TEMA 1: Introducción a la programación

Sumario

Introducción

El propósito general de un lenguaje de programación es permitir a un ser humano (el programador) traducir la idea de un programa en una secuencia de instrucciones que el ordenador sea capaz de ejecutar.

Como los lenguajes humanos, los lenguajes de programación son herramientas de comunicación, pero al contrario que los lenguajes corrientes como el inglés o el chino, los destinatarios de los lenguajes de programación no son sólo humanos sino también los ordenadores

Si nos queremos entender con el ordenador, el ordenador solo entiende el llamado código máquina (1 y 0, 1=hay o 0=no hay corriente).

Es por eso que tenemos 2 tipos diferentes de lenguajes de programación, dependiendo de su cercanía al lenguaje del ordenador. Los de bajo y los de alto nivel. Estos lenguajes sirven para dar órdenes directas al hardware del ordenador.Los lenguajes más cercanos al idioma del ordenador, llamados de bajo nivel, son muy complicados (casi como el código máquina) y poco usados.

El más conocido es el código o lenguaje máquina, un código que el ordenador puede interpretar directamente. Aquí tienes un ejemplo: 8B542408 83FA0077 06B80000 0000C383

Este es el lenguaje utilizado directamente por el procesador, consta de un conjunto de instrucciones codificadas en binario. Es el sistema de códigos directamente interpretable por la máquina.

Este fue el primer lenguaje utilizado para la programación de computadores. De hecho, cada máquina tenía su propio conjunto de instrucciones codificadas en ceros y unos. Cuando un algoritmo está escrito en este tipo de lenguaje, decimos que está en código máquina.

Programar en este tipo de lenguaje presentaba los siguientes inconvenientes:

- Cada programa era válido sólo para un tipo de procesador u ordenador.
- La lectura o interpretación de los programas era extremadamente difícil y, por tanto, insertar modificaciones resultaba muy costoso.
- Los programadores de la época debían memorizar largas combinaciones de ceros y unos, que equivalían a las instrucciones disponibles para los diferentes tipos de procesadores.
- Los programadores se encargaban de introducir los códigos binarios en el computador, lo que provocaba largos tiempos de preparación y posibles errores.

A continuación, se muestran algunos códigos binarios equivalentes a las operaciones de suma, resta y movimiento de datos en lenguaje máquina.

Algunas operaciones en lenguaje máquina.

Operación	Lenguaje máquina
SUMAR	00101101
RESTAR	00010011
MOVER	00111010

Ensamblador:

La evolución del lenguaje máquina fue el lenguaje ensamblador. Las instrucciones ya no son secuencias binarias, se sustituyen por códigos de operación que describen una operación elemental del procesador. Es un lenguaje de bajo nivel, al igual que el lenguaje máquina, ya que dependen directamente del hardware donde son ejecutados.

Mnemotécnico: son palabras especiales, que sustituyen largas secuencias de ceros y unos, utilizadas para referirse a diferentes operaciones disponibles en el juego de instrucciones que soporta cada máquina en particular.

En ensamblador, cada instrucción (mnemotécnico) se corresponde a una instrucción del procesador. En la siguiente tabla se muestran algunos ejemplos.

Algunas operaciones y su mnemotécnico en lenguaje Ensamblador.

Operación	Lenguaje Ensamblador
MULTIPLICAR	MUL
DIVIDIR	DIV
MOVER	MOV

El código de un programa suele escribirse en uno o varios ficheros de texto. Este código que es interpretable por un humano (o al menos debiera serlo) no puede ser ejecutado directamente por el computador. El problema de los lenguajes de alto nivel es que necesitan un compilador o interprete para traducirlo al código máquina.Existen unos lenguajes llamados lenguajes interpretados, en los que existe un programa llamado intérprete que lee las líneas de código y las ejecuta inmediatamente, es decir, se analizan y ejecutan las instrucciones por el propio programa directamente, pero siguen siendo lenguajes de alto nivel. Ejemplos de este tipo es Java

Uso general de un programa informático

Un programa es como una receta de cocina: es una secuencia de pasos que se deben realizar. Ejemplo:

```
Programa freirHuevo
        reservar huevo, pan, sal, mantequilla;
        usar cocina;
        colocar sarten en cocina;
        poner la mantequilla en la sarten;
        encender cocina;
        esperar a que la mantequilla se caliente;
        romper el huevo;
        derramar el huevo en la sarten;
        poner sal en el huevo;
        esperar a que el huevo se fria;
        apagar cocina;
        servir el huevo;
Fin programa
```

Observar que el texto se ha introducido desplazando con tabulador todas las lineas que corresponden que estén dentro de una estructura lógica (en este caso el programa freirHuevo) Veremos más adelante la importancia de identar el texto de esta formatos

Lenguajes de programación.

Lenguaje de programación: Conjunto de reglas sintácticas y semánticas, símbolos y palabras especiales establecidas para la construcción de programas. Es un lenguaje artificial, una construcción mental del ser humano para expresar programas.

Gramática del lenguaje: Reglas aplicables al conjunto de símbolos y palabras especiales del lenguaje de programación para la construcción de sentencias correctas.

Léxico: Es el conjunto finito de símbolos y palabras especiales, es el vocabulario del lenguaje.

Sintaxis: Son las posibles combinaciones de los símbolos y palabras especiales. Está relacionada con la forma de los programas.

Semántica: Es el significado de cada construcción del lenguaje, la acción que se llevará a cabo.

Hay que tener en cuenta que pueden existir sentencias sintácticamente correctas, pero semánticamente incorrectas. Por ejemplo, “Un avestruz dio un zarpazo a su cuidador” está bien construida sintácticamente, pero es evidente que semánticamente no.

Una característica relevante de los lenguajes de programación es, precisamente, que más de un programador pueda usar un conjunto común de instrucciones que sean comprendidas entre ellos. A través de este conjunto se puede lograr la construcción de un programa de forma colaborativa.

Los lenguajes de programación pueden ser clasificados en función de lo cerca que estén del lenguaje humano o del lenguaje de los computadores. El lenguaje de los computadores son códigos binarios, es decir, secuencias de unos y ceros. Detallaremos seguidamente las características principales de los lenguajes de programación.

Lenguaje máquina.

Este es el lenguaje utilizado directamente por el procesador, consta de un conjunto de instrucciones codificadas en binario. Es el sistema de códigos directamente interpretable por un circuito microprogramable.

Este fue el primer lenguaje utilizado para la programación de computadores. De hecho, cada máquina tenía su propio conjunto de instrucciones codificadas en ceros y unos. Cuando un algoritmo está escrito en este tipo de lenguaje, decimos que está en código máquina.

Programar en este tipo de lenguaje presentaba los siguientes inconvenientes:

Cada programa era válido sólo para un tipo de procesador u ordenador.
La lectura o interpretación de los programas era extremadamente difícil y, por tanto, insertar modificaciones resultaba muy costoso.
Los programadores de la época debían memorizar largas combinaciones de ceros y unos, que equivalían a las instrucciones disponibles para los diferentes tipos de procesadores.
Los programadores se encargaban de introducir los códigos binarios en el computador, lo que provocaba largos tiempos de preparación y posibles errores.

Lenguaje Ensamblador.

La evolución del lenguaje máquina fue el lenguaje ensamblador. Las instrucciones ya no son secuencias binarias, se sustituyen por códigos de operación que describen una operación elemental del procesador. Es un lenguaje de bajo nivel, al igual que el lenguaje máquina, ya que dependen directamente del hardware donde son ejecutados.

Pero aunque ensamblador fue un intento por aproximar el lenguaje de los procesadores al lenguaje humano, presentaba múltiples dificultades:

Los programas seguían dependiendo directamente del hardware que los soportaba.
Los programadores tenían que conocer detalladamente la máquina sobre la que programaban, ya que debían hacer un uso adecuado de los recursos de dichos sistemas.
La lectura, interpretación o modificación de los programas seguía presentando dificultades.

Todo programa escrito en lenguaje ensamblador necesita de un intermediario, que realice la traducción de cada una de las instrucciones que componen su código al lenguaje máquina correspondiente. Este intermediario es el programa ensamblador. El programa original escrito en lenguaje ensamblador constituye el código fuente y el programa traducido al lenguaje máquina se conoce como programa objeto que será directamente ejecutado por la computadora.

Lenguajes compilados.

Para paliar los problemas derivados del uso del lenguaje ensamblador y con el objetivo de acercar la programación hacia el uso de un lenguaje más cercano al humano que al del computador, nacieron los lenguajes compilados. Algunos ejemplos de este tipo de lenguajes son: Pascal, Fortran, Algol, C, C++, etc.

Al ser lenguajes más cercanos al humano, también se les denomina lenguajes de alto nivel. Son más fáciles de utilizar y comprender, las instrucciones que forman parte de estos lenguajes utilizan palabras y signos reconocibles por el programador.

¿Cuáles son sus ventajas?

Son mucho más fáciles de aprender y de utilizar que sus predecesores.
Se reduce el tiempo para desarrollar programas, así como los costes.
Son independientes del hardware, los programas pueden ejecutarse en diferentes tipos de máquina.
La lectura, interpretación y modificación de los programas es mucho más sencilla.

Pero un programa que está escrito en un lenguaje de alto nivel también tiene que traducirse a un código que pueda utilizar la máquina. Los programas traductores que pueden realizar esta operación se llaman compiladores.

Compilador: Es un programa cuya función consiste en traducir el código fuente de un programa escrito en un lenguaje de alto nivel a lenguaje máquina. Al proceso de traducción se le conoce con el nombre de compilación.

El compilador realizará la traducción y además informará de los posibles errores. Una vez subsanados, se generará el programa traducido a código máquina, conocido como código objeto. Este programa aún no podrá ser ejecutado hasta que no se le añadan los módulos de enlace o bibliotecas, durante el proceso de enlazado. Una vez finalizado el enlazado, se obtiene el código ejecutable.

Lenguajes interpretados.

Se caracterizan por estar diseñados para que su ejecución se realice a través de un intérprete. Cada instrucción escrita en un lenguaje interpretado se analiza, traduce y ejecuta tras haber sido verificada. Una vez realizado el proceso por el intérprete, la instrucción se ejecuta, pero no se guarda en memoria.

Intérprete: Es un programa traductor de un lenguaje de alto nivel en el que el proceso de traducción y de ejecución se llevan a cabo simultáneamente, es decir, la instrucción se pasa a lenguaje máquina y se ejecuta directamente. No se genera programa objeto, ni programa ejecutable. Primer plano de las manos de dos pianistas interpretando una partitura.

Los lenguajes interpretados generan programas de menor tamaño que los generados por un compilador, al no guardar el programa traducido a código máquina. Pero presentan el inconveniente de ser algo más lentos, ya que han de ser traducidos durante su ejecución. Por otra parte, necesitan disponer en la máquina del programa intérprete ejecutándose, algo que no es necesario en el caso de

un programa compilado, para los que sólo es necesario tener el programa ejecutable para poder utilizarlo.

Ejemplos de lenguajes interpretados son: Perl, PHP, Python, JavaScript, etc.

A medio camino entre los lenguajes compilados y los interpretados, existen los lenguajes que podemos denominar pseudo-compilados o pseudo-interpretados, es el caso del Lenguaje Java. Java puede verse como compilado e interpretado a la vez, ya que su código fuente se compila para obtener el código binario en forma de bytecodes, que son estructuras parecidas a las instrucciones máquina, con la importante propiedad de no ser dependientes de ningún tipo de máquina (se detallarán más adelante). La Máquina Virtual Java se encargará de interpretar este código y, para su ejecución, lo traducirá a código máquina del procesador en particular sobre el que se esté trabajando.

El lenguaje de programación Java.

¿Qué y cómo es Java?

Java es un lenguaje sencillo de aprender, con una sintaxis parecida a la de C++, pero en la que se han eliminado elementos complicados y que pueden originar errores. Java es orientado a objetos, con lo que elimina muchas preocupaciones al programador y permite la utilización de gran cantidad de bibliotecas ya definidas, evitando reescribir código que ya existe. Es un lenguaje de programación creado para satisfacer nuevas necesidades que los lenguajes existentes hasta el momento no eran capaces de solventar.

Una de las principales virtudes de Java es su independencia del hardware, ya que el código que se genera es válido para cualquier plataforma. Este código será ejecutado sobre una máquina virtual denominada Maquina Virtual Java (MVJ o JVM – Java Virtual Machine), que interpretará el código convirtiéndolo a código específico de la plataforma que lo soporta. De este modo el programa se escribe una única vez y puede hacerse funcionar en cualquier lugar. Lema del lenguaje: “Write once, run everywhere”.

Antes de que apareciera Java, el lenguaje C era uno de los más extendidos por su versatilidad. Pero cuando los programas escritos en C aumentaban de volumen, su manejo comenzaba a complicarse. Mediante las técnicas de programación estructurada y programación modular se conseguían reducir estas complicaciones, pero no era suficiente.

Fue entonces cuando la Programación Orientada a Objetos (POO) entra en escena, aproximando notablemente la construcción de programas al pensamiento humano y haciendo más sencillo todo el proceso. Los problemas se dividen en objetos que tienen propiedades e interactúan con otros objetos, de este modo, el programador puede centrarse en cada objeto para programar internamente los elementos y funciones que lo componen.

Las características principales de lenguaje Java se resumen a continuación:

El código generado por el compilador Java es independiente de la arquitectura.
Está totalmente orientado a objetos.
Su sintaxis es similar a C y C++.
Es distribuido, preparado para aplicaciones TCP/IP.
Dispone de un amplio conjunto de bibliotecas.

Es robusto, realizando comprobaciones del código en tiempo de compilación y de ejecución.

La seguridad está garantizada, ya que las aplicaciones Java no acceden a zonas delicadas de memoria o de sistema.

Algoritmos, Diagramas de Flujo y Pseudocódigo

Los lenguajes de programación, cuentan todos en su haber con un juego de "instrucciones".

Una instrucción no es más que una orden que nosotros le damos a la máquina.Y es que, al fin y al cabo, un programa no es más que una secuencia de instrucciones, escritas en algún lenguaje de programación y pensadas para resolver algún tipo de problema.

Eso sí, si no sabemos resolver este problema, no podremos escribir el programa.
Sería absurdo intentar hacer un programa informático para resolver un ejercicio que requiere de una fórmula, si no sabemos qué fórmula necesitamos, ni cómo aplicarla.Por eso cuando te plantees crear un programa primero tendrás que saber para qué servirá y cómo lo resolverá.

A ti se te puede ocurrir una manera de resolverlo, a tu compañero, otra. Y las dos formas pueden ser correctas.

Este método con el que resolvéis el problema, es lo que se llama algoritmo, y es lo primero que vamos a ver.

¿Qué es un Algoritmo?
Un algoritmo es una secuencia de PASOS a seguir para resolver un problema.
Por ejemplo, cuando quiero ver una película de vídeo, podría hacer los siguientes pasos = algoritmo:
-Elijo una película de las de mi colección.
- Compruebo SI TV y vídeo están conectados a la red (y procedo).
-SI la TV está apagada, la enciendo, SI NO, pues no. Y lo
mismo con el vídeo.
-Introduzco la película en el vídeo. Dejo el estuche sobre el vídeo.
-SI la TV no está en el canal adecuado, la cambio, SI NO, pues no.
-Cojo los mandos a distancia (el del TV y el del vídeo).
-Pulso PLAY en el mando del vídeo.

Definición de algoritmo:

“Un algoritmo es una sucesión finita de pasos (no instrucciones como en los programas) no ambiguos y que se pueden llevar a cabo en un tiempo finito.”

Hay distintas formas de escribir un algoritmo, bien usando un lenguaje específico de descripción de algoritmos, bien mediante representaciones gráficas como diagramas de flujo. Esta última suele ser muy utilizada por los programadores. Veamos un pequeño resumen de creación de diagramas de flujo.

Diagramas de Flujo

Los diagramas de flujo representan la secuencia o los pasos lógicos para realizar una tarea mediante unos símbolos.

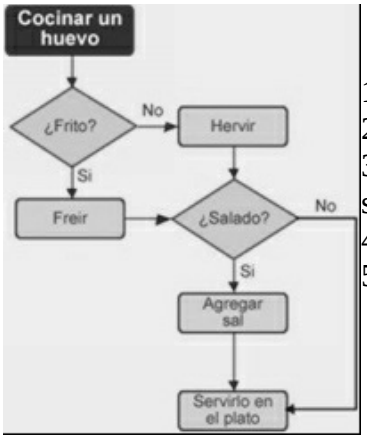
Dentro de los símbolos se escriben los pasos a seguir.

Un diagrama de flujo debe proporcionar una información clara, ordenada y concisa de todos los pasos a seguir. Veamos un ejemplo muy sencillo.

Haremos el algoritmo y el diagrama de flujo para cocinar un huevo para otra persona.

Los pasos a seguir o algoritmo escrito sería:
- Pregunto si quiere el huevo frito.
- Si me dice que sí, lo frio, si me dice que no, lo hago hervido.
- Una vez cocinado le pregunto si quiere sal en el huevo.
- Si me dice que no lo sirvo en el Plato. Si me dice que si le hecho sal y después lo sirvo en el plato.

Ahora vamos hacer el diagrama de flujo:



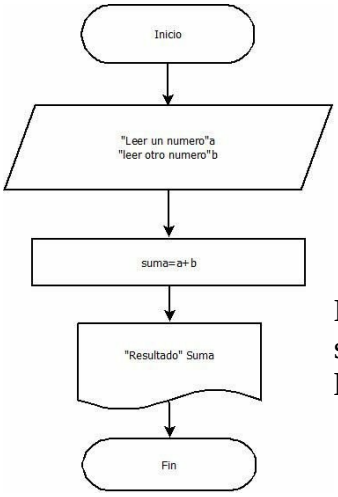
Reglas Básicas Para la Construcción de un Diagrama de Flujo
1. Todos los símbolos han de estar conectados
2. A un símbolo de proceso pueden llegarle varias líneas
3. A un símbolo de decisión pueden llegarle varias líneas, pero sólo saldrán dos (Si o No, Verdadero o Falso).
4. A un símbolo de inicio nunca le llegan líneas.
5. De un símbolo de fin no parte ninguna línea.

Los símbolos que se usan para realizar los diagramas de flujo son los siguientes:

En el Símbolo de decisión a tomar, los valores de salida pueden ser SI o NO o VERDADERO o FALSO.

El símbolo de Inicio o Final del Diagrama puede ser un cuadrado con los bordes redondeados o una elipse.

Veamos otro ejemplo muy sencillo. Queremos hacer un programa informático que nos sume dos números y nos dé el resultado en pantalla.



El símbolo de resultado es un símbolo usado en los diagramas para soluciones con el ordenador. Es el símbolo de salida del resultado por la pantalla del ordenador.

En el ejercicio tenemos el inicio y el fin, una entrada de datos, para meter los 2 números, una operación a realizar, la suma, y un resultado a mostrar. Cada uno de esos pasos con su símbolo correspondiente en el diagrama.

Recuerda que estos diagramas de flujo te facilitarán mucho la tarea antes de empezar con tu programa. Independiente del lenguaje de programación que vaya a usarse; un algoritmo que esté escrito en pseudocódigo o con un diagrama de flujo es fácilmente traducible a muchos lenguajes de programación.

Pseudocódigo

El pseudocódigo es una forma de escribir los pasos, pero de la forma más cercana al lenguaje de programación que vamos a utilizar, es como un falso lenguaje, pero en nuestro idioma, en el lenguaje humano. Una de las mayores dificultades con las que se encuentran los hispanoparlantes que empiezan a programar es el idioma. Por eso es bueno utilizar el pseudocódigo, algo así como un falso lenguaje de programación, que ayuda a asimilar con más facilidad las ideas básicas.

Hay software intérprete de pseudocódigo. Por ejemplo, PSeInt es una herramienta para asistir a un estudiante en sus primeros pasos en programación, mediante un simple e intuitivo pseudolenguaje en español (complementado con un editor de diagramas de flujo).Por ejemplo si queremos escribir algo en pantalla, en pseudocódigo podríamos poner:

```
Escribir "Hola"
Escribir 20+30.
```

Realmente el pseudocódigo lo podríamos escribir como nosotros quisiéramos, ya que realmente no es el programa en sí, solo es una ayuda para posteriormente realizar el programa mediante el lenguaje de programación que utilicemos, eso sí, es de gran ayuda.

Recuerda que el pseudocódigo para un programador es fundamental. Si sabes hacer el pseudocódigo del programa, pasarlo a cualquier lenguaje de programación es muy sencillo, solo tendrás que aprender los comandos equivalentes a las instrucciones en pseudocódigo.

Además, la mayoría de los lenguajes utilizan prácticamente los mismos comandos en su lenguaje.

Sigamos hablando un poco más sobre el pseudocódigo. Para especificar el principio y el fin del programa pondremos:

```
Inicio
    Aquí iría el programa en pseudocódigo
Fin
```

Otra forma muy utilizada sería:

```
Proceso NombreDelPrograma
    Aquí iría el programa en pseudocódigo
FinProceso
```

```
Leer edad
```

nos lee desde lo que el usuario marque desde el teclado y guarda el valor, por ejemplo dentro de una variable, en este caso la variable Edad (luego veremos lo que son las variables).

```
Escribir “este texto saldrá en pantalla”
```

La instrucción precedente está diciendo que se debe mostrar en pantalla el texto que está entrecomillado. Veremos más adelante que todo lo que vaya entre comillas se lo toman los programas como un texto

Nota: Todas las actividades de este tema las realizaremos en PSeint

Práctica 01: Realizar un pseudocódigo en PSeint que muestre el mensaje:
Hola mundo! Mi nombre es: nombrealumno

Elementos Comunes en la Programación

Comentarios

Poner comentarios de lo que vamos haciendo es muy útil, sobre todo cuando llega la hora de revisar el programa, si no, más de una vez nos encontraremos diciendo ¿qué hacía esto aquí?.

No cuesta nada documentar el programa y nos ahorrará dolores de cabeza.La norma que se sigue en todos los programas es poner // delante de los comentarios para identificarlos:

```
// Esto será un comentario y no hará nada en el programa, solo servirá para leerlo nosotros
```

Las variables

Esto es una de las cosas más importantes en programación. Entender bien que es una variable es fundamental.

Una variable es como una caja donde metemos cosas (datos). Estos datos los podemos ir cambiando, ahora meto un 3, ahora lo quito y meto un 5. Una variable tiene un nombre, que

puede ser una letra, una palabra, varias palabras unidas por el guión bajo o varias palabras sin separar.

Ejemplos para escribir el nombre de una variable:
vidasPerdidas, vidas_perdidas. Ojo las mayúsculas y minúsculas son muy importantes en las variables, no es la misma variable numero que Numero,
son dos diferentes. ¿Pero no lleva acento la palabra número?

OJO NO se pueden poner acentos en el nombre de las variables. Las variables, además de un nombre, también tienen un valor que es lo que hay dentro de ella (recuerda dentro de la caja) en ese momento y que puede ir variando según se vaya desarrollando el programa, por eso se llama variable.

Una variable dependiendo de su valor puede ser:
- Variable Numérica, si solo puede tener un valor numérico.
- Variable de texto, si solo puede contener texto dentro de ella. Esto podría ser desde un carácter hasta incluso varias frases

Típicamente usamos la expresión string para referirnos a una cadena de texto

En las variables de texto, su valor (el texto), debe ir entre comillas, para diferenciar que el texto es texto y no es el nombre de otra variable.

Por ejemplo definamos 2 variables:

```
diferentesVidas = "Cinco"
vidas_Fin = "5"
```

En los dos casos el valor es un texto, nunca el valor de 5.
Las numéricas no llevan comillas en su valor.

Por ejemplo
```
vidas = 5
```

En este caso su valor sí que es el número 5. Hay otras variables que se llaman booleanas que solo pueden tener dos valores true o false.

En ocasiones true se puede sustituir por el valor 1 y false por el 0.

Veamos algunos ejemplos de los tipos de variables:
```
edad=3; //variable numérica
```

Observar que el texto que está después de las barras no hace nada es solo un comentario.

```
variableDeTexto= “Edad”;
edad= 3;
variableNumerica= edad + 2 ;
```

En este caso variableNumerica va a valer 5 ya que edad es 3 y le hemos sumado 2 a ese dato
Ese número puede cambiar a lo largo del programa y es por eso que se llama variable

```
variableBooleana = true;
```

en este caso sería de valor 1 ¿Te has dado cuenta que hemos puesto un punto y coma (;) al acabar de definir cada variable?

En muchos lenguajes de programación se terminan las sentencias con (;) para decir al programa que pasamos a otra instrucción diferente.

Las variables según el lenguaje de programación que se use pueden verse obligadas a ser declaradas antes de usarlas

Declarar las variables es decir qué tipo de dato es el que queremos representar con ese nombre de variable.

Veamos un ejemplo:

```
definir finalizarPrograma como Logico;
```

De la forma anterior la variable finalizarPrograma puede tener los valores: verdadero (true) y falso (false)

Observar la siguiente estructura de Pseint:

```
Repetir
        …
Hasta que finalizarPrograma  =  verdadero
```

Los puntos suspensivos hacen alusión a una serie de instrucciones que no son relevantes para lo que queremos detallar en estos momentos pero que probablemente aparecerán

En el texto de arriba observamos que vamos a estar repitiendo una serie de instrucciones hasta que finalizarPrograma sea igual a verdadero ¿ tiene sentido verdad ?

Podemos sumar, restar, multiplicar y dividir las variables o cualquier número.La resta se indicará con -, la suma con +, la multiplicación con *, la división con / y la potencia con ^.

Una operación menos habitual es calcular el resto de una división. Esto también tiene un símbolo asociado: % (Pseint soporta tambien MOD si se prefiere al uso de %)

Esto se suele usar con frecuencia para saber si un número es múltiplo de otro (por ejemplo, será múltiplo de 10 si su resto entre 10 es 0, o será impar si su resto entre 2 es 1).

15%2 nos dará el resto de dividir 15 entre 2, es decir será 1. Además como el resto es 1 el número será impar.

Práctica 02: Realizar un un algoritmo que calcule el IVA (16%) de un producto dado su precio de venta sin IVA.

Cuando tenemos varias variables numéricas o de texto las podemos declarar todas a la vez de la siguiente foma:

```
definir altura, edad, peso como Entero;
```

En la instrucción anterior hemos declarado tres variables de tipo entero

Hay que tener en cuenta que en el momento de la definición las variables no tienen un valor que les hayamos asignado. Es especialmente importante tener cuidado con que las variables siempre controlemos el valor que puedan tener. Una varible sin un valor inicial puede generar errores importantes en el código que estemos desarrollando.

Ejemplo de asignación de valor inicial y suma de dos variables:

```
definir primerNumero, segundoNumero, tercerNumero como Entero;
primerNumero <- 5;
segundoNumero <- 7;
tercerNumero <- segundoNumero + primerNumero;
Escribir tercerNumero;
```

En el anterior trozo de código observamos que creamos tres variables de tipo numérico entero. Le asignamos a las dos primeras respectivamente los valores 5 y 7. Hacemos que la suma de ambas variables sea el valor de la tercera variable. Así la variable tercerNumero valdrá: 5 + 7 y eso será lo que se escribirá en pantalla

Introducción por teclado

Anteriormente hemos visto una operación que hemos hecho nosotros completamente desde código pero podríamos haber hecho que fuera el usuario quién introdujera los valores. Veamos ejemplos

```
definir primerNumero, segundoNumero, tercerNumero como Entero;
Leer primerNumero;
Leer segundoNumero;
tercerNumero <- segundoNumero + primerNumero;
Escribir tercerNumero;
```

En este caso el programa le solicita al usuario que introduzca por teclado los números para las variables y luego hacemos la suma como antes.

Nota: A partir de ahora definiremos las variables que usemos en las prácticas

- **Práctica 03**: Realizar un algoritmo que sirva para convertir pulgadas a centímetros. (Recuerda que 1 pulgada = 2.54 centímetros). (utilizar una variable llamada pulgada)

- **Práctica 04**: Diseña un algoritmo que pida el valor del lado de un cuadrado y muestre el valor de su perímetro y el de su área.

- **Práctica 05**: Diseña un algoritmo que pida el valor de los dos lados de un rectángulo y muestre el valor de su perímetro y el de su área.

Estructuras de control

Las estructuras de control son de tres tipos:
- Secuenciales
- Selectivas
- Repetitivas

Estructuras secuenciales

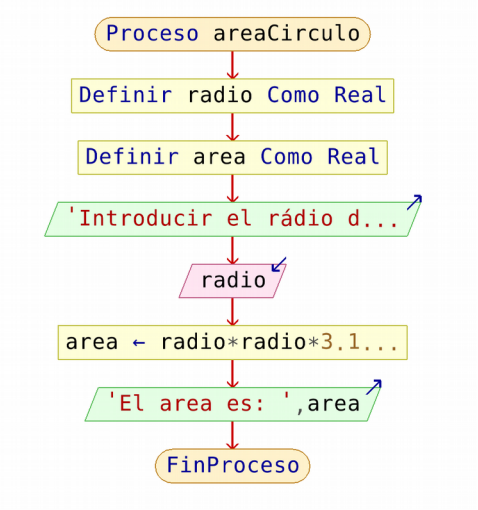
Una estructura de control secuencial, en realidad, no es más que escribir un paso del algoritmo detrás de otro. El que primero que se encuentre escrito será el que primero se ejecute.

```
Proceso areaCirculo
    definir radio Como Real;
    definir area como real;
    Escribir "Introducir el rádio de la circunferencia";
    Leer radio;

    area <- radio * radio * 3.14159;
    Escribir "El area es: ", area;

FinProceso
```

Las instrucciones se irán ejecutando una detrás de otra hasta el final. Observemos su diagrama de flujo:



Es fácil ver que se ejecuta una detrás de otra hasta que llega al final.

Estructuras selectivas

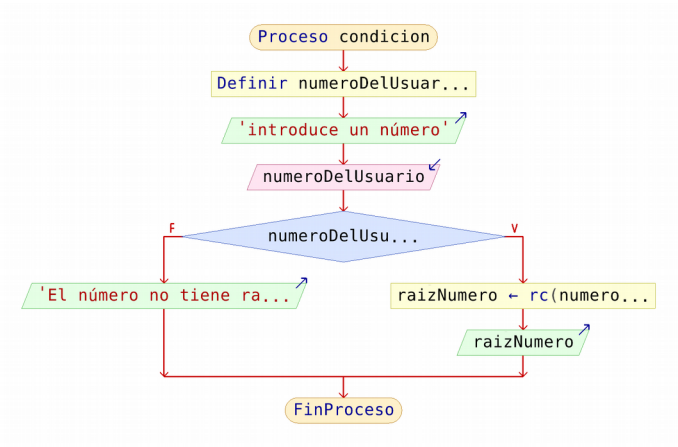
Estas estructuras se utilizan para TOMAR DECISIONES (por eso también se llaman estructuras de decisión o alternativas).

Lo que se hacen es EVALUAR una condición, y, a continuación, en función del resultado, se lleva a cabo una opción u otra.

Alternativas simples
(condicional IF)
Son los conocidas condicionales SI (if en ingles). Su traducción sería: "si se cumple esto… entonces hacemos algo".

Ejemplo:

```
definir numeroDelUsuario, raiz como Real;
Escribir “introduce un número”;
Leer numeroDelUsuario;
Si  numeroDelUsuario >= 0 Entonces
        raiz ← raiz_cuadrada(numeroDelUsuario);
        Escribir raiz;
SiNo
        Escribir “El número no tiene raíz por ser negativo”;
                                FinSi
```



El diagrama de flujo queda:

Ahora podemos observar dos caminos alternativos que se elegirán según lo que haya introducido el usuario

Las estructuras selectivas hacen uso de expresiones lógicas de comparación para determinar la selección que vayan a realizar. Para esas expresiones lógicas de comparación se hace uso de los siguientes operadores:

Operador relacional	*Significado*	*Ejemplo*
>	Mayor que	3>2
<	Menor que	"ABC"<"abc"
=	Igual que	4=3
<=	Menor o igual que	"a"<="b"
>=	Mayor o igual que	4>=5

Veamos de nuevo el condicional que tenemos puesto arriba:

```
Si  numeroDelUsuario >= 0 Entonces
        raiz ← raiz_cuadrada(numeroDelUsuario);
        Escribir raiz;
SiNo
        Escribir “El número no tiene raíz por ser negativo”;
FinSi
```

Se puede observar que aparece la expresión: “SiNo” mediante esa expresión podemos especificar que pasaría si no cumple la condición. Es el famoso trío "si se cumple ... haces esto ... sino esto otro".

Práctica 06: Diseña un algoritmo que, dado un número entero, muestre por pantalla el mensaje "El número es par." cuando el número sea par y el mensaje "El número es impar." cuando sea impar. (Una pista: un número es par si el resto de dividirlo por 2 es 0, e impar en caso contrario.)

Práctica 07: Diseña un algoritmo que, dado un número entero, determine si éste es el doble de un número impar. (Ejemplo: 14 es el doble de 7, que es impar.)

Práctica 08: Diseña un algoritmo que, dados dos números enteros, muestre por pantalla uno de estos mensajes: "El segundo es el cuadrado exacto del primero.", "El segundo es menor que el cuadrado del primero." o "El segundo es mayor que el cuadrado del primero.", dependiendo de la verificación de la condición correspondiente al significado de cada mensaje.

Veamos cómo sería la estructura en muchos lenguajes:

if (condición) {se hace esto} else {si no cumple la condición se hace esto otro};

Lo que hay entre los primeros corchetes es lo que haría el programa si se cumple la condición. Lo que hay entre los corchetes después del else es lo que haría si la condición no se cumple.

Alternativas múltiples o con varias condiciones.

Es muy probable que tengamos la necesidad de incluir en nuestros programas alternativas con muchas opciones posibles (varios SI diferentes).

variableOpciones= un valor a elegir, por ejemplo desde el teclado
if (variableOpciones=0) {lo que corresponda};
if (variableOpciones=1) {lo que corresponda};
if (variableOpciones=2) {lo que corresponda};
…….

Podemos poner tantas if como queramos.

En este caso el programa realizará las acciones en función de lo que valga la variable variableOpciones.

Ejercicio propuesto: Intenta hacer un programa que nos diga la nota de un alumno, en función del número sacado en el examen.

Por ejemplo: si saca menos de 5 “Suspenso”, Si saca entre 5 y 6 suficiente, si saca entre 6 y 8 “notable”.

También existe la posibilidad de que deban de cumplirse dos condiciones a la vez:

if (condición1 && condición2) {Se cumple esto}

Esto sería si se cumplen las condiciones 1 y 2 a la vez.

if (condición1 && condición2) {Se cumple esto} else {se cumple esto otro}

Esto sería si se cumplen las condiciones 1 y 2 a la vez o si no se cumplen a la vez. OJO tienen que cumplirse las dos a la vez. Si solo se cumple una de las dos se realizará lo que tengamos en los corchetes del else.

Los símbolos && significan "y", es decir si se cumple la condición1 y la condición2 a la vez.

Aquí tienes una tabla con los operadores lógicos utilizados para posibilidades de opciones:

Operador lógico	*Significado*	*Ejemplo*
& ó Y	Conjunción (y).	(7>4) & (2=1) //falso
| ó O	Disyunción (o).	(1=1 | 2=1) //verdadero
~ ó NO	Negación (no).	~(2<5) //falso

Atención: lo anterior es meramente para ir viendo como expresaremos en lenguajes del estilo de C, java,… De momento nos quedaremos con la idea de que podemos hacer preguntas compuestas mediantes la conjunción: “y”, “o”

```
Si  (miNumero > 3)  Y (miNumero < 5) Entonces
        Escribir “Mi número es 4”
FinSi
```

Mirando lo anterior se observa que la variable miNumero debe ser mayor que 3 y a la vez debe ser menor que 5. Eso significa que si es un número entero sólo puede ser 4. Como podemos ver hacemos uso de la conjunción “Y” para que se cumplan las dos condiciones a la vez. Exactamente igual que cuando hablamos

```
Si  (miNumero < 0)  o (miNumero > 0) Entonces
        Escribir “Mi número no es igual a cero”
FinSi
```

Nos propone dos opciones: la variable miNumero es menor que cero ó la variable miNumero es mayor que cero. Si se cumple cualquiera de esas dos condiciones (que sea menor que cero o que sea mayor que cero) mostraremos el mensaje: “Mi número no es igual a cero”

Estructuras REPETITIVAS

Estas estructuras son instrucciones que se repiten formando un bucle (algo que se repite una y otra vez).

El bucle se repite mientras se cumpla una condición que ha de ser especificada claramente.

Cuando deje de cumplirse la condición, se sale fuera del bucle y no se repiten más las instrucciones. Un BUCLE (loop, en inglés) es un trozo de algoritmo, pseudocódigo o de programa cuyas instrucciones son repetidas un cierto número de veces, mientras se cumple una cierta condición que ha de ser claramente especificada.

Básicamente, existen tres tipos de estructuras repetitivas:
- Los bucles "mientras…" (bucles while)
- Los bucles "repetir hasta (hacer)… (bucles do-while, ó bucles do-until)
- Los bucles “para...” (bucles for)

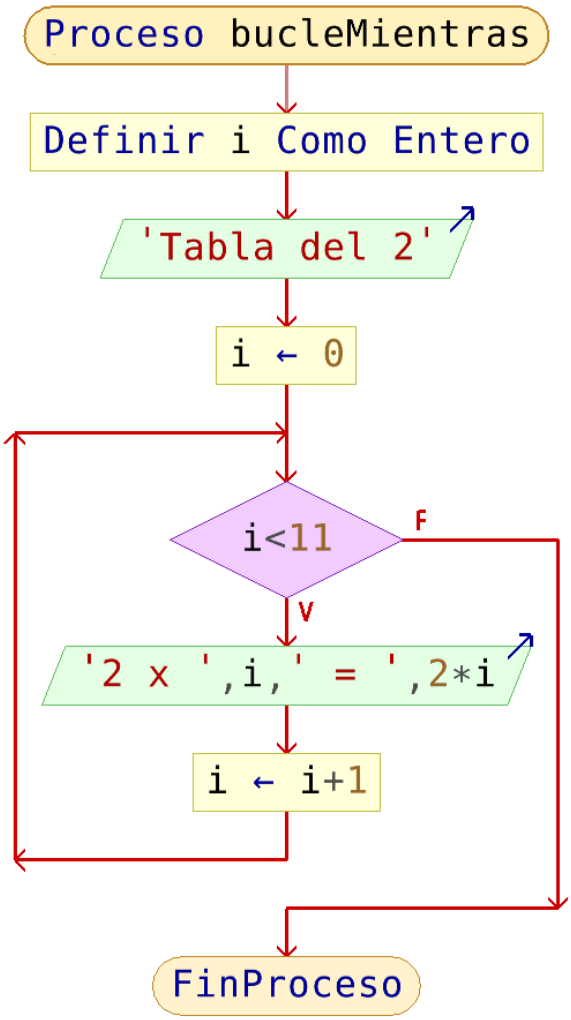
Bucle Mientras

Veamos un ejemplo:

```
Proceso bucleMientras

    definir i Como Entero;
    Escribir "Tabla del 2";
    i ← 1;
    Mientras i < 11 Hacer
        Escribir "2 x ",i," = ",2*i;
        i ←  i + 1;
    FinMientras

FinProceso
```



El anterior código nos muestra la tabla del 2. Observar que le hemos asignado el valor inicial a la variable i por fuera del bucle y que en cada iteración vamos sumándole uno a esa variable:

```
i ←  i + 1;
```

Mientras la variable i es menor a 11 vamos imprimiendo en pantalla y aumentando el contador (la variable i) Cuando la variable llega a 11 sale del bucle y finaliza el proceso

Es MUY IMPORTANTE darse cuenta que en los bucles de este tipo (bucles while) no se ejecutan nunca si no se cumple la condición al incio. Si la variable i se hubiera inicializado a 20 por ejemplo:

i ← 20

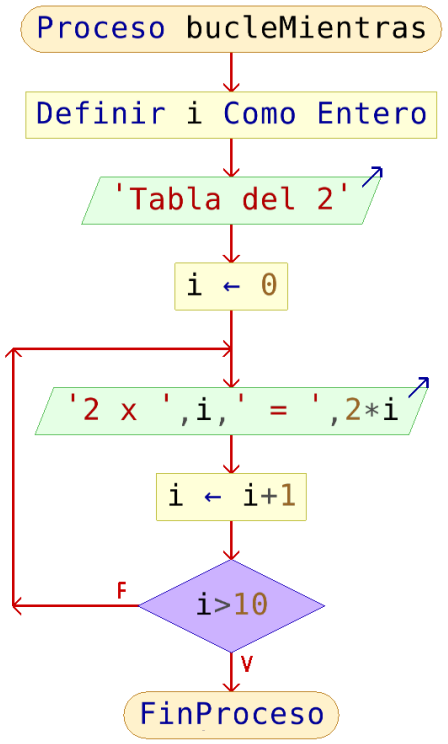
Nunca se habría ejecutado ninguna instrucción del bucle

Bucles repetir hasta

Veamos el código anterior adaptándolo con este tipo de bucle

```
Proceso bucleRepetirHasta

        definir i Como Entero;
        Escribir "Tabla del 2";
        i <- 0;
        Repetir
                Escribir "2 x ",i," = ",2*i;
                i <- i + 1;
        Hasta que i > 10;
FinProceso
```



Comparar el diagrama de flujo anterior y el actual. Las diferencias principales son que la condición en el anterior está al comienzo del bucle y en este está al final. Eso significa que como mínimo se ejecutará una vez las instrucciones que hayamos puesto en este bucle. Siguiendo con el ejemplo anterior, si ahora pusiéramos que

i ← 20

al inicializar la variable nos realizaría la salida en pantalla de la tabla hasta que luego se diera cuenta que el número que hemos introducido no cumple las condiciones que queríamos para imprimir en pantalla (los número del 1 al 10)

La otra característica importante es que ahora cambiamos la condición. Antes decíamos:
“Mientras i sea menor que 11”
Ahora decimos:
“Hasta que i sea mayor que 10”

En ambos casos realizaremos el bucle para los valores que llegan hasta el 10 pero lo hemos expresado de formas diferentes

Bucles para (bucles for)

Este tipo de bucles tiene un buen comportamiento cuando están asociados a un contador. Se establece el inicio, la condición de parada y el valor que va tomando el contador en cada paso

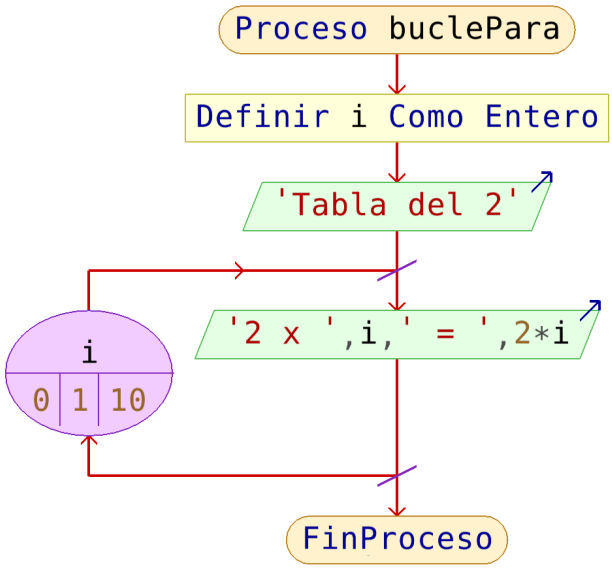
Siguiendo con el ejemplo anterior:

```
Proceso buclePara

        definir i Como Entero;
        Escribir "Tabla del 2";
        Para i <- 0 Hasta 10 Con Paso 1 Hacer
                Escribir "2 x ",i," = ",2*i;
        FinPara

FinProceso
```

Y el diagrama de flujo asociado:



La variable i irá dando pasos de uno en uno desde el valor cero hasta que llegue al valor 10 en cuyo caso acabará el bucle

Se puede observar que está instrucción es más cómoda para este caso de mostrar la tabla de multiplicar que los dos bucles anteriores al realizarse la inicialización de la variable, la condición de parada y el incremento que tiene el contador en una única instrucción

Sin embargo hay múltiples casos en los que resultan más apropiados los anteriores bucles
Un ejemplo es finalizar el bucle cuando lo solicite el usuario:

```
Proceso mostrarAleatorios
        definir tecla como texto;
        definir numeroAleatorio Como Entero;
        Escribir "Este programa muestra numeros aleatorios hasta que el usuario escribe la tecla f";

        Repetir
                numeroAleatorio <-  azar(100);
                Escribir numeroAleatorio;
                Leer tecla;
        Hasta Que tecla = "f";
FinProceso
```

El bucle para (bucle for) obliga a definir una variable contador y establecerle valores iniciales, etc. En el caso anterior no precisamos nada de eso al ser un proceso de generación de números aleatorios hasta que el usuario introduzca la letra f

- **Práctica 09**: Crear un algoritmo que le solicite al usuario un número y muestre la tabla de multiplicar para ese número. Hacerlo para los tres tipos de bucles que hemos visto

- **Práctica 10**: Implementa un algoritmo que muestre todos los múltiplos de 6 entre 6 y 150, ambos inclusive.

- **Práctica 11**: Implementa un algoritmo que muestre todos los múltiplos de n entre n y m·n, ambos inclusive, donde n y m son números introducidos por el usuario.

- **Práctica 12**: Implementa un algoritmo que muestre todos los números potencia de 2 entre 20 y 230, ambos inclusive.

- **Práctica 13**: Diseña un algoritmo que, dados cinco números enteros, determine cuál de los cuatro últimos números es más cercano al primero. (Por ejemplo, si el usuario introduce los números 2, 6, 4, 1 y 10, el programa responderá que el número más cercano al 2 es el 1.)

Creación de funciones y procedimientos (subprocesos)

En muchos casos, nos encontraremos con tareas que tenemos que repetir varias veces en distintos puntos de nuestro programa. Si tecleamos varias veces el mismo fragmento de programa no sólo tardaremos más en escribir: además el programa final resultará menos legible, será más también será más fácil que cometamos algún error alguna de las veces que volvemos a teclear el fragmento repetitivo, o que decidamos hacer una modificación y olvidemos hacerla en alguno de los fragmentos. Por eso, conviene evitar que nuestro programa contenga código repetitivo. Una de las formas de evitarlo es usar "subrutinas", una posibilidad que la mayoría de lenguajes de programación permiten, y que en ocasiones recibe el nombre de "procedimientos" o de "funciones" (existe algún matiz que hace que esas palabras no sean realmente sinónimas y que comentaremos más adelante).

Vamos a empezar por crear un subproceso (o "subrutina", o "procedimiento") que escriba 20 guiones, que podríamos utilizar para subrayar textos. Un programa completo que escribiera tres textos y los subrayara podría ser:

```
Proceso EjemploDeSubprocesos
    Escribir " Primer ejemplo"
    Para x <- 1 Hasta 20 Hacer
        Escribir Sin Saltar "-"
    FinPara
    Escribir ""

    Escribir " Segundo ejemplo"
    Para x <- 1 Hasta 20 Hacer
        Escribir Sin Saltar "-"
    FinPara
    Escribir ""

    Escribir " Tercer ejemplo"
    Para x <- 1 Hasta 20 Hacer
        Escribir Sin Saltar "-"
    FinPara
    Escribir ""
FinProceso
```

Mediante los subprocesos podemos escribirlo mucho mejor:

```
Subproceso subrayar()
    Para x <- 1 Hasta 20 Hacer
        Escribir Sin Saltar "-"
    FinPara
    Escribir ""
FinSubproceso


Proceso EjemploDeSubprocesos
    Escribir " Primer ejemplo"
    subrayar()

    Escribir " Segundo ejemplo más grande"
    subrayar()

    Escribir " Tercer ejemplo muchísimo más grande que el anterior"
    subrayar()
FinProceso
```

Lo primero que debemos entender es que el programa se iniciará donde siempre (allí donde hayamos escrito proceso o algoritmo). Así que no debemos confundirnos por ver primero que nada escrito Subproceso.

Observar que cada vez que se encuentra con la expresión: subrayar( )

Ejecuta el trozo de código que se llama: “Subproceso subrayar()” . De esa forma evitamos estar copiando siempre el mismo código.

Pero todavía puede ser mejor. Ya que dentro de los paréntesis le podemos pasar información al subproceso. Y así conseguir que nos ponga más o menos guiones:

```
Subproceso subrayar( cantidadDeGuiones )
    Para x <- 1 Hasta cantidadDeGuiones Hacer
        Escribir Sin Saltar "-"
    FinPara
    Escribir ""
FinSubproceso

Proceso EjemploDeSubprocesos
    Escribir " Primer ejemplo"
    subrayar(10)

    Escribir " Segundo ejemplo más grande"
    subrayar(25)
    Escribir " Tercer ejemplo muchísimo más grande que el anterior"
    subrayar(50)
FinProceso
```

Aquí se le está dando al subproceso diferentes datos: 10, 25, 50 para que así en ocasiones nos imprima más o menos guiones.

El subproceso recibe esa información que le pasamos en la variable: cantidadDeGuiones

Otro ejemplo: Vamos a variar el código para que le enviemos dos datos al subproceso: El texto que queremos que se subraye y la cantidad de guiones que queremos que use para subrayar

```
Subproceso subrayar( textoParaSubrayar, cantidadDeGuiones )
    Escribir textoParaSubrayar
    Para x <- 1 Hasta cantidadDeGuiones Hacer
        Escribir Sin Saltar "-"
    FinPara
    Escribir ""
FinSubproceso


Proceso EjemploDeSubprocesos

    subrayar(" Primer ejemplo",10)

    subrayar(" Segundo ejemplo más grande",25)

    subrayar(" Tercer ejemplo muchísimo más grande que el anterior",50)
FinProceso
```

Observar lo limpio que queda. Cuando leemos:

subrayar(" Primer ejemplo",10)

no nos es difícil imaginar que estamos subrayando ese texto con una raya de longitud 10

Estos subprocesos (subrutinas, funciones, procedimientos, métodos) nos permiten ahorrar estar escribiendo muchas veces el mismo código y nos otorgan una lectura más limpia de nuestro código

Finalmente veamos una opción adicional que tienen. Es la opción de devolver un resultado.

Vamos a crear un subproceso que nos calcule el valor absoluto de un número:

```
Funcion resultado <- valorAbsoluto ( numeroRecibido )

    Si numeroRecibido >= 0 Entonces
        resultado <- numeroRecibido
    SiNo
        //como es un número negativo si multiplicamos por -1
        //obtenemos lo mismo pero en positivo
        resultado <-  ( -1 * numeroRecibido )
    FinSi
Fin Funcion

Algoritmo ProbandoSubprocesos

    Escribir "El valor absoluto de -5 es: "
    Escribir valorAbsoluto(-5);

    Escribir "El valor absoluto de 7 es: "
    Escribir valorAbsoluto(7);


FinAlgoritmo
```

Si miramos dentro de las funciones disponibles dentro de Pseint observamos que aparece la función: abs() que nos obtiene precisamente el valor absoluto de un número. De esta forma podemos empezar a entender que nosotros mismos nos podemos crear las funciones que necesitemos para poderlas usar

Práctica 13: Queremos hacer un algoritmo que calcule el factorial de un número entero positivo. El factorial de n se denota con n!, pero no existe ningún operador que permita efectuar este cálculo directamente. Sabiendo que n! = 1 · 2 · 3 · . . . · (n − 1) · n y que 0! = 1, haz un algoritmo que pida el valor de n y muestre por pantalla el resultado de calcular n!.

Práctica 14: Usando el código anterior como una función desarrollar el siguiente programa: El número de combinaciones que podemos formar tomando m elementos de un conjunto con n elementos es: C(m,n) = n! / (n − m)!m! . Diseña un algoritmo que pida el valor de n y m y calcule Cm n . (Ten en cuenta que n ha de ser mayor o igual que m.) (Puedes comprobar la validez de tu programa introduciendo los valores n = 15 y m = 10: el resultado es 3003.)

Array

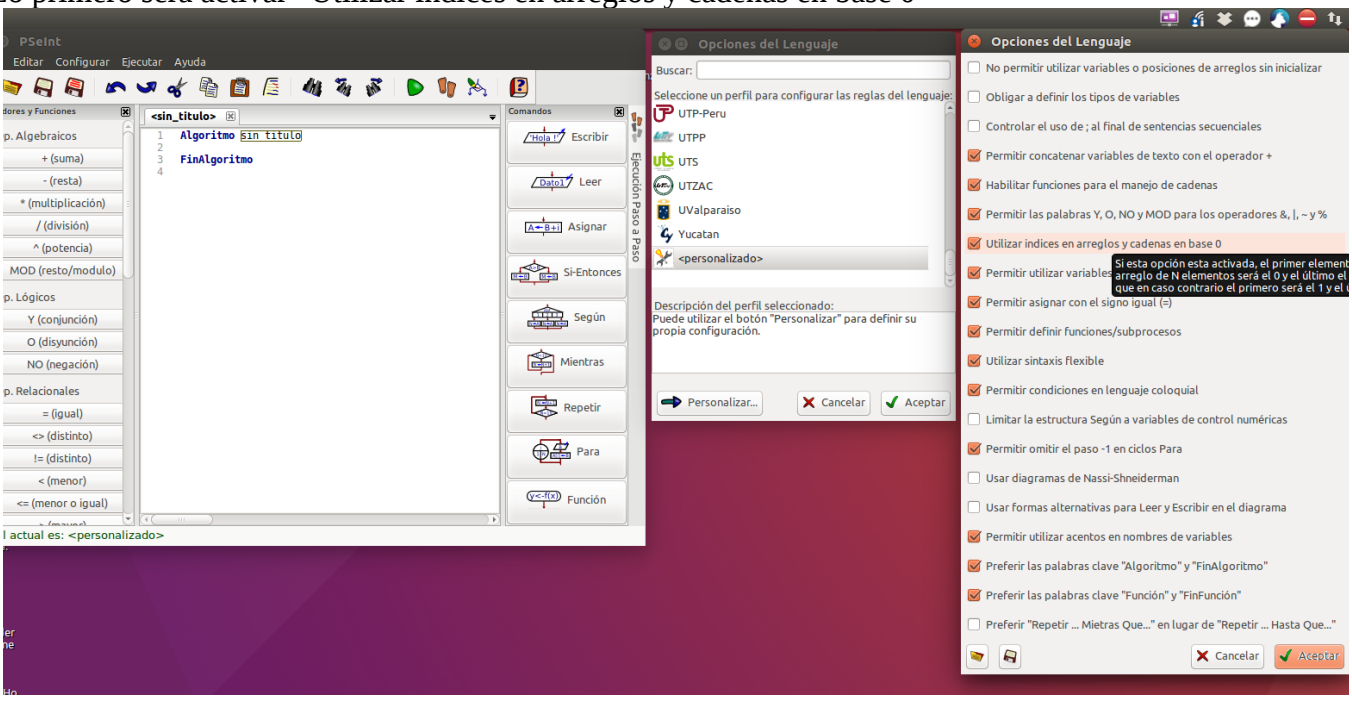
En programación se denomina matriz, vector (de una sola dimensión) o formación (en inglés array) a una zona de almacenamiento contiguo que contiene una serie de elementos del mismo tipo, los elementos de la matriz. Desde el punto de vista lógico una matriz se puede ver como un conjunto de elementos ordenados en fila (o filas y columnas si tuviera dos dimensiones).

En principio, se puede considerar que todas las matrices son de una dimensión, la dimensión principal, pero los elementos de dicha fila pueden ser a su vez matrices (un proceso que puede ser recursivo), lo que nos permite hablar de la existencia de matrices multidimensionales, aunque las más fáciles de imaginar son los de una, dos y tres dimensiones.

Para trabajar con array debemos:
1- declarar el array
2- Crear el array
3- inicializar los elementos del array
4- finalmente usar el array

Veamos el procedimiento en Pseint

Lo primero será activar “Utilizar índices en arreglos y cadenas en base 0”



Está en:
configurar→ opciones del lenguaje → personalizar

Realmente activamos esta opción para irnos acostumbrando para los lenguajes C, C++, C#, java,… que todos ellos empiezan los índices de los array en el número 0

Ahora usemos este código:

```
Algoritmo ejemploArrayBasico

    definir i como entero;

    Definir almacen como entero; //se define la variable

    Dimension almacen[5]; //se crea el objeto que vamos a alcanzar con esa
variable

    //introducimos los primeros números impares que conocemos
    almacen[0] <- 1;
    almacen[1] <- 3;
    almacen[2] <- 5;
    almacen[3] <- 7;
    almacen[4] <- 9;

    //Ahora los mostramos en pantalla recorriendo todo el array
    Para i<-0 Hasta 4 Con Paso 1 Hacer
        escribir almacen[i];
    Fin Para
FinAlgoritmo
```

Debemos darnos cuenta que cada uno de los elementos del array es una variable que podemos usar. Así:

almacen[3]

es una variable en la que tenemos guardado el número 7.

Si se prefiere se puede pensar en un array como en un conjunto de variables con la ventaja, respecto a las varibles que ya conocemos, que podemos usar un índice numérico para movernos de una variable a otra

En el ejercicio anterior podríamos haber escrito lo siguiente y habría funcionado también:

```
Algoritmo ejemploArrayBasico

    definir i como entero;
    Definir almacen1, almacen2, almacen3, almacen4, almacen0 como entero; //se
define la variable


    //introducimos los primeros números impares que conocemos
    almacen0 <- 1;
    almacen1 <- 3;
    almacen2 <- 5;
    almacen3 <- 7;
    almacen4 <- 9;

    //Ahora los mostramos en pantalla recorriendo todas las variable

    escribir almacen0;
    escribir almacen1;
    escribir almacen2;
    escribir almacen3;
    escribir almacen4;
FinAlgoritmo
```

Con este último código hemos obtenido el mismo resultado, pero cuando queremos hacer uso de las variables tenemos que escribirlas nosotros, una a una para manejarlas. En concreto para que nos las muestre en pantalla:

```
escribir almacen0;
escribir almacen1;
escribir almacen2;
escribir almacen3;
escribir almacen4;
```

Sin embargo con el array lo hacíamos con un bucle en el cuál se iba cambiando el índice del array:

```
Para i<-0 Hasta 4 Con Paso 1 Hacer
    escribir almacen[i];
Fin Para
```

Ya sólo por eso resulta interesante un array. Imaginar que tenemos que manejar 200 números por ejemplo.

Consideraciones con los array:

- Cuidado con los índices. El array está definido entre cero y el tamaño del array menos uno:

```
Dimension almacen[14];
```

Para el anterior array, el primer valor válido es en almacen**[0]** y el último es almacen**[13].**

Si usáramos almacen[-1] o por ejemplo almacen[14] nos dará error

- No confundir la variable indice con el elemento del array

Sobre todo al principio, se puede confundir la variable índice que se usa para recorrer el array con el elemento del array que se quiere acceder

Por ejemplo, para guardar en la posición 2 del array el número 11 escribimos:

```
i <- 2;
almacen[i] <- 11;
```

- Dimensionar el array con un tamaño que luego no se nos vaya a quedar corto pero que no sea tan grande que estemos desperdiciando muchos recursos.

Según el lenguaje de programación que usemos y las librerías a las que accedamos quizás tengamos acceso a herramientas de redimensión, pero éstas puede que internamente estén creando un nuevo array y copiando todos los elementos. Es por esto que debemos establecer bien su tamaño al crearlo, siendo preferible tomar un tamaño muy grande antes que quedarnos cortos

En concreto en Pseint tenemos que no podemos crear un array usando una variable:

Dimension almacen[size];

ya que únicamente admite un número entero positivo.

Hagamos uso de un array para guardar los datos que nos introduce el usuario y luego se los mostramos todos juntos:

Algoritmo ejemploArrayBasico

```
    Definir num como entero;
    Dimension num[5];

    Para i<-0 Hasta 4 Con Paso 1 Hacer
        escribir "introduzca un número: ";
        leer num[i];
    Fin Para

    Para i<-0 Hasta 4 Con Paso 1 Hacer
        escribir Sin Saltar num[i], ", ";
    Fin Para

FinAlgoritmo
```

Tomemos del usuario nombres de personas y cuando escriba “Fin” nos muestre todos los nombres:

```
Algoritmo ejemploArrayBasico

    definir i, sizeArray como entero;
    definir textoDelUsuario Como Caracter;
    definir nombres Como Caracter;
    dimension nombres[300]; //observar que sobredimensionamos el array para garantizar que tenemos espacio

    i<-0;
    Repetir
        Escribir "Dame un nombre. Escribir Fin para finalizar: ";
        Leer textoDelUsuario;
        nombres[i] <- textoDelUsuario; //guardamos el nombre en el array

        i <-  i + 1; //aumentamos el indice

    Hasta Que textoDelUsuario = "Fin";

    //Observar que el valor de la variable i ahora es el tama? real del array + 1
    sizeArray <- i - 1;

    //Listamos el contenido del array
    Escribir "Los nombres introducidos son: "
    Para i<-0 hasta sizeArray - 1 con paso 1 Hacer
        Escribir nombres[i]
    FinPara

FinAlgoritmo
```

Observar que hemos sobredimensionado el array para garantizar que nos caben todos los posibles nombres que introduzca el usuario

También debemos fijarnos que el anterior algoritmo guarda la palabra Fin en el array de nombres. Es por eso que recorremos hasta: sizeArray -1 cuando vamos a mostrarlos en pantalla

- **Práctica 15**: Calcular usando un array, el valor mínimo, máximo y valor medio de 7 números introducidos por el usuario. El programa le irá pidiendo todos los números y cuando se hayan introducido mostrará el menor de los números, el mayor y el valor medio

- **Práctica 16**: Mostrar usando un array los dos números mayores de 10 números introducidos por teclado

- **Práctica 17**: Modificar el anterior para que sirva con cualquier cantidad de números mayores que se quiera tener guardados.