Unidad 1

Introducción a la programación

Programación

Índice

Introducción

En la primera unidad de este módulo introduciremos la programación y sus principales nociones básicas hablando desde sus orígenes hasta la programación moderna.

Los principales conceptos serán nombrados y trataremos en detalle lo que es la programación estructurada y modular, así como los principales paradigmas de la programación en general.

Para terminar, hablaremos sobre el pseudocódigo a través de los diagramas de flujo como ayuda a empezar a programar.

Al finalizar esta unidad

- Conoceremos los conceptos básicos relacionados con la programación y el diseño de aplicaciones.
- Describiremos los paradigmas de programación más utilizados.
- Aprenderemos a utilizar sistemas de descripción de programas de alto nivel.

1.1. Orígenes de la programación

Nos referimos a la programación como el proceso con el que se agrupan, diseñan y codifican diferentes instrucciones que al unirse implementan un algoritmo de un uso específico.

Antes se realizaban numerosas tareas de forma manual, debido a esto surgió la programación con el fin de automatizar y llevar a cabo todas estas tareas que resultaban tediosas de realizar dado que eran muy simples y quitaban mucho tiempo. La programación tiene su verdadero origen en las primeras calculadoras de operaciones matemáticas, estas calculadoras, mediante algoritmos muy básicos implementados en su estructura física eran capaces de resolver sumas, restas, etc. Las primeras calculadoras a las que nos referimos recibían a través de tarjetas perforadas la información de los datos de las operaciones en los telares automáticos.

No obstante, podemos decir que no se puede comenzar a hablar de programación como la conocemos hoy en día hasta que Charles Babbage creo la máquina analítica en 1822, la que es considerada la primera computadora, o hasta que Augusta Ada Byron creo el primer algoritmo informático.

Actualmente la programación no es la misma que hace unos años, porque ha ido evolucionando a la par con los dispositivos electrónicos, cuanto más sofisticados son los dispositivos, necesariamente más sofisticados tienen que ser los medios de programación.

Esto se explica con el hecho de que al principio la programación era en lenguaje máquina, es decir, binario (0 y 1), lo que hacía muy difícil su implementación y solo estaba al alcance de unos pocos.

En una segunda instancia, se decidió que lo mejor era sustituir ciertas secuencias de ceros y unos por palabras, surgiendo el lenguaje ensamblador, que seguía en casi total comunicación con el hardware del equipo.

Tras comprobar que este nuevo método tampoco permitía una comprensión fácil del programa, fue cuando empezaron a surgir los lenguajes de alto nivel que permiten que se desarrollen nuevos programas informáticos aún más fáciles y sofisticados.

1.2. Paradigmas de programación

Como se ha nombrado en el punto anterior, la programación ha ido cambiando y evolucionando a lo largo de la historia, dando lugar a diversas técnicas para programar que se parecen, en mayor o menor medida, unas a otras y generalmente están orientadas a problemas distintos. Si una de estas técnicas se empieza a usar de manera continuada y es un sistema de referencia a la hora de generar código de una cierta calidad podemos hablar ya de un paradigma de programación.

Hay distintos paradigmas que difieren unos de otros, pero no son excluyentes entre sí. De hecho, la mayoría de los programadores se basan en varios paradigmas a la vez para poder crear el código más completo y eficiente posible, esta técnica se llama programación multiparadigma.

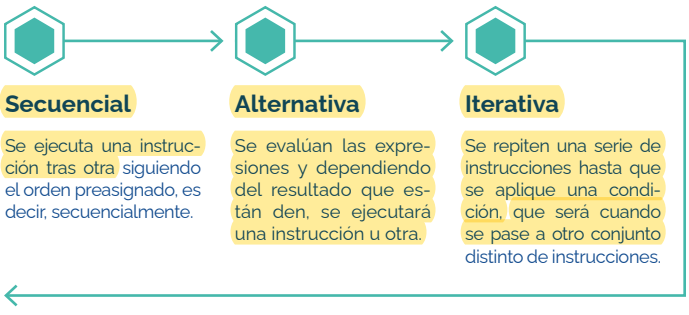
Cuando trabajamos con paradigmas de programación, siguiendo sus reglas y estándares, daremos lugar a un código más limpio y sencillo de entender.

Aunque son muchos los distintos paradigmas de programación que existen, a continuación, vamos a hablar de la programación estructurada y la programación modular.

1.2.1. Programación estructurada

Este paradigma de programación, que viene derivado del paradigma de programación imperativa tiene sus orígenes en los setenta y se crea para poder mejorar la calidad del software desarrollando al mismo tiempo que se reducen costes.

La programación estructura se basa en escribir todo el programa usando simplemente tres tipos de estructuras de control, lo que se llama el teorema del programa estructurado. Estas tres estructuras básicas de control son la siguientes:



Es verdad que hay muchas estructuras de control en la programación, pero todas vienen fundadas por las tres anteriormente nombradas.

Otra estructura, que difiere de las anteriores, muy usada es la de salto incondicional (go to), pero no se usa mucho ya que los expertos en programación la desaconsejan debido a que hace el programa poco legible, lo que deriva en un programa difícil a la hora de la detección de errores o de depurarlo.

Cabe destacar que esta programación estructura no es eficiente a la hora de diseñar un gran programa, es en estos momentos cuando se debe usar la programación orientada a objetos.

Por último, es necesario recordar lo siguiente:

En la programación estructurada, las funciones que manejan datos y estos últimos se definen de forma separada.

1.2.2. Programación modular

El anterior método presentado de programación nos daba las claves para poder realizar un código con una cierta calidad y fácil de mantener, pero sin tener en cuenta un código muy grande.

La programación modular sí que tiene en cuenta esta posibilidad de crecimiento en el código y por ello propone la siguiente solución: descomponer el problema en varios subproblemas de manera sucesiva hasta tener varios problemas los cuales sean más sencillos de corregir en comparación con todo el conjunto a la vez. Esta técnica recibe el famoso nombre de divide y vencerás además de poner en práctica el método de desarrollo de software top-down.

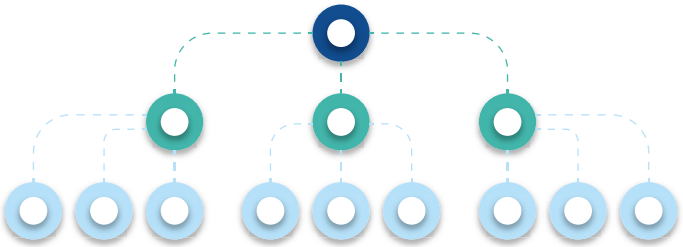


Imagen 1. Ejemplo top-down.

Este código se divide en módulos o subprogramas, que son pequeños bloques que almacenan cada uno de los subprogramas que tendremos que resolver.

A la hora de llevar a cabo el proceso de programación modular los elementos que se usen pueden ser muy distintos y dependen del nivel de abstracción:

- En un nivel alto se tratará de las unidades en el caso de diseños estructurados y de paquetes y librerías en diseños orientados a objetos.
- En un más bajo nivel se tratará de procedimientos y funciones en diseños estructurados y también para diseños orientados a objetos, cada clase debe de ir con sus correspondientes métodos.

Para que la popularización sea la correcta y funcione eficientemente se debe de realizar un diseño de calidad siguiendo los siguientes criterios:

- Cada módulo debe de tener un único punto de entrada y otro de salida.
- Los módulos deben de realizar por sí solos una única función bien definida.
- Los módulos solo dependen de las entradas.
- Debe de poder verse un módulo entero en una pantalla (unas 30-50 líneas de código para cada módulo).
- Debe de haber una cohesión máxima y un acoplamiento mínimo, que están directamente relacionados, cuanto menor sea el acoplamiento, mayor cohesión y al revés.

Cohesión

La cohesión de un módulo hace referencia a la relación que haya entre los distintos elementos de software albergados en cada módulo. Estos elementos de software los conforman las instrucciones, definiciones de datos o llamadas a otros módulos.

Existen tipos diferentes de cohesión:

1. **Cohesión funcional.** Todos los elementos software que constituyen el módulo tienen una sola tarea definida. Es decir, cada uno de los elementos forma parte de la estructura del módulo. Además, el módulo se representa mediante un identificador simple. Un ejemplo es el módulo matemático de resta.
2. **Cohesión secuencial.** En este tipo de cohesión, el módulo realiza una serie de tareas que tiene que cumplir con lo siguiente: la salida de una instrucción debe de coincidir de manera estricta con la entrada de la siguiente instrucción hasta que termine su tarea.
3. **Cohesión comunicacional.** En el módulo conviven varias actividades paralelas que no siguen un orden específico y que comparten los mismos datos de entrada y salida. En este caso, se recomienda que el módulo se descomponga en otros más pequeños con cohesión funcional.
4. **Cohesión procedural.** Los elementos almacenados en el módulo realizan diferentes acciones que de manera general no se encuentran relacionadas. Además, es posible que no se relacionen tampoco los datos de salida con los de entrada.
5. **Cohesión temporal.** Las actividades que llevan a cabo ciertos elementos se relacionan con el momento en el que se llevan a cabo, por ejemplo, cuando se resetea el sistema. Puede que se dé el caso en el que no haya datos ni de entrada ni de salida.
6. **Cohesión lógica.** Los elementos están destinados a realizar actividades de la misma categoría general, pero la selección de la actividad específica se realiza desde fuera del módulo.
7. **Cohesión casual.** No hay ningún tipo de organización entre los elementos y se crea un caos a la hora de organizarse.

Aunque siempre hay una tendencia a pensar que cuanto mayor es la cohesión, mejor es el diseño del programa, hay que tener en cuenta que no todos los tipos de cohesión son deseables.

La siguiente escala muestra los diferentes tipos de cohesión y como afectan en el mantenimiento.

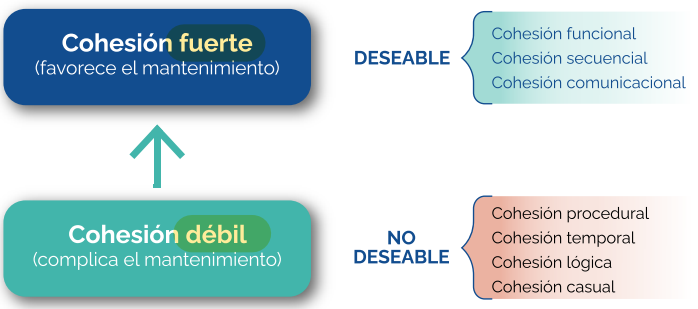


Imagen 7. Escala de cohesión por Stevens, Myers, Constantine y Yourdon.

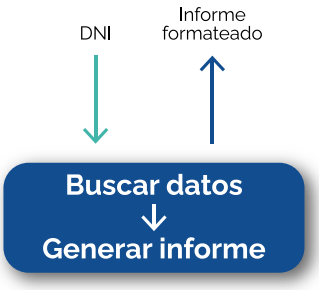


Imagen 2. Ejemplo de cohesión secuencial 1

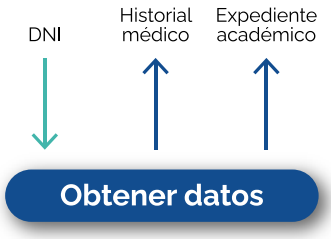


Imagen 3. Ejemplo de cohesión comunicacional 1

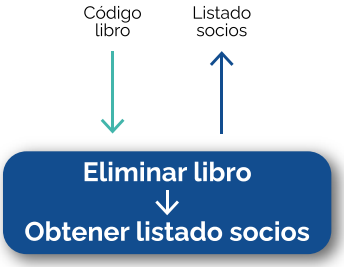


Imagen 4. Ejemplo de cohesión procedural 1

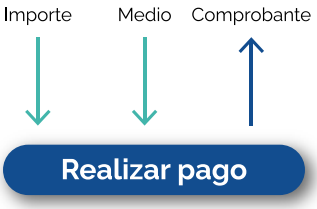


Imagen 5. Ejemplo de cohesión lógica. 1

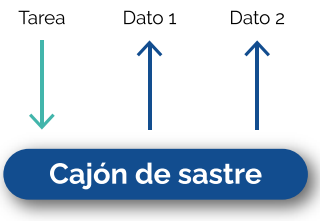


Imagen 6. Ejemplo de cohesión casual. 1

En el siguiente árbol de cohesión vemos todos los tipos de cohesión a los que se puede enfrentar un programador.

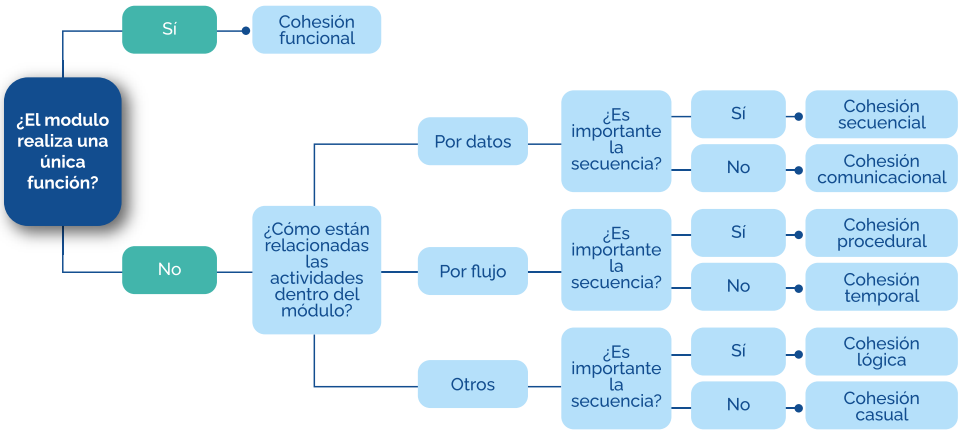


Imagen 8. Árbol de cohesión. 1

Acoplamiento

El término acoplamiento hace referencia al grado y forma de dependencia entre los módulos: cuanta menos información se ha compartido entre los diferentes módulos, menos acoplamiento tiene y por tanto mejor es el diseño.

Hay varios tipos de acoplamientos que se describen a continuación:

1. **Acoplamiento normal:** ocurre en los casos en que un módulo A llama a otro módulo B. Si se produce un intercambio de información será solo en lo referente a la llamada. En este caso y dependiendo del tipo de información, tenemos tres subtipos:
   a. **Acoplamiento normal por datos:** este se da cuando los parámetros que se intercambian son datos elementales.
   b. **Acoplamiento normal por marca o estampado:** este se da cuando los parámetros que se intercambian son un dato compuesto de datos de tipo básico como los anteriores.
   c. **Acoplamiento normal de control:** este se da cuando los parámetros se pasan de un módulo a otro intentando controlar la lógica de funcionamiento interna del último.
2. **Acoplamiento externo.** Se produce siempre que entre dos o más módulos se usen las mismas fuentes externas de datos.
3. **Acoplamiento global.** Se produce siempre que los módulos usen los mismos datos globales.
4. **Acoplamiento patológico o por contenido.** Se produce siempre que un módulo acceda a otro para leer o modificar los datos internos del último.

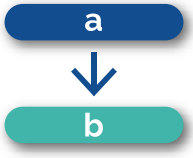


Imagen 9. Ejemplo de acoplamiento normal.

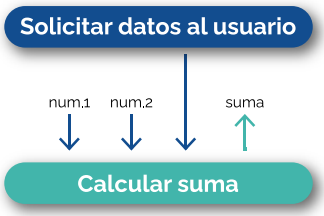


Imagen 10. Acoplamiento normal por datos.

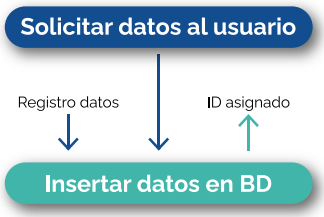


Imagen 11. Acoplamiento normal por marca o estampado.

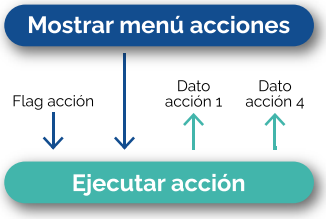


Imagen 12. Acoplamiento normal de control.

En el siguiente diagrama podemos ver todos los tipos de acoplamiento y de qué manera afecta al mantenimiento:

Imagen 13. Tipos de acoplamiento.

1.3. Pseudocódigo

El pseudocódigo es un lenguaje similar a un lenguaje natural cuyo objetivo es desarrollar algoritmos que puedan ser interpretados fácilmente por un programador, independientemente del lenguaje de programación del que provenga. Por sí solo, no es un lenguaje de programación, utiliza un conjunto limitado de expresiones que permiten las estructuras de control y módulos descritos en los paradigmas de programación anteriormente desarrollados.

Cuando escribimos con pseudocódigo, buscamos escribir los algoritmos que tienen una solución no infinita y que tienen su inicio en un único punto de partida. Esta estructura en pseudocódigo para desarrollar algoritmos tiene la intención de favorecernos cuando empecemos la transcripción del programa a un lenguaje de programación elegido.

1.3.1. Operadores, palabras reservadas y tipos de datos

Aunque no existe una regla rígida sobre cómo escribir programas en pseudocódigo, es recomendable seguir una serie de recomendaciones que permiten transcribir programas en el lenguaje de programación que se utilizará con la mayor facilidad.

Las siguientes tablas son un resumen de operadores, palabras reservadas y tipos de datos usados en el pseudocódigo.

Aritméticos		Relacionales (Usados para formar condiciones)		Lógicos (Usados para formar condiciones)	
+	Suma	=	Igual	and	Y lógico (conjunción)
-	Resta	<	Menor	or	O lógico (disyunción)
*	Multiplicación	<=	Menor o igual	no	Negación lógica
/	División real	>	Mayor	**Especiales**	
Div	División entera	>=	Mayor o igual	←	Asignación
Mod	Resto o módulo	<>	Distinto	//	Comentario
^	Potencia				

Imagen 14. Resumen de operadores. 1

Inicio	Si no	Otro	Para	En
Fin	Según	Mientras	Hasta	Procedimiento
Si	Hacer	Repetir	Incremento	Función
Entonces	Caso	Hasta que	Cada	Imprimir
Leer	Retornar			

Imagen 15. Palabras reservadas. 1

Carácter	Cadena	Entero	Real	Booleano

Imagen 16. Tipos de datos. 1

1.3.2. Estructuras de control

Ya hemos visto en el aparatado anterior que las estructuras de control que se tratan en la programación estructurada son las propias que usa el pseudocódigo. Por esto se usarán secuencias que representarán los tres tipos de estructuras de control: secuencial, alternativa e iterativa.

Estructural de control secuencial

Describen bloques de instrucciones que se ejecutan en orden de aparición, es decir, de manera secuencial, una seguida de la otra. Cada bloque se encuentra delimitado por las expresiones Inicio-Fin o contenidos en otras estructuras. Un ejemplo de estructura secuencial sería:

```
Inicio
        <instrucción1>
        …
        <instrucciónN>
Fin
```

Estructuras de control alternativa

Este tipo de estructuras funciona de la siguiente manera: se lanza una condición y dependiendo de si esta se cumple o no, se procederá con una serie u otra de instrucciones.

Hay diferentes subtipos dentro de este mismo tipo de estructuras de control:

1. **Alternativa simple.** Las instrucciones se ejecutarán siempre que se cumpla una condición que devolverá un valor booleano. Un ejemplo:

```
Si <condición> entonces
  <instrucción1>
        …
  <instrucciónX>
Fin Si
```

2. **Alternativa doble.** Igual que la anterior, pero en este caso se añade unas nuevas instrucciones que serán aplicadas en caso de que la condición no se cumpla. Un ejemplo:

```
Si <condición> entonces
  <instrucciones1>
Si no
  <instrucciones2>
Fin Si
```

3. **Alternativa múltiple.** Se puede realizar de dos maneras distintas, o bien con varios bloques de instrucciones que se ejecutarán si su caso en concreto es verdadero, como el ejemplo siguiente:

```
Según <expresión> hacer
  Caso <valor1>
        <instrucciones1>
  Caso <valor2>
        <instrucciones2>
        …
  Caso <valorX>
        <instruccionesX>
  Otro caso
        <instruccionesN>
Fin Según
```

O bien, el otro modo de que se cumplan estas expresiones es anidando tantas estructuras de alternativa doble como necesitemos. Un ejemplo:

```
  Si <condición1> entonces
        <instrucciones1>
  Si no
        Si <condición2> entonces
              <instrucciones2>
        Si no
              Si <condición3> entonces
                    <instrucciones3>
              Si no
                    <instrucciones4>
              Fin Si
        Fin Si
  Fin Si
```

Estructuras de control iterativa

La estructura iterativa crea un bucle en el que se repetirán las diferentes instrucciones mientras se cumpla la condición que se haya propuesto.

Hay varios subtipos de estructuras de control iterativas:

1. **Iteración con salida al principio (while):** Lo primero que se hace es evaluar la condición que se haya expuesto y si esta se cumple, comienza a ejecutar las instrucciones. Esta condición preasignada debe de cambiar el valor a medida que las instrucciones cambien, o si no, será un bucle infinito. También cabe la posibilidad de que no se cumpla en ningún momento la condición y no llegue a entrar en el bucle. Por ejemplo:

```
Mientras <condición> Hacer
  <instrucciones>
Fin Mientras
```

2. **Iteración con salida al final (repeat y do while):** Lo primero que hace es ejecutar las instrucciones que se le hayan asignado y después comprobará la condición para saber si seguir o no con el bucle. Hay dos tipos, en el primero, la expresión repeat, se ejecutan las condiciones hasta que llegue a un resultado en particular, es decir, hasta que la condición sea cierta, es en este momento cuando acabará el bucle. Su estructura es:

```
Repetir
  <instrucciones>
Hasta Que <condición>
```

El otro caso es el de la expresión do while, en el que se ejecutan las instrucciones mientras que la condición sea verdadera, en cuanto no se cumpla la condición, se dará por finalizado el bucle. La estructura que se sigue es:

```
  Hacer
        <instrucciones>
  Mientras <condición>
```

3. **Iteración con contador (for):** se hace uso de una variable que irá cambiando su valor hasta llegar a un cierto número en el que parará de ejecutar todo el tiempo el mismo conjunto de instrucciones. La variable puede tener un incremento positivo en la que su estructura es:

```
Entero i
Para i ← 1 Hasta N Incremento 1 Hacer
  <instrucciones>
Fin Para
```

Pero, también puede tener un incremento negativo con la siguiente estructura:

```
  entero i
  Para i ← N Hasta 1 Incremento -1 Hacer
        <instrucciones>
  Fin Para
```

4. **Iteración para cada (for each):** se va a ejecutar un conjunto de expresiones para un conjunto delimitado en una variable. Su estructura es:

```
entero i
Para Cada elemento En conjunto Hacer
  <instrucciones>
Fin Para Cada
```

Estructuras modulares

Tenemos también la posibilidad de escribir pseudocódigo para ejemplificar la descomposición modular de la que hablamos en el apartado anterior.

1. Procedimientos

Cada bloque de código contenido en un procedimiento ejecutará u conjunto de instrucciones cuando se invoca.

Puede recibir argumentos con los que trabajar cuando se llame si se han establecido los parámetros correspondientes, pero no devolverá un valor de salida en ninguna circunstancia.

Aquí hay un ejemplo de pseudocódigo para un procedimiento:

```
Procedimiento nombre (tipo parámetro1, tipo parámetro2, …)
        <instrucciones>
Fin Procedimiento
```

Vamos a mostrar ahora dos ejemplos de procedimientos, primero con Desarrollo de aplicaciones:

```
Procedimiento DesarrolloMultiplataforma ()
        imprimir (“Desarrollo de Aplicaciones”)
        imprimir (“----------------------------“)
Fin Procedimiento
Inicio
        DesarrolloMultiplataforma ()
Fin
```

Ahora vamos a ver el ejemplo con Aprendiendo a programar:

```
Procedimiento Programar (cadena asignatura)
        Imprimir (“Aprendiendo” + asignatura)
Fin Procedimiento
Inicio
        cadena asignatura
        imprimir (“Introduce que estás aprendiendo: “)
        leer (nombre)
        Programar (asignatura)
Fin
```

2. Funciones

Igual que nos pasaba anteriormente con los procedimientos, las funciones pueden recibir argumentos para trabajar con ellos siempre y cuando haya un parámetro definido previamente. La diferencia con los procedimientos es que las funciones si que retorna un valor de salida, por ejemplo, tenemos este pseudocódigo:

```
Función nombre (tipo parámetro1, tipo parámetro2, …)
: tipoRetorno
        <instrucciones>
        …
        Retornar X
Fin Función
```

1.4. Diagramas de flujo

Un diagrama de flujo, también llamado ordinograma o flujograma representa gráficamente un algoritmo o proceso concreto. Aunque su uso principal es en el ámbito informático, se puede usar en muchos sectores.

Este diagrama nos ayuda a entender mejor el algoritmo al aportarnos una descripción gráfica y mucho más visual.

1.4.1. Simbología

Para poder definir de manera correcta los diagramas de flujo, necesitaremos de unos símbolos específicos que cuando los unamos den un significado coherente que nos ayude a comprender el proceso. Ahora, vamos a explicar esta simbología, la cual también se encuentra en el Cuadro 4:

- Cada uno de los diagramas de flujo comienzan y terminan con terminal que se representa con un óvalo o elipse.
- Cuando se tiene una entrada de datos desde teclado, esta se representa como un trapecio rectángulo. Esto indica que el proceso se detiene a la espera de que el usuario aporte los datos.
- El paralelogramo nos describe la comunicación con los periféricos para la entrada o salida de información.
- Las operaciones que se vayan realizando deben de seguir un orden en concreto, y este orden se indica con flechas.
- Las decisiones se indican en un rombo donde pondremos tantas líneas de flujo como alternativas finales tengamos.
- Hay ocasiones en las que el algoritmo sea muy grande y no nos permite su diseño en un solo bloque, por lo que cuando esto pase, habrá que dividirlo. Estas divisiones se unen mediante conectores que nos indican el momento justo en el que se produce el salto en ambos bloques. Además, estos saltos se representan como una circunferencia para la misma página y con un pentágono irregular para páginas distintas.
- Las operaciones de cálculo interno del programa se describen con un rectángulo.
- Puede que haya que dividir también los procesos en los llamados subprocesos, en ese caso, cada subproceso se representará con un rectángulo con doble línea para indicar que eso se define en otro lugar.
- Las bases de datos para el intercambio y almacenamiento de la información tendrán un cilindro haciendo de símbolo.
- Por último, cada documento impreso usará un símbolo de media ola en un extremo y recto en el otro para su representación.

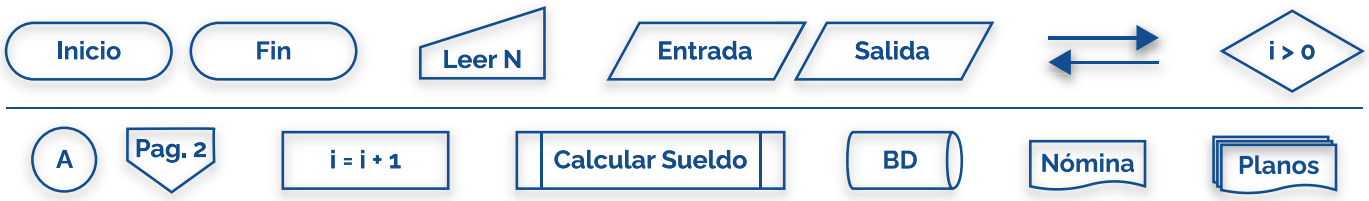


Imagen 17. Simbología propuesta por la American National Standars Institute (ANSI).

Aunque lógicamente estos no son todos los símbolos posibles, se han omitido todos los demás porque están obsoletos en gran parte.

1.4.2. Estructuras de control

Mediante diagramas de flujo, se pueden representar las estructuras de control de pseudocódigo descritas anteriormente.

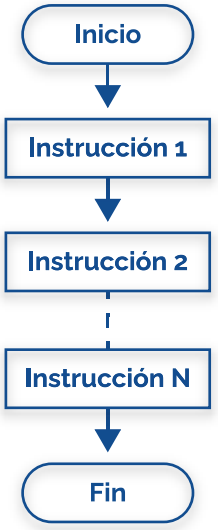


Imagen 18. Diagrama de flujo de Estructura de control secuencial.

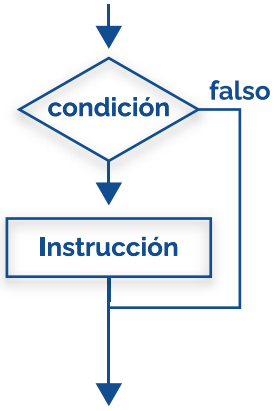


Imagen 19. Diagrama de flujo de Estructura de Control de alternativa simple.

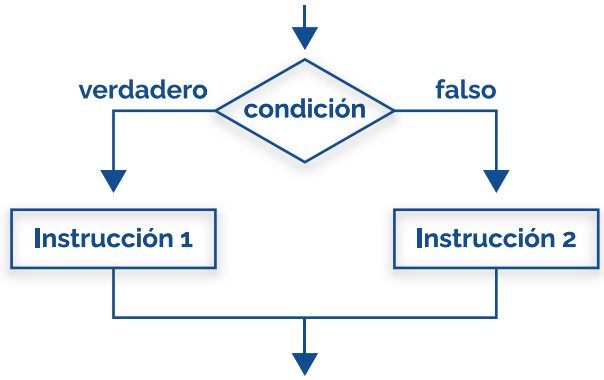


Imagen 20. Diagrama de flujo de Estructura de control con alternativa doble.

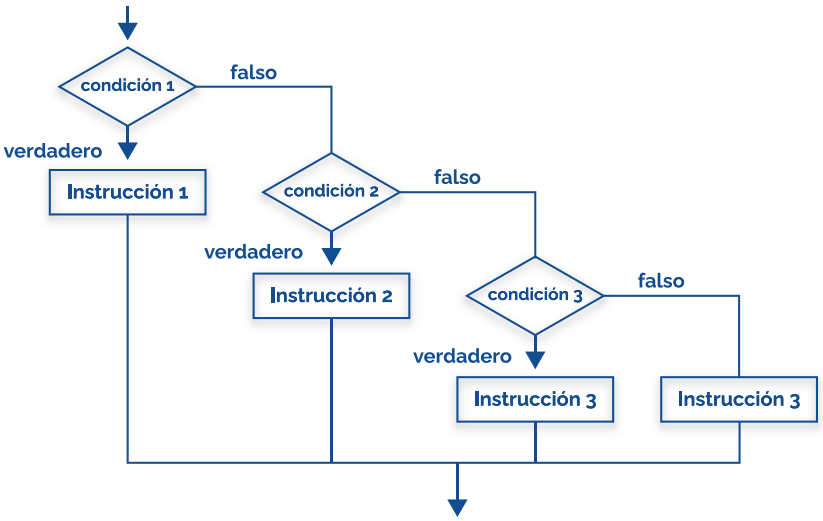


Imagen 21. Alternativa múltiple con simples anidadas.

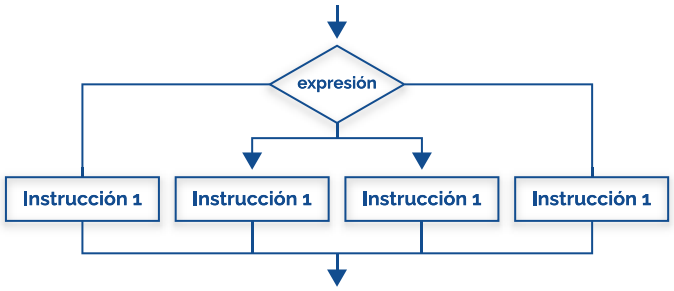


Imagen 22. Alternativa múltiple II.

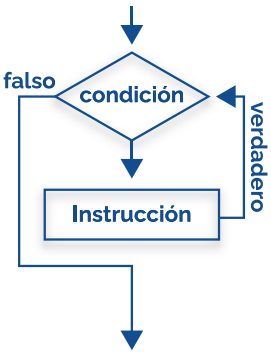


Imagen 23. Iterativa while. 1

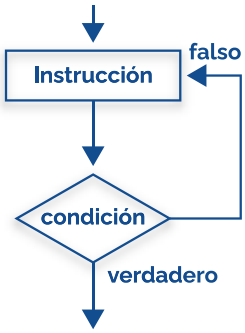


Imagen 24. Iterativa repeat.

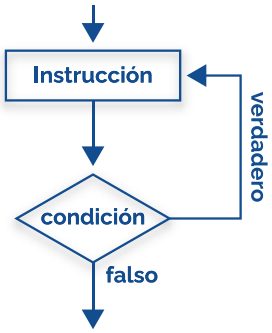


Imagen 25. Iterativa do while.

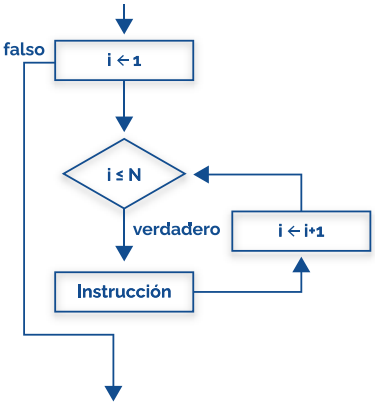


Imagen 26. Iterativa for incremental.

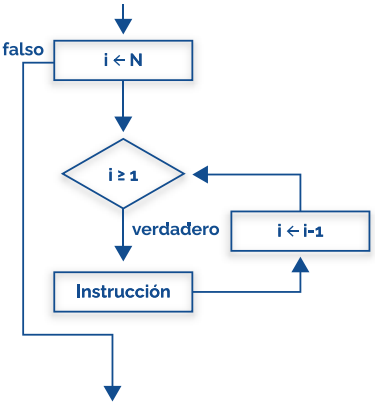


Imagen 27. Iterativa for decremental.

Cada uno de los procedimientos o funciones que hemos descrito en las estructuras modulares se describen con diagramas independientes, pero la llamada al subprograma siempre se hará desde el diagrama principal.

> Cuando invocamos a cierto procedimiento, la llamada se hace con el subproceso que lo alberga, representado con un rectángulo con lados dobles. Estos son dos ejemplos:

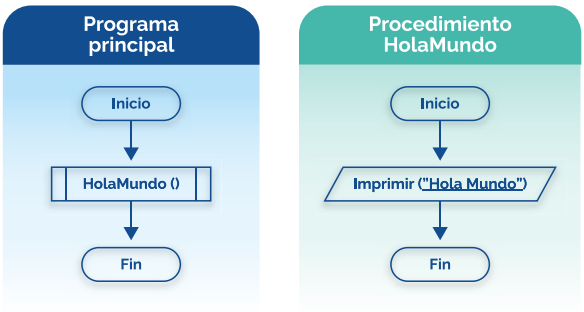


Imagen 28. Ejemplo HolaMundo.

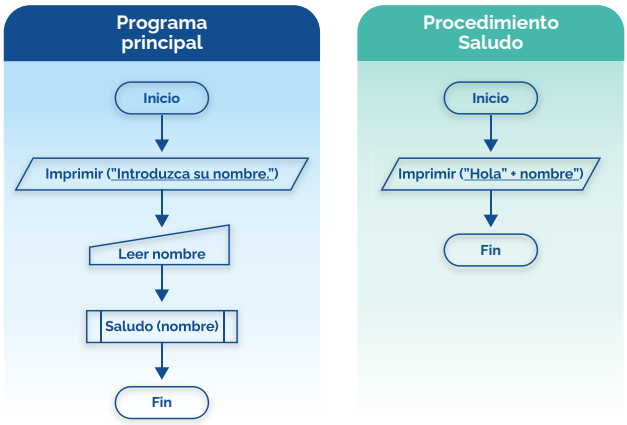


Imagen 29. Ejemplo Saludo.

> Por otra parte, si lo que queremos es llamar a la función, lo hacemos indicando la llamada en el proceso:

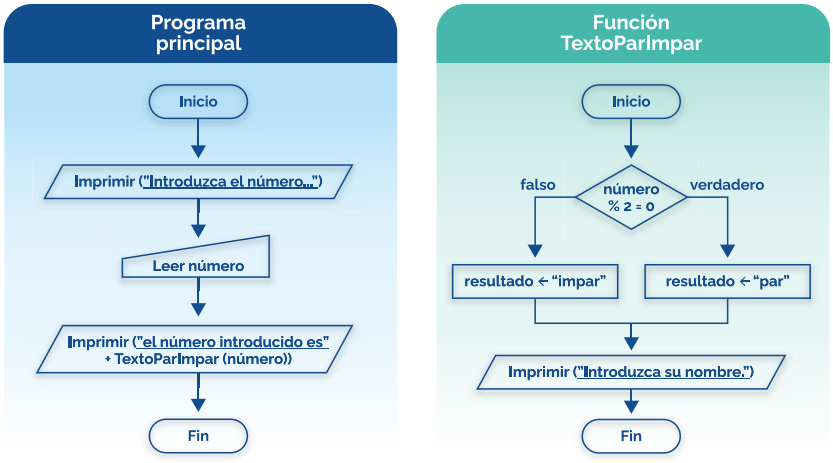


Imagen 30. Ejemplo de función.

www.universae.com