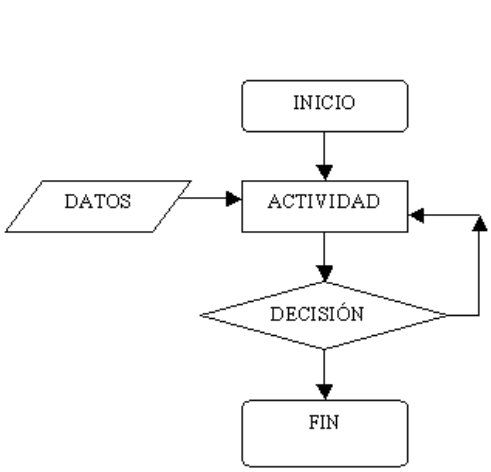
1.3. Métodos de representación de algoritmos

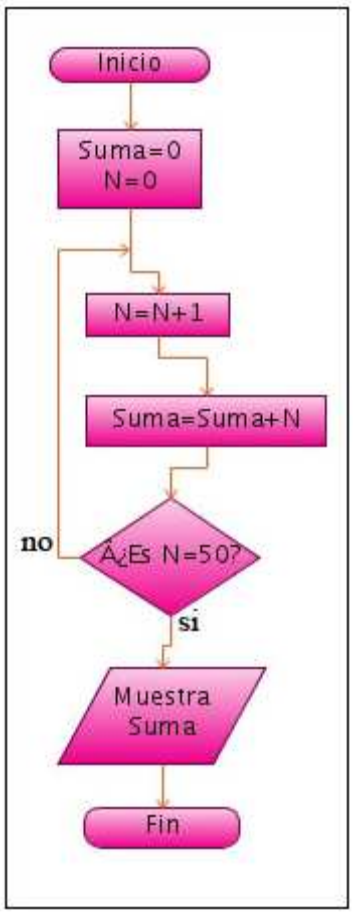
Podemos expresar un algoritmo, básicamente con dos tipos de métodos:

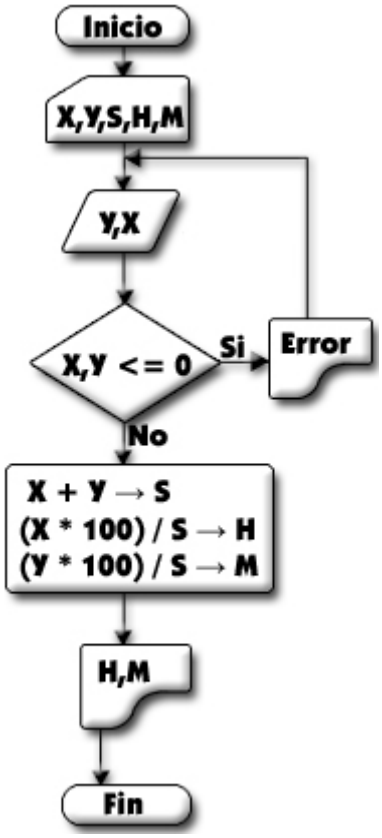
- **Método informal**, como por ejemplo el lenguaje natural. De esta forma, describiremos un algoritmo como si contáramos a otra persona los pasos que ha de seguir para resolver un problema dado. Esta es la forma de descripción que se ha utilizado para explicar los ejemplos anteriores, ya que el lenguaje natural es fácilmente comprensible por todos, y además, es la forma más intuitiva de describir un método para resolver un problema. Sin embargo, las acciones expresadas a través de él pueden no tener un significado preciso y por tanto el algoritmo puede no estar definido, por lo que no es una forma de descripción aconsejable. Es preferible usar cualquiera de los métodos formales que se señalan a continuación.

- **Métodos formales**, entre los que tenemos:
  - Pseudocódigo. Es un lenguaje específico de descripción de algoritmos. Su uso hace que el paso de un algoritmo a un programa sea relativamente fácil. Podríamos decir que se trata de un lenguaje natural limitado y sin ambigüedad. Por su limitación se consigue expresar el conjunto de pasos que resuelven un problema, en función de las estructuras de control básicas. Su no ambigüedad hace que el método sea preciso.

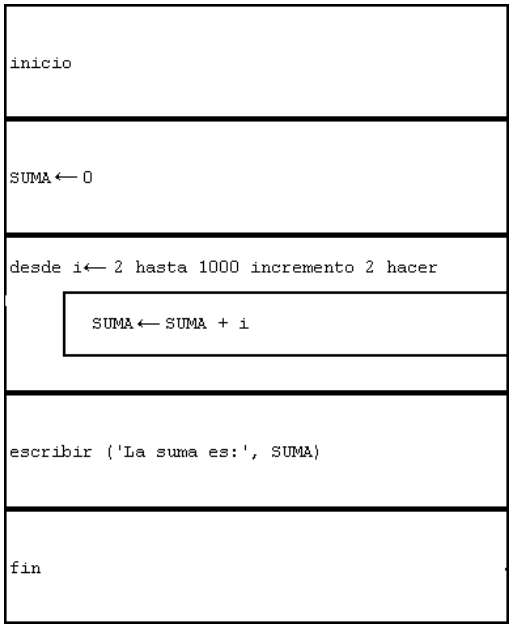
  - Diagramas. Pueden ser de distintos tipos:
    - *Diagramas de flujo u organigramas*. Es un método de representación, que utiliza un conjunto de símbolos de forma que, cada paso del algoritmo se visualiza dentro del símbolo adecuado y el orden en que se realizan los pasos se representa mediante líneas de flujo que indica el flujo lógico del algoritmo. A continuación se muestran algunos ejemplos de diagramas de flujo.

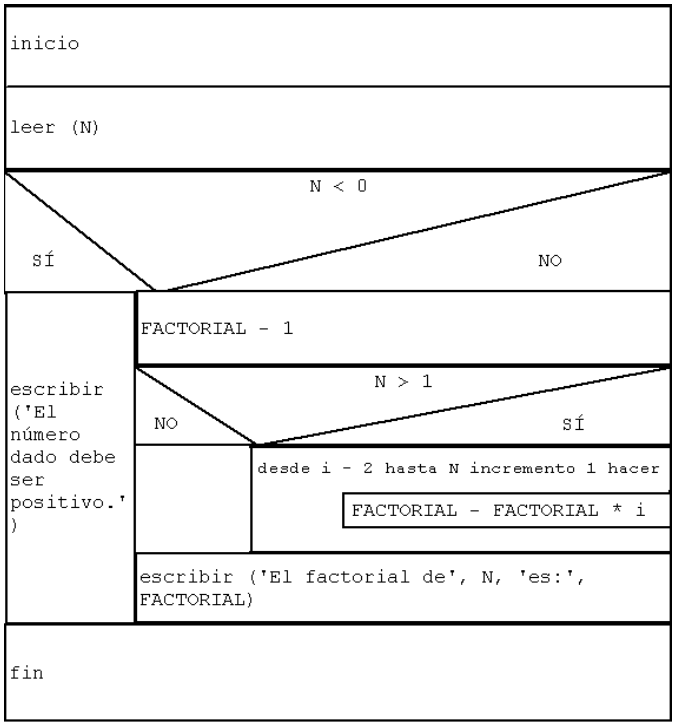






  - *Diagramas de Nassi-Schneiderman* (N-S). Con esta herramienta, los pasos sucesivos se escriben en cajas sucesivas, con distintas formas según la estructura que representen. Ejemplos de este tipo de diagramas son:





2.- ELEMENTOS DE UN ALGORITMO

Antes de comenzar a ver los elementos de un algoritmo, vamos a hacer un pequeño paréntesis para hablar de la memoria del ordenador. Toda la información que maneja un algoritmo ha de almacenarse necesariamente en algún sitio accesible, que en nuestro caso es la **memoria principal.** La memoria de un ordenador se divide en ***casillas*** o ***celdas*** de igual tamaño. Cada casilla es accesible mediante una dirección, a la que se denomina ***dirección de memoria*** o posición de memoria, que identifica unívocamente a cada casilla.

	101	102	103	
	107	108	109	

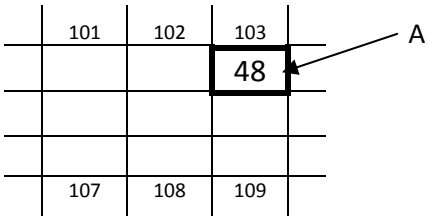
Ejemplo de una porción de memoria con las casillas numeradas de 101 a 109.

Cuando pasamos un algoritmo a programa, tanto este como la información que maneja (datos) se alojan en la memoria. Los lenguajes de programación de alto nivel manejan los datos asociándoles un nombre y no mediante su dirección. La gestión de las localizaciones y asignación de memoria a los datos que manejan los programas, es tarea del compilador.

2.1. Datos y tipos de datos

Un ***dato*** es un conjunto de celdas o posiciones de memoria que tiene asociado un nombre (*identificador*) y un valor (*contenido*).

No todos los datos ocupan el mismo número de celdas en memoria. En los datos hay que distinguir pues, dos partes: nombre y contenido.



En la figura se muestra un ejemplo en el que tenemos un dato al que hemos llamado *A* (es decir, el identificador de este dato es *A*), que almacena el valor 48, y que en este caso sólo ocupa una celda de memoria (la 103).

Los datos que se utilizan en un algoritmo, atendiendo a las propiedades que poseen y las operaciones que se pueden realizar con ellos, se clasifican en distintos tipos, de forma que cada dato pertenece a un tipo de datos concreto. Cada lenguaje de programación tiene su conjunto propio de tipos de datos.

Cuanto en un algoritmo o programa se especifica que un dato es de un determinado tipo, se está dando al ordenador, de manera implícita, la siguiente información:

- El *rango* de los valores permitidos para ese dato.
- El *conjunto de operaciones* (primitivas) que pueden aplicarse a los datos de ese tipo.

Java dispone de ocho tipos **de datos primitivos**:

- **Lógico** (*boolean*). Es aquel que sólo puede tomar uno de los dos valores siguientes ***true*** y ***false.***

- **Carácter** (*char*). Un dato de tipo carácter contiene un único carácter de los que el ordenador reconoce: caracteres alfabéticos (a,...z, A,..., Z), numéricos (0,...,9), especiales (@, #, $, - , %, +, etc.). Se encierran entre comillas simples, por ejemplo, ‘a’, ‘8’, ‘y’, …

- **Numéricos**. Un dato de tipo numérico se puede representar de dos formas distintas: como real y como entero. Los enteros son números positivos o negativos, sin decimales, o dicho de otra forma un subconjunto finito de **Z**. Los reales siempre tienen un punto decimal y pueden ser positivos o negativos, o lo que es lo mismo, un subconjunto finito de **R**. En el caso de Java, tenemos:
  - Cuatro tipos para enteros (***byte***, ***short***, ***int*** y ***long***).
  - Dos para valores reales de punto flotante (***float*** y ***double***).

Los rangos y la memoria que ocupa cada uno de estos tipos se muestran en la siguiente tabla:

Tipo de variable	**Descripción**
boolean	1 byte. Valores true y false
char	2 bytes. Unicode. Comprende el código ASCII
byte	1 byte. Valor entero entre -128 y 127
short	2 bytes. Valor entero entre -32768 y 32767
int	4 bytes. Valor entero entre -2.147.483.648 y 2.147.483.647
long	8 bytes. Valor entre –9.223.372.036.854.775.808 7 9.223.372.036.854.775.807
float	4 bytes (entre 6 y 7 cifras decimales equivalentes). De -3.402823E38 a -1.401298E-45 y de 1.401298E-45 a 3.402823E38
double	8 bytes (unas 15 cifras decimales equivalentes). De -1.79769313486232E308 a -4.94065645841247E-324 y de 4.94065645841247E-324 a 1.79769313486232E308

Los tipos de datos primitivos de ***Java*** tienen algunas características importantes que se resumen a continuación:

1. El tipo ***boolean*** no es un valor numérico: sólo admite los valores ***true*** o ***false***. El tipo ***boolean*** no se identifica con el igual o distinto de cero, como en C/C++. El resultado de la expresión lógica que aparece como condición en un bucle o en una bifurcación debe ser ***boolean***.
2. El tipo ***char*** contiene caracteres en código UNICODE (que incluye el código ASCII), y ocupan 16 bits por carácter. Comprende los caracteres de prácticamente todos los idiomas.
3. Los tipos ***byte***, ***short***, ***int*** y ***long*** son números enteros que pueden ser positivos o negativos, con distintos valores máximos y mínimos.
4. Los tipos ***float*** y ***double*** son valores de punto flotante (números reales) con 6-7 y 15 cifras decimales equivalentes, respectivamente.
5. Se utiliza la palabra ***void*** para indicar la ausencia de un tipo de variable determinado.
6. Existen extensiones a partir de ***Java 1.2*** para aprovechar la arquitectura de los procesadores ***Intel***, que permiten realizar operaciones de punto flotante con una precisión extendida de 80 bits.

No podemos acabar este apartado sin hablar de los que son los ***tipos de datos ordinales***, son aquellos cuyos valores pertenecen a un conjunto finito y ordenado. Cada posible valor a tomar por un dato de un tipo ordinal, tiene un número de orden, dentro del conjunto al que pertenece.

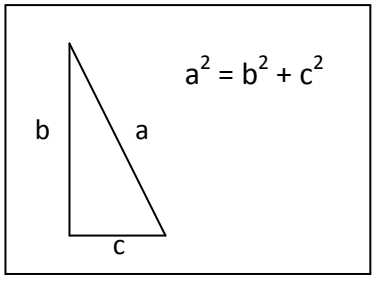
Los tipos de datos entero, lógico y carácter son ordinales, porque cada valor tiene un antecesor y un predecesor, salvo el primero y el último, respectivamente. El tipo real, sin embargo, no es un tipo ordinal.

2.2. Variables y constantes

Una **VARIABLE** es un dato que posee un valor y que es conocido en un programa o en un algoritmo por un nombre (identificador de la variable). Desde otra perspectiva, se puede considerar una variable como una zona de memoria identificada por un nombre.

El valor que contiene una variable puede ser modificado en el seguimiento de un algoritmo (al igual que en la ejecución de un programa).

En Matemáticas, el concepto es muy conocido. Por ejemplo, si se denotan los lados de un triángulo rectángulo mediante *a*, *b* y *c*, el teorema de Pitágoras proporciona la siguiente relación válida para los tres lados:



Ésta es una relación que se expresa de forma general utilizando variables y que puede aplicarse a cálculos específicos. Por ejemplo si *a*=5, y *b*=4, la fórmula determina que *c*=3. En este caso, 5, 4 y 3 son los valores específicos que tienen en un determinado momento las variables *a*, *b* y *c* respectivamente. También en un algoritmo, o en un programa de ordenador el uso de las variables permite la especificación de una fórmula general de cálculo, y al igual que en las fórmulas matemáticas, las variables de los algoritmos o de los programas, tienen nombres y toman diversos valores, pero en un instante concreto sólo tienen un valor determinado.

Existen algunas reglas simples en la denominación de las variables, aunque generalmente dependen del lenguaje de programación que se esté usando. En nuestro caso, para tener una regla general, se establecerá la restricción de que la variable comience con una letra preferiblemente minúscula, y seguidamente vayan letras (excepto la *ñ en la mayoría de los lenguajes, sin embargo, Java, si permite utilizar la ñ*), dígitos numéricos o el carácter especial '_', en cualquier orden, sin embargo el símbolo '_' se está tendiendo últimamente a no utilizarse. No se permitirán espacios en blanco en el nombre de una

variable. También hay que tener en cuenta que los nombres de ***Java*** son sensibles a las letras mayúsculas y minúsculas, así, las variables ***masa***, ***Masa*** y ***MASA*** son consideradas variables completamente diferentes.

Ejemplos:

- Identificadores de variables correctos: hola; total; dia_Del_mes; diaDeMiAniversario, w; r4
- Identificadores incorrectos: 3j; total suma; x+y

En el proceso de diseño y construcción de un programa es preferible utilizar nombres descriptivos, para aumentar la legibilidad de los programas resultantes. En el caso del teorema de Pitágoras, podríamos cambiar el nombre a las variables, pasando a tener:

$$hipotenusa^2 = cateto1^2 + cateto2^2$$

En realidad la línea anterior no sería correcta en nuestro algoritmo, ya que como se ha indicado antes, al estudiar el operador exponenciación, no se expresaría de esta forma, sino como se muestra a continuación,

```
Math.pow(hipotenusa,2) = Math.pow(cateto1, 2) + Math.pow(cateto2, 2)
```

Todavía seguiría sin ser compresible para el compilador esta fórmula, ya que no puedo poner el resultado de una expresión en la parte izquierda de la asignación, por tanto si despejo la hipotenusa, me quedaría finalmente:

```
hipotenusa = Math.sqrt(Math.pow(cateto1, 2) + Math.pow(cateto2, 2))
```

Se ha supuesto que el valor que queramos averiguar es el de la *hipotenusa*, y que las variables c*ateto1* y c*ateto2*, ya tengan almacenados los valores de los lados del triángulo.

Math.pow y **Math.sqrt**, son métodos que proporciona Java, y que nos permiten calcular el cuadrado de un número y la raíz cuadrada respectivamente.

Otros elementos que podemos utilizar en un algoritmo son las **CONSTANTES**, las cuales contienen valores que no deben cambiar a lo largo del desarrollo del algoritmo. A la acción por la que las constantes toman el primer y único valor se denomina *inicialización* de la constante. Las reglas que rigen la forma de crear los identificadores y asignar valores a las constantes son las mismas que para las variables. En Java a las constantes se les conoce como ***variables finales***, como se verá en temas posteriores.

Un concepto relacionado con las constantes son los *valores constantes*: valores que aparecen explícitamente en un algoritmo y que no tienen un identificador asociado, por tanto, no pueden ser referenciadas más que por su propio valor. Así –4.5567 es un valor constante real, "Burgos" un valor constante de tipo cadena de caracteres, y *true* es un valor constante de tipo lógico.

Todas las variables, valores constantes y constantes utilizadas en un algoritmo son de un cierto tipo (entero, real, cadena, lógico, etc.). Una variable entera podrá tomar sólo valores enteros; una variable real podrá tomar sólo valores reales, y una variable lógica tomará también sólo los valores *true y false*. Por eso es conveniente que, al principio del programa, se indique cada variable utilizada y su tipo.

Por último mencionar que existen una serie de ***palabras reservadas,*** las cuales tienen un significado especial para ***Java*** y por lo tanto no se pueden utilizar como nombres de variables o constantes. Dichas palabras son:

abstract	boolean	break	byte	case	catch	char	class
const*	continue	default	do	double	else	extends	final
finally	float	for	goto*	if	implements	import	instanceof
int	interface	long	native	new	null	package	private rotected
public	return	short	static	super	switch	synchronized	
this	throw	throws	transient	try	void	volatile	while.

(*) son palabras reservadas, pero no se utilizan en la actual implementación del lenguaje ***Java***.

2.3. Clasificación de las variables en Java

Como ya hemos visto, una ***variable*** es un ***nombre*** que contiene un valor que puede cambiar a lo largo del programa. De acuerdo con el tipo de información que contienen, en ***Java*** hay dos tipos principales de variables:

1. Variables de ***tipos primitivos***. Están definidas mediante un valor único que puede ser entero, de punto flotante, carácter o booleano. ***Java*** permite distinta precisión y distintos rangos de valores para estos tipos de variables (***char***, ***byte***, ***short***, ***int***, ***long***, ***float***, ***double***, ***boolean***). Ejemplos de valores de tipos primitivos podrían ser: 123, 3456754, 3.1415, 12e-09, 'A', True, etc.
2. Variables ***referencia***. Las variables referencia son referencias o nombres de una información más compleja: ***arrays*** u ***objetos*** de una determinada clase.

Desde el punto de vista de su papel en el programa, las variables pueden ser:

1. Variables ***miembro*** de una clase: Se definen en una clase, fuera de cualquier método; pueden ser *tipos primitivos* o *referencias.*
2. Variables ***locales***: Se definen dentro de un método o más en general *dentro de cualquier bloque* entre llaves {}. Se crean en el interior del bloque y se destruyen al finalizar dicho bloque. Pueden ser también *tipos primitivos* o *referencias*.

2.4. Declaración e inicialización de variables y constantes

Una variable se define especificando el tipo y el nombre de la variable. Estas variables pueden ser tanto de tipos ***primitivos*** como ***referencias*** a objetos de alguna clase perteneciente al ***API*** de ***Java*** o generada por el usuario. Tras la reserva de memoria de una variable, las de tipos ***primitivos*** se inicializan a cero (salvo *boolean* y *char*, que se inicializan a ***false*** y '\0') si no se especifica un valor en su declaración. Análogamente las variables de tipo ***referencia*** son inicializadas por defecto a un valor especial: ***null***.

Hay un tipo de variable que **no se inicializa a ningún valor por defecto, por tanto tendrán un valor desconocido, estas son las variables locales**, pero Java nos obligará a inicializarlas antes de usarlas.

Por tanto la notación para declarar variables en Java es:

<tipo> <nombre-variable>;

donde *<nombre-variable>* es el identificador de la variable y *<tipo>* es el tipo de dicha variable. Como se observa toda sentencia de inicialización (al igual que la mayoría de las sentencias de Java), finaliza con un punto y coma.

Cuando vayamos a declarar varias variables del mismo tipo, éstas las podemos poner en la misma línea de declaración, separadas por comas.

Ejemplos de declaración e inicialización de variables:

```
int x;                  // Declaración de la variable primitiva x. Se inicializa a 0
float suma, total;      // Declaración de dos variables de tipo float, se han
                        // inicializado por defecto a 0.
int y = 5;              // Declaración de la variable primitiva y. Se inicializa a 5
char letra = 'A'        // Declaración de la variable letra, se inicializa al valor 'A'
MyClass unaRef;         // Declaración de una referencia a un objeto MyClass.
                        // Se inicializa por defecto a null
```

En el caso de las constantes se definen de la misma forma pero en la declaración hay que añadir la palabra final. Ya volveremos a hablar de ellas en temas posteriores.

Ejemplo: `final double PI=3.14`

2.5. Operadores de Java

Con un dato de cualquier tipo podemos realizar operaciones primitivas o no. Una **operación primitiva** es aquella que se realiza directamente en un lenguaje de programación concreto, es decir, que no tenemos que indicar como realizarla. Dentro del grupo de operaciones **no primitivas** se incluyen todas aquellas no proporcionadas por el lenguaje de programación que, previamente a su uso, tendremos que indicar cómo realizar a partir de las operaciones primitivas proporcionadas.

Estas operaciones primitivas u ***operadores*** se pueden clasificar, según el número de datos sobre los que operan en:

- Unarios: se aplican sobre un solo dato o argumento. *Por ejemplo: abs(-5), dará 5.*
- Binarios: se aplican sobre dos datos o argumentos. *Por ejemplo: pow(5,2), dará 25, 5+2, dará 7.*
- Enerarios: se aplican sobre N argumentos.

Denominaremos ***operadores*** a las funciones y ***operandos*** a sus argumentos. En el caso de Java muchas de las operaciones para tratar con números se encuentran en una clase llamada Math, que se estudiará más adelante.

Java es un lenguaje rico en operadores, que son casi idénticos a los de C/C++. Estos operadores se describen en los apartados siguientes.

2.5.1. Operadores aritméticos

Son operadores binarios (requieren siempre de dos operandos) que realizan las operaciones aritméticas habituales: ***suma*** (+), ***resta*** (-), ***multiplicación*** (*), ***división*** (/) y ***resto de la división entera*** (%).

Ejemplo: `15 % 2 devolvería 1.`
`7 + 23 devolvería 30.`

Es importante destacar que el resultado de toda operación tiene el mismo tipo que los operandos implicados. Por ejemplo, si se suman dos números reales, se obtiene un número real; si se multiplican dos números enteros, el resultado es un número entero. Si se dividen dos números reales, el resultado también es un número real. Pero si se dividen dos números enteros, el resultado se redondea (o se trunca) para convertirlo a un número entero.

Ejemplo: `1 / 10 * 10 = 0; ya que 1/10 = 0 y 0*10 = 0.`

Afortunadamente, siempre es posible mezclar dos tipos diferentes de datos en las operaciones numéricas. Esto quiere decir que un operando puede ser real y el otro entero. En estos casos el resultado se expresa siempre como un dato del tipo que incluye a los demás (en el ejemplo anterior como un dato real).

Ejemplo: `1.0 / 10 * 10 = 1.0.`

2.5.2. Operadores de asignación.

Los operadores de asignación permiten asignar un valor a una variable. El operador de asignación por excelencia es el ***operador igual*** (=). La forma general de las sentencias de asignación con este operador es:

variable = expresión;

donde <*expresión*> podrá ser una expresión cualquiera, siempre y cuando dicha expresión sea del mismo tipo que la variable a la cual se le va a asignar el resultado.

Ejemplo: A = 3

En este caso, estamos realizando una asignación constante, es decir, estamos diciendo que, a partir de ahora, la variable A tiene el valor constante 3.

En realidad cuando se asigna a una variable una expresión, estamos haciendo dos operaciones:
1. Evaluar la expresión.
2. Almacenar el resultado de la evaluación en la variable.

Ejemplo: A=3*B+5

La operación de asignación es destructiva, por lo que cualquier valor que tuviese la variable de asignación se pierde y se reemplaza por el nuevo. Veamos como quedan las variables A y B tras la siguiente secuencia de asignaciones:

A=1	A	1	B	*desconocido*
B=5	A	1	B	5
A=2*B	A	10	B	5
A=A+1	A	11	B	5

Finalmente A almacena el valor 11, y B el valor 5.

La asignación implica siempre una transferencia o movimiento de datos en memoria.

Java dispone de otros operadores de asignación. Se trata de versiones abreviadas del operador (=) que realizan operaciones "acumulativas" sobre una variable. La siguiente tabla muestra estos operadores y su equivalencia con el uso del ***operador igual*** (=).

Operador	*Utilización*	*Expresión equivalente*
+=	`op1 += op2`	`op1 = op1 + op2`
-=	`op1 -= op2`	`op1 = op1 - op2`
*=	`op1 *= op2`	`op1 = op1 * op2`
/=	`op1 /= op2`	`op1 = op1 / op2`
%=	`op1 %= op2`	`op1 = op1 % op2`

2.5.3. Operadores unarios

Los operadores ***más*** (+) y ***menos*** (-) unarios sirven para mantener o cambiar el signo de una variable, constante o expresión numérica. Su uso en ***Java*** es el estándar de estos operadores.

2.5.4. Operador instanceof

El operador ***instanceof*** permite saber si un objeto pertenece a una determinada clase o no. Es un operador binario cuya forma general es,

```
objectName instanceof ClassName
```

y que devuelve ***true*** o ***false*** según el objeto pertenezca o no a la clase.

2.5.5. Operador condicional ?

Este operador, tomado de C/C++, permite realizar bifurcaciones condicionales sencillas. Su forma general es la siguiente:

```
booleanExpression ? res1 : res2
```

donde se evalúa ***booleanExpression*** y se devuelve ***res1*** si el resultado es ***true*** y ***res2*** si el resultado es ***false***. Es el único operador ternario (tres argumentos) de ***Java***. Como todo operador que devuelve un valor puede ser utilizado en una expresión. Por ejemplo las sentencias:

```
x=1; y=10; z = (x<y)?x+3:y+8;
```

asignarían a ***z*** el valor 4, es decir x+3.

2.5.6. Operadores incrementales

Java dispone del operador ***incremento*** (++) y ***decremento*** (--). El operador (++) incrementa en una unidad la variable a la que se aplica, mientras que (--) la reduce en una unidad. Estos operadores se pueden utilizar de dos formas:

1. *Precediendo a la variable* (por ejemplo: ***++i***). En este caso primero se incrementa la variable y luego se utiliza (ya incrementada) en la expresión en la que aparece.
2. *Siguiendo a la variable* (por ejemplo: ***i++***). En este caso primero se utiliza la variable en la expresión (con el valor anterior) y luego se incrementa.

Ejemplo:
```
a=5
b=++a
```
Los valores de las variables quedan como a = 6, b = 6

Cuando el operador está como sufijo, primero se hace la operación donde está incluido el operando y a continuación se hace el incremento.

Ejemplo:
```
a=5
b=a++
```
Los valores de las variables quedan como a = 6, b = 5

Veamos algunos ejemplos más:

Ejemplo: `i=++j`
Esto equivaldría a: `j=j+1; i=j;`

Ejemplo: `i=++j + 5`
Esto equivaldría a: `j=j+1; i=j+5;`

Ejemplo: `i=j++`
Esto equivaldría a: `i=j; j=j+1;`

Ejemplo: `i=j++ + 5`
Esto equivaldría a: `i=j+5; j=j+1;`

En muchas ocasiones estos operadores se utilizan para incrementar una variable fuera de una expresión. En este caso ambos operadores son equivalentes. Si se utilizan en una expresión más complicada, el resultado de utilizar estos operadores en una u otra de sus formas será diferente. La actualización de contadores en bucles ***for*** es una de las aplicaciones más frecuentes de estos operadores.

El operador decremento se utiliza para disminuir el valor de un operando en 1. La operación `--c` equivale a `c=c-1`. Puede ir como prefijo o como sufijo, actuando de forma similar al operador ++.

2.5.7. Operadores relacionales

Los ***operadores relacionales*** sirven para realizar comparaciones de igualdad, desigualdad y relación de menor o mayor. El resultado de estos operadores es siempre un valor ***boolean*** (***true*** o ***false***) según se cumpla o no la relación considerada. La siguiente tabla muestra los operadores relacionales de ***Java***.

Operador	*Utilización*	*El resultado es true*
`>`	`op1 > op2`	`si op1 es mayor que op2`
`>=`	`op1 >= op2`	`si op1 es mayor o igual que op2`
`<`	`op1 < op2`	`si op1 es menor que op2`
`<=`	`op1 <= op2`	`si op1 es menor o igual que op2`
`==`	`op1 == op2`	`si op1 y op2 son iguales`
`!=`	`op1 != op2`	`si op1 y op2 son diferentes`

Estos operadores se utilizan con mucha frecuencia en las ***bifurcaciones*** y en los ***bucles***, que se verán en próximos temas.

La razón por la cual se pueden comparar caracteres es la existencia de juegos de caracteres (como por ejemplo, los códigos ASCII, EBCDIC, UNICODE, …), que los representan en memoria como un número binario. Es por esto, que los caracteres se pueden ver como números y, por tanto, se establece una relación de orden entre ellos. Así, el carácter 'A' tiene el valor 65 en el código ASCII y la 'H', el 72, por lo que 'A' < 'H'. En general se verifica que:

'A' < 'B' < 'C' < ... < 'Y' < 'Z' < 'a' < 'b' < ... < 'y' < 'z'

'0' < '1' < ... < '8' < '9'

y que el número asociado con cualquier letra es menor que el asociado con cualquier número.

2.5.8. Operadores lógicos.

Las operaciones primitivas para datos de **tipo lógico o booleano** vienen determinadas por la siguiente *tabla de verdad*:

Operador A	Operador B	A Y B	A O B	NO A
Verdadero	Verdadero	Verdadero	Verdadero	Falso
Verdadero	Falso	Falso	Verdadero	Falso
Falso	Verdadero	Falso	Verdadero	Verdadero
Falso	Falso	Falso	Falso	Verdadero

Operadores lógicos básicos

En esta tabla aparecen los tres ***operadores lógicos*** básicos que se pueden aplicar a un dato de tipo lógico y son:

- Conjunción (Y o AND). Devolverá verdadero siempre que los dos operandos sean verdaderos y falso en cualquier otro caso.
- Disyunción (O u OR). Devolverá verdadero siempre que alguno de los dos operandos sea verdadero, y falso solo en el caso de que los dos operandos sean falsos.
- Negación (NO o NOT o !). Negará el valor que tenga el operando.

Los operadores lógicos se utilizan para construir ***expresiones lógicas***, combinando valores lógicos (***true*** y/o ***false***) o los resultados de los operadores ***relacionales***. La siguiente tabla muestra los operadores lógicos de ***Java***. Debe notarse que en ciertos casos el segundo operando no se evalúa porque no es necesario (si ambos tienen que ser ***true*** y el primero es ***false*** ya se sabe que la condición de que ambos sean ***true*** no se

va a cumplir). Esto puede traer resultados no deseados y por eso se han añadido los operadores (&) y (|) que garantizan que los dos operandos se evalúan siempre.

Operador	*Nombre*	Utilización	*Resultado*
&&	AND	op1 && op2	true si op1 y op2 son true. Si op1 es false ya no se evalúa op2
||	OR	op1 || op2	true si op1 u op2 son true. Si op1 es true ya no se evalúa op2
!	NOT	! op	true si op es false y false si es true
&	AND	op1 & op2	true si op1 y op2 son true. Siempre se evalua op2
|	OR	op1 | op2	true si op1 u op2 son true. Siempre se evalúa op2

2.5.9. Operador de concatenación de cadenas de caracteres (+).

El operador más (+) se utiliza también para concatenar cadenas de caracteres. Por ejemplo, para escribir una cantidad con un rótulo y unas unidades puede utilizarse la sentencia:

```
System.out.println("El total asciende a " + result + " unidades");
```

donde el operador de concatenación se utiliza dos veces para construir la cadena de caracteres que se desea imprimir por medio del método ***println()***. La variable numérica ***result*** es convertida automáticamente por ***Java*** en cadena de caracteres para poderla concatenar. En otras ocasiones se deberá llamar explícitamente a un método para que realice esta conversión.

2.5.10. Operadores que actúan a nivel de bits.

Java dispone también de un conjunto de operadores que actúan a nivel de bits. Las operaciones de bits se utilizan con frecuencia para definir señales o ***flags***, esto es, variables de tipo entero en las que cada uno de sus bits indican si una opción está activada o no. La Tabla 2.5 muestra los operadores de ***Java*** que actúan a nivel de bits.

Operador	*Utilización*	*Resultado*
>>	op1 >> op2	Desplaza los bits de op1 a la derecha una distancia op2
<<	op1 << op2	Desplaza los bits de op1 a la izquierda una distancia op2
>>>	op1 >>> op2	Desplaza los bits de op1 a la derecha una distancia op2(positiva)
&	op1 & op2	Operador AND a nivel de bits
|	op1 | op2	Operador OR a nivel de bits
^	op1 ^ op2	Operador XOR a nivel de bits
~	op1 ~ op2	Operador complemento

En binario, las potencias de dos se representan con un único bit activado. Por ejemplo, los números (1, 2, 4, 8, 16, 32, 64, 128) se representan respectivamente de modo binario en la forma (00000001, 00000010, 00000100, 00001000, 00010000, 00100000, 01000000, 10000000), utilizando sólo 8 bits. La suma de estos números permite construir una variable ***flags*** con los bits activados que se deseen. Por ejemplo, para construir una variable ***flags*** que sea 00010010 bastaría hacer ***flags***=2+16. Para saber si el segundo bit por la derecha está o no activado bastaría utilizar la sentencia,

if (flags & 2 == 2) {...}

La siguiente tabla muestra los operadores de asignación a nivel de bits.

Operador	*Utilización*	*Equivalente a*
&=	op1 &= op2	op1 = op1 & op2
|=	op1 |= op2	op1 = op1 | op2
^=	op1 ^= op2	op1 = op1 ^ op2
<<=	op1 <<= op2	op1 = op1 << op2
>>=	op1 >>= op2	op1 = op1 >> op2
>>>=	op1 >>>= op2	op1 = op1 >>> op2

2.5.11. Precedencia de operadores

El orden en que se realizan las operaciones es fundamental para determinar el resultado de una expresión. Por ejemplo, el resultado de *x/y*z* depende de qué operación (la división o el producto) se realice primero. La siguiente lista muestra el orden en que se ejecutan los distintos operadores de Java en una sentencia, de ***mayor a menor*** precedencia:

operadores postfijos	[] . (params) expr++ expr--
operadores unarios	++expr --expr +expr -expr ~ !
creacion o cast	new (type)expr
multiplicación / división	* / %
suma / resta	+ -
shift	<< >> >>>
relacionales	< > <= >= instanceof
igualdad	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	|
lógico AND	&&
lógico OR	||
condicional	? :
asignación	= += -= *= /= %= &= ^= |= <<= >>= >>>=

En ***Java***, todos los operadores binarios, excepto los operadores de asignación, se evalúan de ***izquierda a derecha***. Los operadores de asignación se evalúan de derecha a izquierda, lo que significa que el valor de la izquierda se copia sobre la variable de la derecha.

2.6. Conversiones de tipo (Casting)

En muchas ocasiones hay que transformar una variable de un tipo a otro, por ejemplo de *int* a *double*, o de *float* a *long*. En otras ocasiones la conversión debe hacerse entre objetos de clases diferentes, aunque relacionadas mediante la herencia. En este apartado se explican brevemente estas transformaciones de tipo.

La conversión entre tipos primitivos es más sencilla. En ***Java*** se realizan de modo automático conversiones implícitas ***de un tipo a otro de más precisión***, por ejemplo de ***int*** a ***long***, de ***float*** a ***double***, etc. Estas conversiones se hacen al mezclar variables de distintos tipos en expresiones matemáticas o al ejecutar sentencias de asignación en las que el miembro izquierdo tiene un tipo distinto que el resultado de evaluar el miembro derecho.

Las conversiones de un tipo de mayor a otro de menor precisión requieren una orden explícita del programador, pues son ***conversiones inseguras*** que pueden dar lugar a errores (por ejemplo, para pasar a ***short*** un número almacenado como ***int***, hay que estar seguro de que puede ser representado con el número de cifras binarias de ***short***). A estas conversiones explícitas de tipo se les llama ***cast***. El ***cast*** se hace poniendo el tipo al que se desea transformar entre paréntesis, como por ejemplo,

```
long result;
result = (long) (a/(b+c));
```

A diferencia de C/C++, en ***Java*** no se puede convertir un tipo numérico a ***boolean***.

Existe un tipo de casting muy común de utilizar entre el tipo carácter y el tipo entero, se pueden hacer las siguientes conversiones:

- De entero a carácter, devuelve el carácter correspondiente al código ASCII usado como valor numérico.
  Ejemplo: `System.out.println((char)65);  // Imprimirá el carácter 'A'`

- De carácter a entero, devuelve el código ASCII del carácter usado.
  Ejemplo: `System.out.println((int)'A');  // Imprimirá el valor 65`

3. EXPRESIONES

Una **EXPRESIÓN** es una combinación de variables, constantes, valores constantes, operadores, paréntesis y nombres de funciones especiales (raíz cuadrada, valor absoluto, etc.). Toda expresión tiene en todo momento un valor concreto que es el resultado de evaluarla de izquierda a derecha. Es decir, es el resultado de tomar el valor de las variables que intervienen en ella y realizar las operaciones que aparecen.

Cuando se desee formar la evaluación de una expresión en un determinado orden, independientemente de la precedencia de los operadores, se utilizarán los paréntesis. Así en una expresión que contiene paréntesis, el orden a seguir para evaluarla es:

1. Las operaciones encerradas entre paréntesis se evalúan primero. Si existen diferentes paréntesis anidados, las expresiones internas se evalúan antes.
2. Las operaciones aritméticas dentro de una expresión se evalúan según el orden de prioridad expresado en el apartado 2.4.11. Si coinciden varios operadores de igual prioridad, el orden a seguir es de izquierda a derecha.

Ejemplos: Hallar el valor de cada una de las siguientes expresiones e indicar el tipo de las mismas.

❑ `2*3+5/3 ⇒ 6+5/3 ⇒ 6+1 ⇒ 7`
`Tipo: Entero`

❑ `(4.0+5)/(2+6)*4 ⇒ 9.0/(2+6)*4 ⇒ 9.0/8*4 ⇒ 1.125*4 ⇒ 4.5`
`Tipo: Real`

❑ `A^3*4+B^((5+C)*2) con A=2; B=3; C=1`
⇓
`A^3*4+B^(6*2) ⇒ A^3*4+B^12 ⇒ 8*4+B^12 ⇒ 32+B^12 ⇒ 32+531441 ⇒ 531473`
`Tipo: Real`

A las expresiones de este tipo se les conoce como **expresiones aritméticas**.

Mención aparte merecen por su importancia, las **expresiones lógicas** que son las que ofrecen como resultado de su evaluación un valor verdadero o falso.

Ejemplo: Son expresiones lógicas:

`a O b, a O (NO b Y c), NO (a O b),`

su equivalente en Java sería `a || b, a||(!b && c), !(a || b)`

donde a, b, c son variables o expresiones booleanas. Dependiendo de los valores de estas variables, las expresiones de los ejemplos se podrán evaluar como verdaderas o falsas.

Se establece también un orden de precedencia de los operadores lógicos: El de mayor prioridad es NO y el de menor prioridad es O, y el operador Y se encuentra entre los dos.

Ejemplos: Determinar el valor de las siguientes expresiones:

❑ `(3<6) Y (5<4) ⇒ Verdadero Y Falso ⇒ Falso`

❑ `(A==B) O (NO C) donde A=3.5; B=3.5; C=Falso`
⇓
`Verdadero O (NO C) ⇒ Verdadero O Verdadero ⇒ Verdadero`

4. ELEMENTOS AUXILIARES DE UN ALGORITMO

Son variables que realizan funciones específicas dentro de un programa, y por su gran utilidad, frecuencia de uso y peculiaridades, conviene hacer un estudio separado de las mismas. Las más importantes son los contadores, acumuladores e interruptores.

4.1. Contadores

Un contador es una zona de memoria representada por un identificador (en definitiva una variable) cuyo valor se incrementa en una cantidad fija, positiva o negativa, generalmente asociado a un *bucle* (se estudiara en el siguiente tema).

CONTADOR = CONTADOR ± INCREMENTO

Se utiliza para contabilizar el número de veces que es necesario repetir una acción, o bien, para contar un suceso particular solicitado por el enunciado del problema.

Un contador toma un valor inicial (0 en la mayoría de los casos) antes de comenzar su función, posteriormente, y cada vez que se realiza el suceso a contar incrementa su valor (1 en la mayoría de los casos).

4.2. Acumuladores

Un acumulador es una zona de memoria cuyo valor se incrementa sucesivas veces en cantidades variables.

ACUMULADOR = ACUMULADOR (OP-ARITM.) EXPRESIÓN

Se utiliza en aquellos casos en que se desea obtener el total acumulado de un conjunto de cantidades, siendo preciso inicializarlo con el valor 0. También en las situaciones en que hay que obtener un total como producto de distintas cantidades se utiliza un acumulador, debiéndose inicializar con el valor 1.

4.3. Interruptores (switches)

Un interruptor es una zona de memoria que puede tomar dos valores exclusivos (0 y 1, -1 y 1, verdadero y falso, ON y OFF, etc.) Se utiliza para la toma de decisiones dentro de un algoritmo.

5.- COMENTARIOS

Existen dos formas diferentes de introducir comentarios entre el código de ***Java*** (en realidad son tres, como pronto se verá). Son similares a la forma de realizar comentarios en el lenguaje C++. Los comentarios son tremendamente útiles para poder entender el código utilizado, facilitando de ese modo futuras revisiones y correcciones. Además permite que cualquier persona distinta al programador original pueda comprender el código escrito de una forma más rápida. Se recomienda acostumbrarse a comentar el código desarrollado. De esta forma se simplifica también la tarea de estudio y revisión posteriores.

Java interpreta que todo lo que aparece a la derecha de dos barras "//" en una línea cualquiera del código es un comentario del programador y no lo tiene en cuenta. El comentario puede empezar al comienzo de la línea o a continuación de una instrucción que debe ser ejecutada. La segunda forma de incluir comentarios consiste en escribir el texto entre los símbolos **/*…*/**. Este segundo método es válido para comentar más de una línea de código. Por ejemplo:

```
// Esta línea es un comentario
int a=1; // Comentario a la derecha de una sentencia
// Esta es la forma de comentar más de una línea utilizando
// las dos barras. Requiere incluir dos barras al comienzo de cada línea
/* Esta segunda forma es mucho más cómoda para comentar un número elevado de líneas
   ya que sólo requiere modificar
   el comienzo y el final. */
```

En ***Java*** existe además una forma especial de introducir los comentarios (utilizando /**…*/ más algunos caracteres especiales) que permite generar automáticamente la documentación sobre las ***clases*** y ***packages*** desarrollados por el programador. Una vez introducidos los comentarios, el programa ***javadoc.exe*** (incluido en el ***JDK –*** *compilador de java*) genera de forma automática la información de forma similar a la presentada en la propia documentación del ***JDK***. La sintaxis de estos comentarios y la forma de utilizar el programa ***javadoc.exe*** se puede encontrar en la información que viene con el ***JDK***.

ACTIVIDADES

1.- Qué diferencia existe entre algoritmo y programa.

2.- Pon un ejemplo de algoritmo no finito.

3.- ¿Qué significa que un algoritmo sea ambiguo?

4.- Diseña una solución para resolver cada uno de los siguientes problemas:

a) Realizar una llamada telefónica desde un teléfono público.
c) Montar un ordenador si nos han dado los siguientes componentes: monitor, teclado, ratón, torre, dos cables de alimentación, un cable de red y un cable de monitor.
d) Restar dos números enteros.
e) Dividir dos números, comprobando que los dos sean positivos y que el numerador sea mayor que el denominador, con la ayuda de una calculadora.

5.- Escribir la expresión algorítmica correcta para las siguientes expresiones:

a) $5^2 + [4 - (2 * 35)]$ b) $\frac{A^2}{B-C} + \frac{D-E}{F - \frac{G*H}{J}}$

6.- Escribir el resultado de las siguientes expresiones:

a) $(6^2 + (8 - 2)) / 7 + (35 / 2) - 8 * 5 / 4 * 2$
b) (27 % 4) + (15 / 4)
c) (37 / 4) ^2 - 2
d) ((9 * 2) / (3 * 2)) + 5 * 3
e) (7 * 3 – 4 * 4) ^ 2 / 4 * 2

7.- Calcula el resultado de las siguientes expresiones lógicas y transfórmalas a sintaxis de Java.

a) 4 > 5 O NO (45 == 7) Y 7 + 3 < 5 - 2
b) 25 >=7 Y NO (7 =< 2)
c) 24 > 5 Y 10 <= 10 O 10 == 5
d) (10 >= 15 O 23 == 13) Y NO (8 == 8)
e) (NO (6 /3 > 3) O 7 > 7) Y (3 <= div(9, 2) O 2 + 3 <= 7 / 2)

8.- Escribir la expresión algorítmica correcta, sabiendo que una variable *cantidad* es mayor o igual a 15 o menor que 9.

9.- Evalúa las siguientes expresiones lógicas:

a) A <> B cuando A = 11 y B = 11
b) A >= B cuando A = 23 y B = 23
c) (A-2) * 3 <= B + 5 cuando A = 3 y B = -1
d) A > B cuando A = 0 y B = 0
e) (1 > 0) y (3 == 3)
f) NO PRUEBA, cuando PRUEBA es verdadero
g) NO (5 <> 5)