1 Metodología de la programación

1.1 LÓGICA DE PROGRAMACIÓN

Si bien la lógica puede ser definida como una parte de la filosofía que estudia las formas y principios generales que rigen el conocimiento y el pensamiento humano sin que intervengan los objetos, como un método o razonamiento en el que las ideas o la sucesión de los hechos se manifiestan o se desarrollan de forma coherente sin que haya contradicciones entre ellas o, como un modo particular de pensar, de ver las cosas, de razonar o, incluso, de actuar, siempre desde un punto de vista coherente y racional, la lógica de programación tiene más que ver con la forma de entender y crear cosas.

La lógica de programación es una técnica que se utiliza para crear una secuencia la cuál permita obtener un determinado objetivo. En otras, palabras, es la habilidad de organizar y planificar instrucciones o acciones con el objetivo de crear o implementar un algoritmo que permita alcanzar un objetivo deseable.

1.2 DESCRIPCIÓN Y UTILIZACIÓN DE OPERACIONES LÓGICAS

Las operaciones lógicas no son otra cosa que expresiones matemáticas cuyo objetivo es el de obtener una validación de verdadero o falso.

En general, este tipo de expresiones suelen utilizarse en las estructuras de control como pueda ser una condición de bifurcación.

Los elementos que componen estas expresiones, como se verá más adelante, normalmente serán variables, constantes, valores concretos de un tipo predefinido o, incluso, funciones que devuelven un valor determinado.

A continuación, se muestra un listado con las operaciones lógicas más frecuentemente utilizadas, independientemente dl lenguaje de programación usado.

Ejemplo	*Nombre*	*Resultado Verdadero*
Expr1 == Expr2	Iguales	Si el valor de Expr1 es igual al valor de Expr2
Expr1 === Expr2	Idénticas	Si el valor de Expr1 es igual al valor de Expr2 y ambas son del mismo tipo (es decir, enteros, cadenas, etc.)
Expr1 != Expr2	Diferentes	Si el valor de Expr1 es distinto al valor de Expr2
Expr1 <> Expr2		
Expr1 !== Expr2	No idénticas	Si el valor de Expr1 es distinto al valor de Expr2 o no son del mismo tipo
Expr1 > Expr2	Mayor que	Si el valor de Expr1 es menor al valor de Expr2
Expr1 < Expr2	Menor que	Si el valor de Expr1 es mayor al valor de Expr2
Expr1 >= Expr2	Mayor o Igual que	Si el valor de Expr1 es mayor o igual al valor de Expr2
Expr1 <= Expr2	Menor o Igual que	Si el valor de Expr1 es menor o igual al valor de Expr2

1.3 SECUENCIAS Y PARTES DE UN PROGRAMA

Si bien un programa es algo impreciso que se compone por un conjunto de acciones o instrucciones dispuestas de una forma determinada y que operan con, o manipulan objetos, las partes de un programa son algo bastante preciso que se suele describirse a partir de dos bloques muy bien diferenciados:

- **Declaraciones**: Bloque en el que se especifican o definen todos los objetos que utiliza el programa o aplicación. En general, aquí podemos encontrar variables, constantes, archivos, objetos personalizados.
- **Instrucciones**: Bloque en el que se especifican o definen las instrucciones o acciones que se han de ejecutar para obtener el resultado deseado. No obstante, este último bloque suele contener otras tres partes o secciones esenciales que, en general, suelen estar perfectamente diferenciadas.
  - **Entrada de datos**: Constituida por instrucciones que recogen datos de los dispositivos externos y los almacenan en la memoria central para, posteriormente, ser procesados.
  - **Proceso o algoritmo**: Constituida por instrucciones que manipulan los datos u objetos y regresan los resultados finales a la memoria central.
  - **Salida de resultados**: Constituida por instrucciones que recogen los datos o resultados finales de la memoria central y los muestran por, o envían a los dispositivos externos.

Cabe destacar que, a menudo, estas partes o secciones también suelen verse como una estructura compuesta por siete capas. Las cuatro primeras constituyen el bloque de declaraciones y, las tres últimas, constituyen el bloque de instrucciones.

Parte	*Descripción*
Cabecera	Contiene o define cosas como el nombre del programa o aplicación, los datos de entrada y datos de salida.
Funciones	Contiene todas las funciones o subrutinas que utiliza el programa.
Declaraciones	Contiene todos los objetos que utiliza el programa. Es decir, variables, constantes, objetos, tipos de datos personalizados.
Asignaciones	Contiene los valores iniciales de los identificadores o declaraciones previas.
Entrada	Contiene las instrucciones que permiten almacenar en memoria los valores de varios identificadores.
Control	Contiene las estructuras de control de flujo. Es decir, instrucciones condicionales, de repetición, etc.
Salida	Contiene las instrucciones que permiten devolver los resultados al usuario.

1.4 ORDINOGRAMAS

1.4.1 DESCRIPCIÓN DE UN ORDINOGRAMA

Un ordinograma, también denominado diagrama de flujo, se puede definir como un diseño esquemático que representa la secuencia de operaciones que se deben seguir para crear un algoritmo o programa determinado. Por desgracia, este tipo de diseños se van quedando, cada vez más, en el olvido y se recurre a otras técnicas, aunque esta sigue siendo una de las más eficientes.

Dependiendo del nivel de detalle o profundidad al que se desee llegar, los ordinogramas podrán representar desde un programa completo, hasta un subproceso, función o subrutina concreta.

1.4.2 ELEMENTOS DE UN ORDINOGRAMA

Todo ordinograma debe empezar por un símbolo de inicio, mostrar una secuencia de desarrollo que defina o describa el correcto funcionamiento de la función, subrutina o programa.

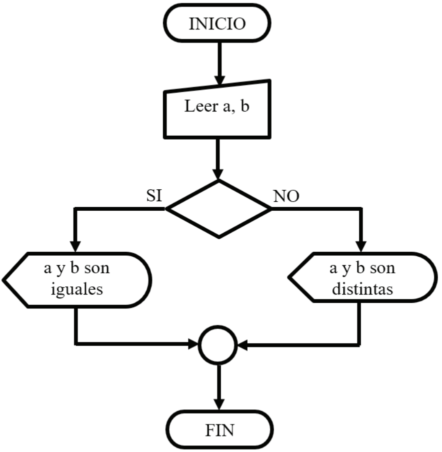
Símbolo	*Descripción*
	Símbolo que representa el inicio, fin o una parada indeterminada.
← ↑↓ →	Símbolos que representan la dirección del proceso o el flujo de datos.
	Símbolos que representan una conexión entre elementos.
	Símbolo que representa una entrada / salida genérica.
	Símbolo que representa una operación o proceso general con datos en memoria.
	Símbolo que representa un proceso de clasificación.

Símbolo	Descripción
	Símbolo que representa un proceso de manipulación.
	Símbolo que representa una subrutina que realiza una llama a un módulo del ordinograma o programa.
	Símbolo que representa una decisión con una entrada y dos posibles salidas. En general, la entrada se suele representar por una línea de entrada ubicada en la parte superior del mismo y, el resultado de la decisión, por dos líneas laterales ubicadas en los extremos derecho e izquierdo.
	Símbolo que representa una decisión con una entrada y múltiples salidas. En general, la entrada se suele representar por una línea de entrada ubicada en la parte superior del mismo y, el resultado de la decisión, por varias líneas de salida ubicadas en su parte inferior.
	Símbolo que representa un bucle o estructura de iteración.
	Símbolo que representa un conector que sirve para agrupar varias líneas que salen del mismo origen.
	Símbolo que representa entrada por teclado.
	Símbolo que representa una salida por pantalla.
	Símbolo que representa una salida por impresora.
	Símbolo que representa un almacenamiento de datos en disco.
	Símbolo que representan líneas de transmisión de datos.
	Símbolo que representa un comentario. El comentario en sí, estaría definido a la derecha del propio símbolo.

Cabe destacar que la creación de ordinogramas debe seguir unas reglas básicas. Por ejemplo:

- El símbolo de inicio sólo puede tener una línea de salida y no puede tener ninguna línea de entrada.
- El símbolo de fin puede tener varias líneas de entrada, pero no puede tener ninguna línea de salida.
- Los procesos pueden tener varias entradas de flujo, pero sólo una única salida.
- Todos los elementos que se conforman el ordinograma deben estar conectados por, al menos, una línea de flujo.
- No se pueden cruzar las líneas de flujo, entre otras razones, para evitar un mal entendimiento.

1.4.3 EJEMPLO DE ORDINOGRAMA



1.5 PSEUDOCÓDIGOS

1.5.1 DESCRIPCIÓN DE UN PSEUDOCÓDIGO

Un pseudocódigo, también denominado lenguaje descriptivo algorítmico, puede definirse como una descripción informal a alto nivel de cómo funciona un programa, función o subrutina.

El pseudocódigo suele ser una herramienta que se utiliza como paso intermedio antes de implementarlo en un lenguaje de programación “real”. Esto es así porque, aunque se trata como un lenguaje de programación más, en realidad no lo es. Es más bien una aproximación en términos humanos que se suele expresar en el idioma nativo que habla y escribe el autor.

Idea(s) → Pseudocódigo → Implementación

Una de las razones por las que el pseudocódigo es una buena herramienta es porque ayuda a los desarrolladores a abstraerse de la complejidad propia del lenguaje de programación y permite concentrarse en lo que de verdad importa, el problema a solucionar. Pongamos un ejemplo:

```
Inicio
    Solicitar primer operando y almacenar en A
    Solicitar segundo operando y almacenar en B
    Establecer C = A + B
    Escribir por pantalla "El resultado es: ", C
Fin
```

Otra cosa buena que tiene el pseudocódigo es que no atiende a una única sintaxis, por lo que es posible escribir un pseudocódigo de muy diversas maneras. Vamos, que en vez de utilizar esta sintaxis que acabamos de mostrar, podríamos haber utilizado la siguiente, que también sería válida:

```
Proceso Suma
    Escribir "Ingrese el primer operando"
    Leer A
    Escribir "Ingrese el segundo operando"
    Leer B

    Establecer C = A + B

    Escribir por pantalla "El resultado es: ", C
FinProceso Suma
```

El uso de pseudocódigo se ha extendido tanto que, incluso, se han llegado a crear intérpretes específicos, como si de PHP, JavaScript o cualquier otro lenguaje se tratase. Buen ejemplo de ello es PSeInt, una herramienta de escritorio que ayuda a los estudiantes y desarrolladores a centrarse en los conceptos esenciales de la algoritmia computacional, reduciendo las dificultades propias del lenguaje, como su sintaxis, y proporcionando un entorno de trabajo con múltiples ayudas y recursos didácticos.

Un ejemplo de PSeInt podría ser:

```
Proceso Suma
       Escribir "Ingrese el primer operando:"
       Leer A

       Escribir "Ingrese el segundo operando:"
       Leer B

       C <- A+B

       Escribir "El resultado es: ", C
   FinProceso
```

Como se puede apreciar en el ejemplo anterior, primero se le solicita al usuario una entrada de datos a través de la instrucción `Escribir`, y luego se lee dicho dato mediante la instrucción `Leer`. Posteriormente, calculamos la suma de A y B, su resultado lo guardamos en C mediante la el operador de asignación `<-` y, finalmente, se muestra el resultado, precedido de un mensaje aclaratorio a través de la instrucción `Escribir`.

1.5.2 CREACIÓN DEL PSEUDOCÓDIGO

Al contrario que los ordinogramas o diagramas de flujo, cuando se trata de crear pseudocódigos no hay unas reglas establecidas. Sin embargo, existen unas normas básicas y conocimientos mínimos que se deben tener.

Lo primero que se debe conocer es la estructura de un pseudocódigo. Estas son la cabecera, que contiene el identificador o nombre del programa, los tipos de datos, constantes y variables y, el cuerpo, que contiene tres secciones denominadas comúnmente inicio, instrucciones y fin.

Por ejemplo, en nuestro ejemplo anterior podríamos definirlo de la siguiente manera:

```
Algoritmo Sumar

       definir A, B como entero
   inicio
       escribir (“Ingrese el primer operando:”)
       leer A

       escribir (“Ingrese el segundo operando:”)
       leer B

       C <- A + B

       escribir (“El resultado es: “, C)
   fin
```

Si observamos detenidamente el código, veremos que la parte superior hasta la palabra clave **`inicio`**, es lo que denominaríamos la **`cabecera`** y, el resto, sería lo que denominamos el **`cuerpo`**.

Por tanto, para crear un pseudocódigo necesitamos saber algunas palabras clave y cómo definir variables, constantes, condiciones y posibles iteraciones.

1.5.2.1 Definición de variables y constantes

Las variables suelen representar los elementos, características o información relevante para el problema a solucionar. Por ejemplo, un algoritmo que requiera calcular el índice de masa corporal, necesitará una variable peso y una variable estatura.

Ahora bien, como hemos visto en el ejemplo anterior, las variables han de tener asignado un tipo, por lo que habrá de indicar de algún modo cuál es. Bien, pues, a continuación, mostramos una forma de definir o describir qué tipo de dato podrá contener cada variable.

Tipo	*Descripción*
entero	Indica que el valor puede contener cualquier número positivo o negativo, pero sin parte decimal.
real	Indica que el valor puede contener cualquier número positivo o negativo con o sin parte decimal.
cadena	Indica que el valor puede contener cualquier letra, número, carácter especial o símbolo, siempre y cuando esté definido entre comillas. Este tipo de dato.
carácter	Este tipo de dato puede estar junto al anterior o sustituirlo. La única diferencia es que, si está, habitualmente suele indicar el tamaño de caracteres entre paréntesis u otra forma similar.
lógico	Indica que el valor puede ser verdadero o falso.

Es muy importante que definamos bien los tipos de datos porque, no sólo nos ayudará a establecer una diferencia visual, sino que también asignará y limitará sus posibles valores y operaciones añadiendo restricciones.

Por ejemplo, si definimos una variable como entera y otra como cadena, cuando queramos almacenar en ellas el valor diez, lo deberemos hacer como 10 o “10”, respectivamente.

Aunque para nosotros 10 y “10” puedan parecer lo mismo, la realidad es que no tienen nada que ver. No sólo porque la primera hace referencia a un valor de tipo numérico y, la segunda, a un tipo cadena, sino porque, la primera, nos permitirá realizar posibles operaciones matemáticas como sumar o multiplicar y, la segunda, no nos permitirá nada más que una posible concatenación.

1.5.2.2 Definición de condiciones

Existen cuatro posibles tipos de condiciones:

Simple

La condición simple es aquella que evalúa una expresión y da como resultado un valor booleano o lógico. Las instrucciones contenidas dentro de esta condición se ejecutarán sí, y sólo sí, su resultado es verdadero.

```
Si condición entonces
      instrucciones
   Fin si
```

Doble

Este tipo de condición es prácticamente igual que la simple, con la diferencia de que, si no se cumple la condición, se ejecutará el bloque de no cumplimiento establecido por “si no”.

```
Si condición entonces
      Instrucciones1
   Si no entonces
      Instrucciones2
   Fin si
```

Múltiple

Este tipo de condición permite unificar varias condiciones, las cuales pueden ser simples y/o una doble. Si no se cumple la primera condición, pasará a la segunda, y si no se cumple, a la tercera. Y esto será así, sucesivamente, hasta completar las posibles condiciones.

```
Si condición1 entonces
      Instrucciones1
   Si no si condición2 entonces
      Instrucciones2
   Si no si condición3 entonces
      Instrucciones3
   ...
   Si no entonces
      instruccionesN
   Fin si
```

Caso Indicador

Este tipo de condición permite tomar un indicador y compararlo con cada caso definido a continuación. Si en alguno de los casos se produce una coincidencia, se ejecutarán las instrucciones que le correspondan.

```
Seleccionar indicador
   Caso valor1
      Instrucciones1
   Caso valor2
      Instrucciones2
   ...
   En cualquier otro caso
      instruccionesN
   Fin Seleccionar
```

1.5.2.3 Definición de iteraciones

Las iteraciones, también denominadas bucles, lazos o ciclos, son estructuras de control con carácter repetitivo, es decir, son estructuras que contienen un conjunto de instrucciones que poseen la cualidad de poder ser ejecutadas 1 o N veces.

Existen tres posibles tipos de iteraciones, ciclos, lazos o bucles:

Mientras

La estructura de control repetitiva `mientras` es aquella que, como su propio nombre indica, se ejecuta mientras se cumpla la condición. Por tanto, dependiendo de su evaluación, se podrá ejecutar ninguna, una o múltiples veces.

```
Mientras condición hacer
      instrucciones
   Fin mientras
```

Repetir

La estructura de control repetitiva **repetir** es aquella que se ejecutará hasta que se no cumpla la condición. Por tanto, dependiendo de su evaluación, se podrá ejecutar una o múltiples veces.

```
Repetir
      instrucciones
   Hasta que condición
```

Para

La estructura de control repetitiva **para** es aquella que se ejecuta un número conocido de veces. El número de iteraciones vendrá predefinido por una variable que se verá incrementada en cada iteración, pero, recalcando que este incremento no tiene por qué ser de uno en uno.

```
Para i <- x hasta n hacer
      instrucciones
   Fin para
```

1.5.3 EJEMPLOS DE PSEUDOCÓDIGO

Escribir un pseudocódigo que muestre la tabla de multiplicar del número que nos indique un usuario a través del teclado.

```
Algoritmo TablaMultiplicar

       definir i, valor, total como real

   inicio
       escribir (“Ingrese un número:”)
       leer valor

       Para i <- 1 hasta 10 hacer
           total = valor * i
           escribir (valor, “x”, i, “ es: “, total)
       Fin Para
   fin
```

Escribir un pseudocódigo que muestre la tabla de multiplicar del número que nos indique un usuario a través del teclado.

```
Algoritmo TablaMultiplicar


        definir peso, estatura, imc como real
        definir masa como cadena

    inicio
        escribir ("Ingrese su peso en kilogramos:")
        leer peso

        escribir ("Ingrese su estatura en metros:")
        leer estatura

        imc = peso / (estatura * estatura)

        si imc <= 18.4 entonces
            masa = "peso bajo"
        si no si imc >= 18.5 y imc <= 24.9 entonces
            masa = "peso normal"
        si no si imc >= 25 y imc <= 29.9 entonces
            masa = "sobrepeso"
        si no entonces
            masa = "obesidad"
        fin si

        escribir ("Su índice de masa corporal es ", imc, ", lo que indica que tiene ", masa)
    fin
```

1.6 OBJETOS

1.6.1 DESCRIPCIÓN DE UN OBJETO

Cuando se habla dentro del contexto de la programación orientada a objetos (POO), un objeto es todo aquello que posee uno o varios estados y uno o varios comportamientos.

Los objetos pueden ser creados mediante la instanciación de clases (es decir, una plantilla para la creación de objetos de datos según un modelo predefinido), como ocurre en todos los lenguajes de programación orientada a objetos, o a través de la escritura directa de código y replicación, como ocurre en la programación basada en prototipos.

Entre las muchas ventajas de usar objetos, posiblemente una de las que más destaque sea la herencia. La herencia es la razón por la cual los diseñadores o autores pueden crear nuevos objetos partiendo de un objeto o de una jerarquía de objetos preexistente ya comprobados, por lo que evitamos rediseñar, modificar y verificar las partes ya implementadas.

La herencia facilita la creación de objetos a partir de otros ya existentes e implica que, un objeto interno o subclase, obtiene todos los comportamientos y estados del objeto padre o superclase.

1.6.2 FUNCIONES DE LOS OBJETOS

La función de los objetos a menudo se identifica con la posibilidad de definir las características y funcionalidades que poseen los objetos de un mismo tipo. Sin embargo, su uso y potencial va más allá porque permiten cosas como la reutilización de código ya que facilita su migración y uso en múltiples contextos y disminuye los tiempos de desarrollo.

1.6.3 COMPORTAMIENTOS Y ESTADOS DE LOS OBJETOS

Los comportamientos de un objeto están directamente relacionados con su funcionalidad y determinan las operaciones que este puede realizar, o a las que puede responder ante mensajes enviados por otros objetos. La funcionalidad de un objeto suele identificarse con las acciones o métodos que puede hacer. Esto es, por ejemplo, calcular el área, volumen, índice de masa corporal, etc.

Los estados de un objeto se refieren al conjunto de atributos y sus valores en un instante de tiempo dado y suelen identificarse con las propiedades o atributos que lo caracterizan. Esto es, por ejemplo, su tamaño, color, textura, forma, etcétera.

Por tanto, los comportamientos manipulan los estados, ya sea como consulta, o como modificación o borrado. Cuando un comportamiento o método modifica uno o varios de sus estados se suele decir que tiene un “efecto colateral”.

Para intentar ser más claros con esto de los estados y comportamientos, veamos un ejemplo:

```
Enumerado Carburante
    diesel, super, sin_plomo, otro
   Fin Enumerado
Objeto Coche

       definir marca, modelo como cadena
       definir vel_max, potencia como entero
       definir tipo_carburante como Carburante

        público:
           función arrancar(){ ... }
           función parar(){ ... }
           función acelerar(){ ... }
           función frenar(){ ... }
   Fin Objeto
```

Como se puede observar en el ejemplo anterior hay dos zonas muy bien definidas. La parte de las **variables** equivale a lo que aquí denominamos estados, porque la marca, modelo, velocidad máxima, potencia y tipo de carburante son atributos o propiedades inherentes al propio objeto.

Las parte que hemos denominado **público**, hace referencia a la sección de comportamientos, es decir, los métodos o acciones que permite dicho objeto y que requieren o manipulan los estados.

1.6.4 EJEMPLOS DE CÓDIGOS EN DIFERENTES LENGUAJES

Objeto coche en Java

```
public class Coche {
private String marca;
private String modelo;
private String tipo_carburante;
private float vel_max;
private float potencia;

public Coche(String marca, String modelo) {
   this.marca = marca;
   this.modelo = modelo;
}

public void arrancar() {
   // instrucciones
}

public void parar() {
   // instrucciones
}

public void acelerar() {
   // instrucciones
}

public void frenar() {
   // instrucciones
}
}
```

Objeto coche en C++

```
class Coche {
public:
   Coche();
   Coche(char *marca, char *modelo);
   ~Coche();
   void arrancar();
   void parar();
   void acelerar();
   void frenar();
private:
   char *marca;
   char *modelo;
   float vel_max;
   int potencia;
   char *tipo_carburante;
};
```

Objeto coche en JavaScript

```
class Coche {
   vel_max = 0;
potencia = 0;
tipo_carburante = “”;

constructor(marca, modelo) {
   this.marca = marca;
   this.modelo = modelo;
}

get arrancar(){
   // incstrucciones
}
get parar(){
   // incstrucciones
}
get acelerar(){
   // incstrucciones
}
get frenar(){
   // incstrucciones
}
}
```