

DESARROLLO DE APLICACIONES MULTIPLATAFORMA

PROGRAMACIÓN

TEMA 1

INTRODUCCIÓN A LA PROGRAMACIÓN

CONTENIDOS

- 1.1. Introducción.
- 1.2. Conceptos básicos.
 - 1.2.1. Algoritmo vs programa.
 - 1.2.2. Características de un programa.
 - 1.2.3. Lenguaje de programación.
 - 1.2.4. Código.
 - 1.2.5. Palabras reservadas del lenguaje.
- 1.3. Diseño de un programa.
 - 1.3.1. Fase de análisis del problema.
 - 1.3.2. Fase de diseño del algoritmo.
 - 1.3.3. Fase de programación (codificación).
 - 1.3.4. Fase de pruebas y depuración.
 - 1.3.5. Fase de documentación.
 - 1.3.6. Ejemplo.
- 1.4. Diagramas de flujo.
- 1.5. Pseudocódigo.
- 1.6. Proceso de compilación.
- 1.7. Entornos de desarrollo integrado de software (IDE).

1.1. INTRODUCCIÓN.

- Cuando se comienza a estudiar informática se inicia analizando qué es un sistema informático. Un sistema informático es un conjunto formado por tres partes:
 - Hardware: Es la parte tangible del sistema, parte física. Se denomina hardware a todos los componentes físicos que forman el sistema informático, todas aquellas piezas que forman “la torre” del ordenador junto a la pantalla, impresora,...
 - Software: Es la parte no tangible. La forman todas aquellas aplicaciones o programas instalados en el equipo. El software se clasifica en software de base (sistema operativo) y software de aplicación (aplicaciones diseñadas para realizar una operación concreta y que pueden ser usadas sobre un software de base concreto).
 - Personas: Es una parte importante del sistema ya que es quien usa el mismo. Sin la iteración de las personas de nada serviría que tuviéramos un hardware y un software.

1.2. CONCEPTOS BÁSICOS.

1.2.1. ALGORITMO VS PROGRAMA.

- Un algoritmo es un conjunto ordenado y finito de operaciones que permite hallar la solución de un problema.
- Ahora bien, cuando un problema debe ser resuelto por un ordenador, el algoritmo se convierte en lo que se denomina programa. Seguimos usando una secuencia ordenada y finita de pasos pero escrita en un lenguaje que la máquina es capaz de entender.
- La unidad mínima de almacenamiento de un ordenador se denomina bit. Un bit puede adquirir los valores 1 o 0 (sistema de numeración binario). Un conjunto de bits representa tanto números como letras. Al conjunto de 8 bits se le denomina byte.

1.2. CONCEPTOS BÁSICOS.

1.2.1. ALGORITMO VS PROGRAMA.

- Debido a que cada vez es mayor la información que se almacena en un equipo no usamos simplemente bytes para definir esta, debemos utilizar los siguientes prefijos para representarla, de forma que:

1 byte	8 bits
1 KB (Kilobyte)	$2^{10} = 1024$ bytes
1 MB (Megabyte)	$2^{20} = 1024$ KB
1 GB (Gigabyte)	$2^{30} = 1024$ MB
1 TB (Terabyte)	$2^{40} = 1024$ GB
1 PB (Petabyte)	$2^{50} = 1024$ TB
1 EB (Exabyte)	$2^{60} = 1024$ PB
1 ZB (Zettabyte)	$2^{70} = 1024$ EB
1 YB (Yottabyte)	$2^{80} = 1024$ ZB

1.2. CONCEPTOS BÁSICOS.

1.2.2. CARACTERÍSTICAS DE UN PROGRAMA.

- Un algoritmo, posteriormente convertido en programa, debe cumplir una serie de características:
 - Debe ser finito. Estar formado por un conjunto de líneas/instrucciones de forma que en algún punto ve alcanzado su fin.
 - Debe ser legible. Es importante crear códigos “limpios” y fáciles de leer, con tabulaciones y espacios que diferencien las partes del programa. Si desarrollamos códigos bien estructurados nos será más fácil la corrección de errores y modificación del mismo.
 - Debe ser fácilmente modificable, es decir, debe estar diseñado de forma que debe ser sencillo el proceso de actualización o modificación del mismo ante las nuevas necesidades del usuario final.

1.2. CONCEPTOS BÁSICOS.

1.2.2. CARACTERÍSTICAS DE UN PROGRAMA.

- Debe ser eficiente. El usuario acabará usando programas que solucionen sus problemas de la mejor y más rápida forma posible, así que debemos crear programas que ocupen poco espacio en memoria y se ejecuten rápidamente.
- Deben ser modulares. Esta característica ayuda a que el programa sea más legible y fácil de entender. Debemos perseguir realizar algoritmos que se encuentren divididos en “subalgoritmos” de forma que se disponga de un grupo principal desde el que llamaremos al resto. Al usar subprogramas además incitamos la reutilización de código y evitamos la repetición de este.
- Debe estar estructurado. Esta característica engloba de alguna forma las anteriores, ya que un programa estructurado será fácil de leer, de modificar y estará compuesto de subprogramas que permitirán la reutilización de código.

1.2. CONCEPTOS BÁSICOS.

1.2.2. CARACTERÍSTICAS DE UN PROGRAMA.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace HolaMundo{class Program{
    static void Main(string[] args){Console.WriteLine("Hola mundo");Console.ReadLine();
    }
}}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace HolaMundo{
    class Program{
        static void Main(string[] args){
            Console.WriteLine("Hola mundo");
            Console.ReadLine();
        }
    }
}
```

1.2. CONCEPTOS BÁSICOS.

1.2.3. LENGUAJE DE PROGRAMACIÓN.

- Lenguaje natural.

- El ser humano se comunica a través del uso del lenguaje, es capaz de generar una serie de sonidos que forman lo que conocemos como fonemas, pasadas al papel son las letras de nuestro alfabeto, que combinadas constituyen las sílabas, palabras y frases:
- *Comunicación hombre <-> hombre.*

- Lenguaje máquina:

- Un ordenador comunica sus componentes hardware mediante la emisión de señales eléctricas, pulsos que vienen a formar los valores 0 (no se aplica voltaje) o 1 (si se aplica voltaje) formando combinaciones de estos:
- *Comunicación máquina <-> máquina (componentes hardware).*

- Lenguaje de programación:

- Es aquel por el que el ordenador es capaz de entender qué está diciéndole el ser humano que haga.
- *Comunicación hombre <-> ordenador.*

1.2. CONCEPTOS BÁSICOS.

1.2.3. LENGUAJE DE PROGRAMACIÓN.

- Clasificación en función de la cercanía a la máquina:
 - Existen lenguajes de programación bastante cercanos al lenguaje máquina, que usan instrucciones similares a las que entiende la CPU de forma directa, mientras que otros están formados por palabras reservadas más cercanas al lenguaje natural.
 - Lenguajes de bajo nivel. Es el más cercano al lenguaje máquina. Está constituido por palabras reservadas difíciles de recordar y la sintaxis de las instrucciones es bastante compleja ya que se trabaja a nivel de registro. Ej.: *el lenguaje ensamblador*.
 - Lenguajes de nivel medio. Se acercan un poco más al lenguaje natural introduciendo palabras reservadas similares a las que usamos habitualmente pero que siguen proporcionando órdenes para trabajar a nivel de bit. Ej.: *el lenguaje de programación C*.
 - Lenguajes de alto nivel. Parecidos al lenguaje humano tal que su forma de proceder, su sintaxis, etc., es más similar al lenguaje natural. Abstira al programador del funcionamiento interno de la máquina.

1.2. CONCEPTOS BÁSICOS.

1.2.3. LENGUAJE DE PROGRAMACIÓN.

- Clasificación en función del propósito del lenguaje:
 - En función del tipo de programas que son capaces de generar tenemos:
 - Lenguajes de propósito general. Son lenguajes diseñados para realizar cualquier tipo de programa, desde software base a software de aplicación.
 - Lenguajes de propósito específico. Son lenguajes que se diseñan para actuar en una determinada área y generar programas con una finalidad determinada. Estos lenguajes dispondrán de palabras reservadas y funciones propias para la explotación del área de trabajo concretas.

1.2. CONCEPTOS BÁSICOS.

1.2.3. LENGUAJE DE PROGRAMACIÓN.

- Clasificación en función de la evolución histórica:

- A lo largo de los años las necesidades de los usuarios que utilizan los sistemas informáticos y el hardware del mismo, han ido evolucionando y cambiando. Los ordenadores de hace 20 años no poseían elementos físicos con características demasiado sofisticadas de forma que solo podíamos ejecutar programas en modo texto, en cambio hoy día estamos acostumbrados a usar entornos visuales:
- Primera generación. La forman los lenguajes máquina y lenguajes ensamblador. Años 40 y 50 donde se usaban tarjetas perforadoras, el programador debía codificar en ellas las combinaciones de ceros y unos que indicaban el tipo de cálculo a realizar. Se comienzan a usar lenguajes ensamblador de forma que no se tenía que especificar cadenas binarias, ya que se crearon una serie de sentencias que representaban los cálculos, pero seguían siendo instrucciones complicadas a nivel de registro y era necesario que conocieráramos el juego de instrucciones del microprocesador.

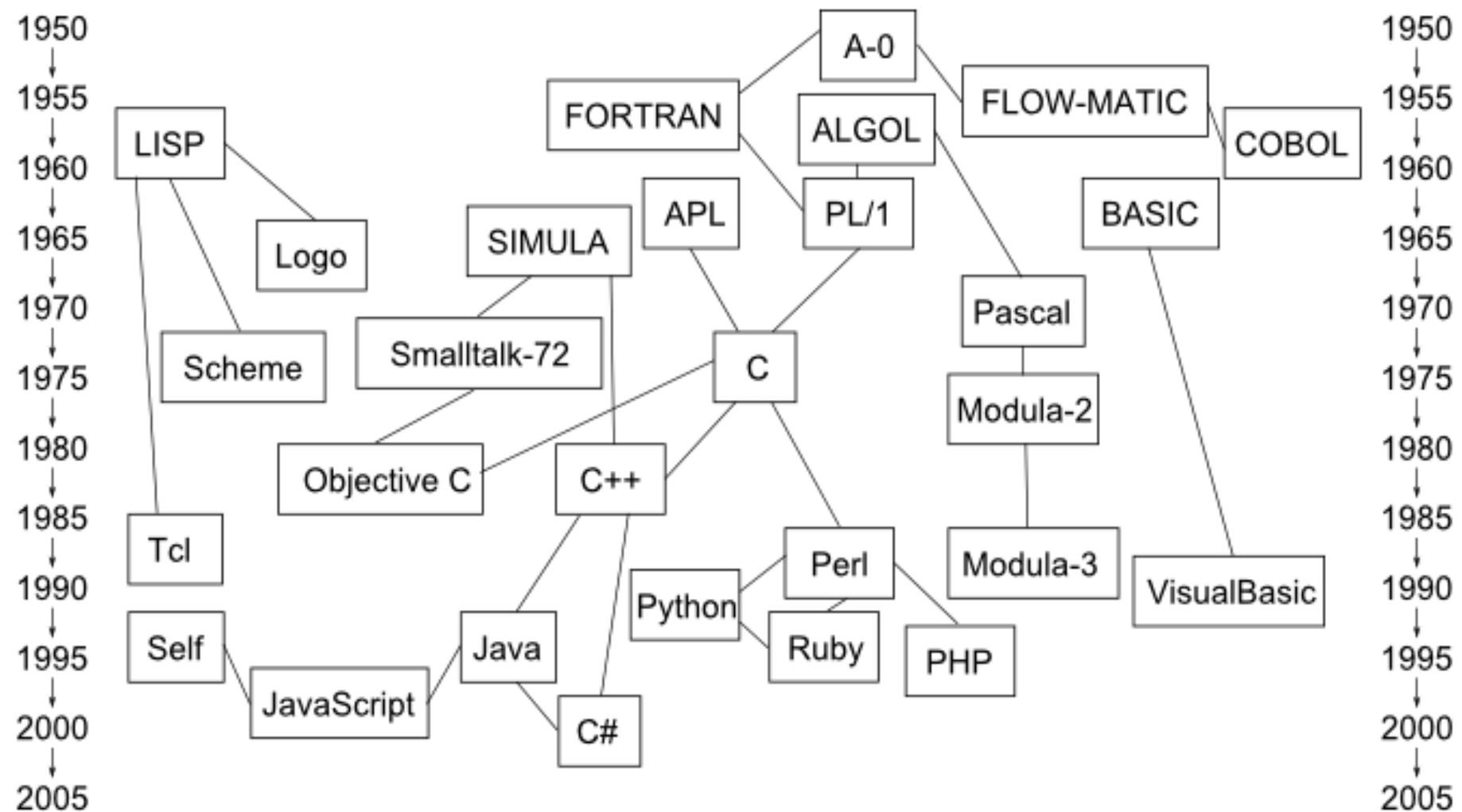
1.2. CONCEPTOS BÁSICOS.

1.2.3. LENGUAJE DE PROGRAMACIÓN.

- Segunda generación. Aparecen lenguajes de nivel medio. Se comienza a desarrollar el primer compilador de FORTRAN. Esta etapa perdura aproximadamente desde 1958 hasta 1970.
- Tercera generación. Aparece la programación estructurada, conceptos como subprogramas, variables locales o estructuras de datos dinámicas. Ej.: *Pascal, Modula o C*.
- Cuarta generación. Aparecen lenguajes desarrollados para crear un determinado tipo de software, se configuran para tareas muy concretas.
- Quinta generación. Generación asociada a lenguajes diseñados para abordar la inteligencia artificial. Este tipo de lenguajes trabajan con una serie de reglas y hechos y en función de estos deben concretar si una premisa conclusión es cierta o no. Ej.: *Prolog*. Además comienza a ponerse en práctica la programación orientada a objetos.
- Generación visual. Nace a principios de la década de los 90 debido a la necesidad de usar interfaces cada vez más amigables y fáciles.

1.2. CONCEPTOS BÁSICOS.

1.2.3. LENGUAJE DE PROGRAMACIÓN.



1.2. CONCEPTOS BÁSICOS.

1.2.3. LENGUAJE DE PROGRAMACIÓN.

- Clasificación en función de la forma de ejecución:

- Después de escribir un programa, el nuevo software debe ser ejecutado, ahora bien, la ejecución se puede realizar en función de si todo el código es correcto o bien, si una instrucción es correcta.
- Lenguajes compilados. Son aquellos que realizan un análisis minucioso de las líneas de nuestro código en búsqueda de posibles fallos léxicos, sintácticos o semánticos. Si nuestro código no tiene errores genera un fichero llamado *código objeto*. A partir de este nuevo código usará un programa llamado *enlazador o linker* que se encargará de añadir *librerías* u otro tipo de software necesario para así obtener el código o archivo ejecutable. Hasta que nuestro *código fuente* no esté libre de fallos no se ejecutará nuestro programa.
- Lenguajes interpretados. Son aquellos que ejecutan línea a línea el código creado. En caso de que exista alguna instrucción con fallos esta no se ejecutará, pero seguirá el proceso con el resto de líneas a no ser que la línea que falla sea una parte importante de nuestro programa.

1.2. CONCEPTOS BÁSICOS.

1.2.3. LENGUAJE DE PROGRAMACIÓN.

- Clasificación en función de cómo afrontan las tareas a realizar:
 - Lenguajes imperativos. Son aquellos en los que se escribe línea a línea qué se debe hacer, de forma que se indica una secuencia de pasos a realizar para llegar a resolver el problema en cuestión que dio pie a la realización del software. Tanto *C#* como *Java* son lenguajes imperativos.
 - Lenguajes declarativos. Son aquellos que incluyen una serie de premisas o conjunto de condiciones finitos y la conclusión a la que se debe llegar. En ellos no se indica de forma expresa los pasos a seguir para alcanzar la solución, solo se plantean una serie de reglas y en función de estas se van describiendo los pasos. Se usan normalmente en inteligencia artificial.

1.2. CONCEPTOS BÁSICOS.

1.2.3. LENGUAJE DE PROGRAMACIÓN.

- Clasificación en función del estilo de programación empleado:
 - A la hora de crear un programa podemos optar por escribir todo el código, línea a línea, como un solo bloque, realizar pequeños bloques incluidos en el programa principal a los que se hace referencia desde este o bien crear ficheros independientes con subprogramas a los que accederemos según nuestra necesidades. Así, encontramos:
 - Lenguajes de programación estructurada. El código se encuentra tal y como se haya estructurado en un conjunto de funciones. Suelen poseer una función principal que es llamada en el momento de ejecución, haciendo uso del resto según decida el programador. Podemos hablar de *programación modular* en los casos en los que se usan otros ficheros, llamados *módulos*, en los que se agrupan funciones diseñadas para trabajar con el mismo tipo de datos o están relacionadas según algún criterio establecido por la persona que crea la aplicación.

1.2. CONCEPTOS BÁSICOS.

1.2.3. LENGUAJE DE PROGRAMACIÓN.

- Lenguajes de programación orientada a objetos. Se usan elementos denominados *clases* de forma que se pretende con ellos reflejar al máximo posible la realidad que rodea al software. Así, si en la programación estructurada se agrupan en funciones, líneas de código según tareas o acciones a realizar, en la programación orientada a objetos la agrupación se realiza en función del *objeto* que se quiere plasmar en el programa.
- Ejemplo. Si queremos realizar un programa que gestione un concesionario de vehículos, programaremos una clase que llamaremos *vehículo*, en ella (en este conjunto de líneas) incluiremos todo lo que rodea a un coche, desde sus características a las funciones que pueden desarrollarse sobre él.

1.2. CONCEPTOS BÁSICOS.

1.2.3. LENGUAJE DE PROGRAMACIÓN.

- Clasificación en función de la capacidad de generar concurrencia:
 - La concurrencia es la ejecución de varios procesos a la vez. Generalmente usamos de forma errónea los términos programa y proceso:
 - Programa. Es el código que hemos generado, que es correcto y está listo para ejecutarse pero aún no está en ejecución.
 - Proceso. Una vez damos la orden de ejecutar nuestro programa, este se coloca en memoria y es decodificado instrucción a instrucción por el procesador, a esto se le denomina *proceso*.
 - Lenguajes de programación concurrente. Proporcionan los mecanismos necesarios para generar señales, tuberías, semáforos, etc., que permiten la ejecución concurrente.
 - Lenguajes de programación no concurrente. No aportan ningún tipo de dato o función para producir concurrencia.

1.2. CONCEPTOS BÁSICOS.

1.2.3. LENGUAJE DE PROGRAMACIÓN.

- Clasificación en función de la interactividad:

- Lenguajes de programación orientados a sucesos o eventos. Son aquellos que permiten la iteración continua del usuario con el software de forma que realizará una tarea u otra según el tipo de acción que el usuario realice.
- Lenguajes de programación no orientados a sucesos o eventos. En este caso, el programa seguirá una secuencia de acciones sin dar opción al usuario a modificar esta.

- Clasificación en función de si son o no visuales:

- Lenguajes de programación no visuales. Generan programas en modo texto. Se usan por ej., en comandos para configuración y administración de equipos.
- Lenguajes de programación visuales. Generan programas a los que se asocia un entorno visual. Estos lenguajes usan entornos de desarrollo que proporcionan las herramientas necesarias para la creación de paneles, botones, cuadros de texto, etc., de forma que el programador se evite la programación de estos componentes.

1.2. CONCEPTOS BÁSICOS.

1.2.4. CÓDIGO.

- Según la Real Academia de la Lengua se define código como la combinación de signos que tiene un determinado valor dentro de un sistema establecido. El código de un programa está formado por signos (elementos del lenguaje, elementos generados por el usuario, signos matemáticos, etc.) que tienen valor dentro de un sistema establecido, es decir, en las especificaciones de un lenguaje de programación concreto.
- Comúnmente se define código (o *código fuente*) como el conjunto o subconjunto de líneas que forman un programa. Estas líneas se configurarán con una estructura concreta y estarán formadas de unos u otros elementos en función del lenguaje de programación que estemos usando.
- Además, en los lenguajes de programación compilados usaremos el término *código objeto*, siendo este el código generado tras la compilación del código fuente. El código objeto es un *código máquina* o *bytecode* de forma que a partir de un código fuente escrito en un lenguaje natural, obtenemos código que la máquina es capaz de procesar de forma directa.
- Cada línea de código se suele llamar con frecuencia *instrucción*.

1.2. CONCEPTOS BÁSICOS.

1.2.5. PALABRAS RESERVADAS DEL LENGUAJE.

- Los lenguajes de programación usan una serie de palabras o símbolos que desempeñan una función específica dentro de un programa. A este conjunto o combinación de ellos se les denomina palabras reservadas.
- Estas palabras solo pueden usarse en un concepto determinado, nunca libremente por el usuario.
 - *Por ejemplo, el lenguaje de programación C tiene como palabras reservadas “IF” o “WHILE”. Ambas solo podrán ser usadas en caso de que se quiera iniciar una sentencia de control condicional o un bucle.*

1.3. DISEÑO DE UN PROGRAMA.

- A la hora de construir un programa se deben seguir una serie de pautas o fases:
 - Análisis del problema.
 - Diseño del algoritmo.
 - Codificación.
 - Prueba y depuración.
 - Documentación.

1.3. DISEÑO DE UN PROGRAMA.

1.3.1. FASE DE ANÁLISIS DEL PROBLEMA.

- En esta fase se debe analizar con detenimiento el problema que se plantea. Tener clara cuál es la finalidad y a quien va dirigido el programa.
- Debemos saber con exactitud qué elementos debe incluir y qué tareas debe realizar. Cuando el software es para uso personal, nosotros mismos especificamos las pautas y decidimos su funcionalidad, en cambio, cuando debe ser usado por otras personas, para conocer sus necesidades debemos entrevistarnos con ellos.
- La fase de análisis es una parte fundamental del proceso de desarrollo de un programa, una de las más importantes, ya que un mal análisis puede hacer que un programa ya codificado y probado deba realizarse de nuevo.

1.3. DISEÑO DE UN PROGRAMA.

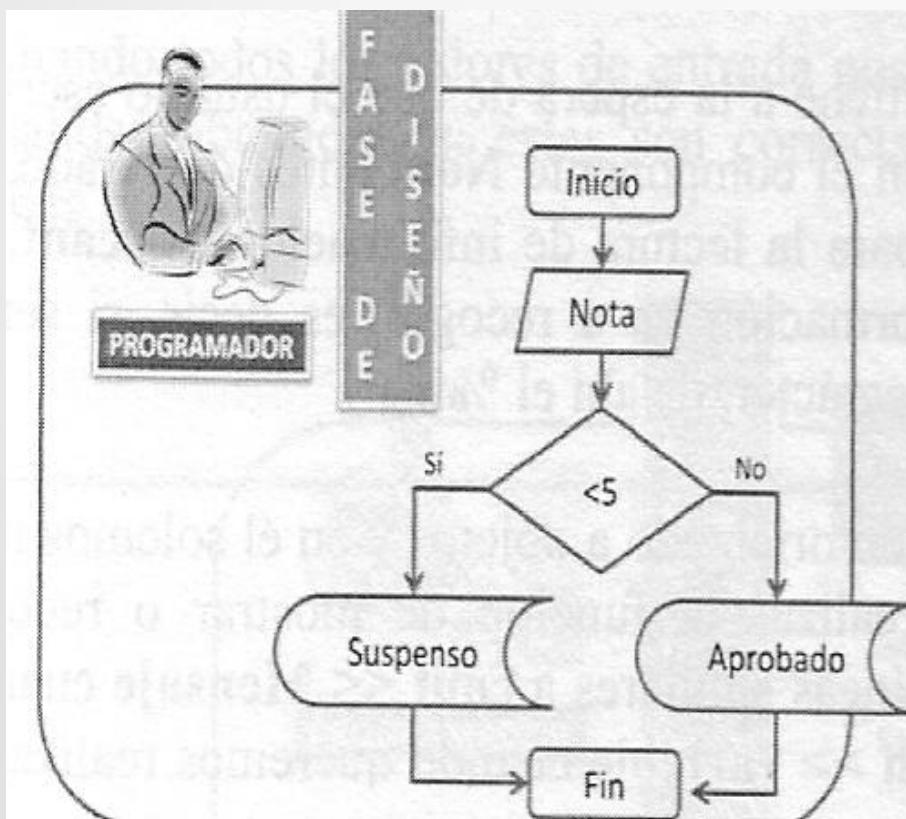
1.3.1. FASE DE ANÁLISIS DEL PROBLEMA.

- En la fase de análisis debemos identificar:
 - En qué tipo de ordenadores esperamos ejecutar el software.
Sistema operativo, microprocesador y memoria mínimos, etc. Hay que adaptar las tareas de codificación a los recursos que dispondremos.
 - A quién va dirigido el software.
Personas acostumbradas o no a utilizar aplicaciones y nuevas tecnologías.
 - Cuáles serán los datos de entrada.
 - Cuáles serán los datos de salida.
 - Cuáles serán las operaciones a realizar para que a partir de los datos de entrada obtengamos la salida que hemos analizado.

1.3. DISEÑO DE UN PROGRAMA.

1.3.2. FASE DE DISEÑO DEL ALGORITMO.

- El algoritmo representa de forma técnica cómo abordar el problema inicial, identificando soluciones a los problemas planteados, así como pasos de ejecución para concretar las tareas.
- En esta fase usaremos diagramas de flujo y pseudocódigo.

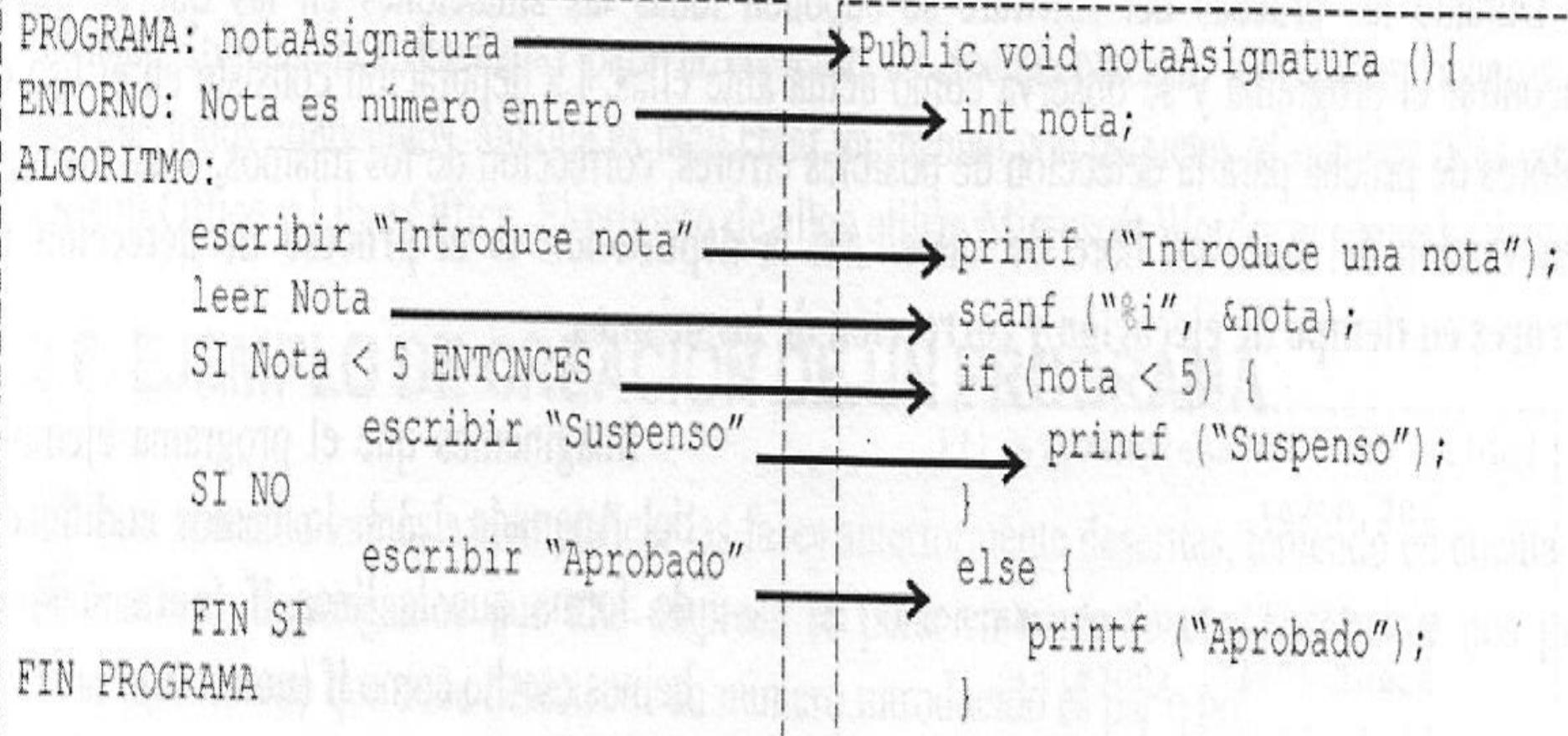


```
PROGRAMA: notaAsignatura
ENTORNO: Nota es número entero
ALGORITMO:
    escribir "Introduce nota"
    leer Nota
    SI Nota < 5 ENTONCES
        escribir "Suspens"
    SI NO
        escribir "Aprobado"
    FIN SI
FIN PROGRAMA
```

1.3. DISEÑO DE UN PROGRAMA.

1.3.3. FASE DE PROGRAMACIÓN (CODIFICACIÓN).

- Utilizamos un lenguaje de programación concreto de forma que el algoritmo anterior se convierta realmente en un programa que pueda ser ejecutado.



1.3. DISEÑO DE UN PROGRAMA.

1.3.4. FASE DE PRUEBAS Y DEPURACIÓN.

- Una vez el programa se crea debemos comenzar a probarlo antes de distribuirlo al usuario final. Si bien es cierto que cuando se crea un software, este se prueba repetidamente, se abordan los fallos más relevantes, pero cuando pasa a ser probado por el usuario final es posible que surjan nuevos fallos de los que no nos hemos percatado.
- Así, cuando se crea un programa y se saca al mercado es normal encontrar cada cierto tiempo actualizaciones del mismo, estas proporcionan nuevas funciones pero además subsanan errores no controlados.
- Durante las pruebas del software se suponen todas las situaciones en las que se puede encontrar el programa y se observa cómo actúa ante ellas. La depuración consiste en el uso de valores de prueba para la detección de posibles errores, corrección de los mismos, y así elaborar una versión del software libre de fallos. Así la depuración es el proceso de detección de errores en tiempo de ejecución y corrección de los mismos.

1.3. DISEÑO DE UN PROGRAMA.

1.3.5. FASE DE DOCUMENTACIÓN.

- Para finalizar el proceso de creación de un programa crearemos documentación relacionada con este. Es necesario que el usuario final conozca qué requisitos debe poseer el hardware, cómo podemos ejecutar el software (formato de ejecución o posibles parámetros a usar), o cuáles son los valores de entrada que acepta.
- Un tema que se suele olvidar y también es muy importante relativo a la documentación es la documentación del propio código del programa. Esta documentación no está dirigida al usuario final sino al programador que en un momento dado debe modificar, ampliar o corregir un código ya existente (y tal vez creado por otra persona).

1.3. DISEÑO DE UN PROGRAMA.

1.3.6. EJEMPLO.

FASE 1. ANÁLISIS DEL PROBLEMA

Concertamos una entrevista con la persona encargada en la empresa que pueda hablarnos sobre el tipo de software que necesita. En esta entrevista preguntaremos qué tipos de datos de entrada se deben usar, una vez se procese un dato qué salida se debe obtener, si se produce un error qué se espera visualizar, etc.

Tras el encuentro queda claro que:

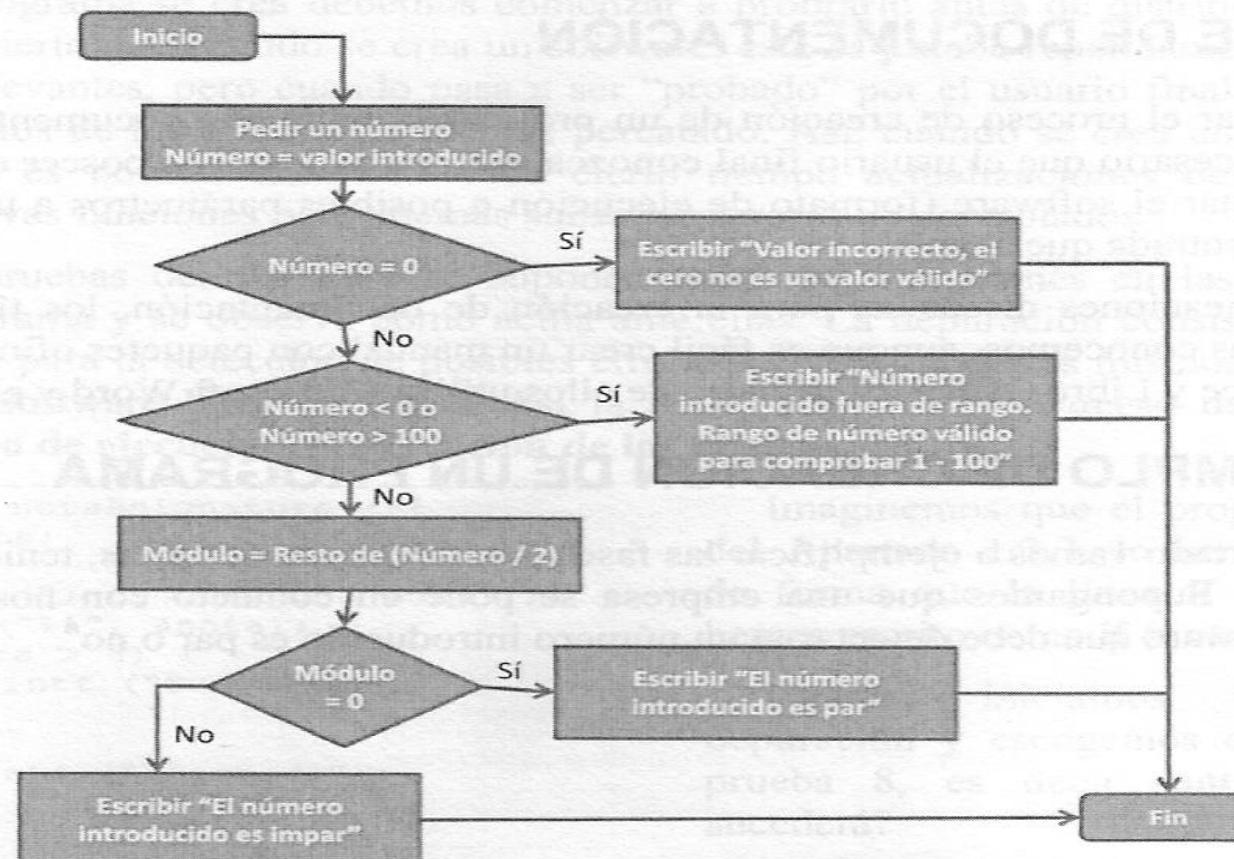
1. El programa solicitará un número. Solo se pueden comprobar los 100 primeros números. El cero no es un valor válido.
2. En caso de que el número sea par se visualizará en la pantalla “El número introducido es par”.
3. En caso de que el número sea impar se visualizará en la pantalla “El número introducido es impar”.
4. Si el número a comprobar es el cero se visualizará por pantalla el mensaje “Valor incorrecto, el cero no es un valor válido”.
5. Si el número introducido es menor que 0 o mayor de 100 se mostrará en pantalla el mensaje “Número introducido fuera de rango. Rango de número válido para comprobar 1 - 100”.

1.3. DISEÑO DE UN PROGRAMA.

1.3.6. EJEMPLO.

FASE 2. DISEÑO DEL ALGORITMO

Ya tenemos los datos necesarios para comenzar a trabajar. Sabemos qué desea el usuario final que ocurra en determinadas situaciones de ejecución así que construiremos el diagrama de flujo del programa y su pseudocódigo. Ejemplos de ambos elementos podrían ser los expuestos a continuación.



1.3. DISEÑO DE UN PROGRAMA.

1.3.6. EJEMPLO.

- (1) PROGRAMA: numeroPar
- (2) ENTORNO: Numero⁷, Modulo son números entero
- (3) ALGORITMO:
 - (4) escribir "Introduce un numero"
 - (5) leer Numero
 - (6) SI Numero = 0 ENTONCES
 - (7) escribir "Valor incorrecto, el cero no es un valor válido"
 - (8) SI Numero < 0 || Numero > 100 ENTONCES
 - (9) escribir "Número introducido fuera de rango. Rango de número válido para comprobar 1 - 100"
 - (10) SI NO
 - (11) Modulo = Resto de Numero/2
 - (12) SI Modulo = 0 ENTONCES
 - (13) escribir "El número introducido es par"
 - (14) SI NO
 - (15) escribir "El número introducido es impar"
 - (16) FIN SI
 - (17) FIN SI
- (18) FIN PROGRAMA

1.3. DISEÑO DE UN PROGRAMA.

1.3.6. EJEMPLO.

FASE 3. CODIFICACIÓN

La base anterior es una de las más complejas ya que concretamos exactamente las partes del algoritmo. En esta todo es más sencillo ya que solo debemos “traducir” el algoritmo a un lenguaje de programación concreto. En realidad esta fase será más o menos compleja en función del lenguaje de programación que usemos.

Veamos línea a línea como será el código de nuestro programa teniendo en cuenta que usaremos C++ como lenguaje de programación.

(1) PROGRAMA: numeroPar	void numeroPar() {
(2) ENTORNO: Número ⁸ , Modulo son números entero	int Numero, Modulo;
(3) ALGORITMO:	
(4) escribir “Introduce un numero”	printf ("Introduce un numero");
(5) leer Numero	scanf ("%i", Numero);
(6) SI Numero = 0 ENTONCES	if (Numero == 0) {
(7) escribir “Valor incorrecto, el cero no es un valor válido”	printf ("Valor incorrecto, el cero no es un valor válido");
(8) SI Numero < 0 Numero > 100 ENTONCES	}
(9) escribir “Número introducido fuera de rango. Rango de número válido para comprobar 1 - 100”	else if (Numero < 0 Numero > 100) {
	printf("Número introducido fuera de rango. Rango de número válido para comprobar 1 - 100");
(10) SI NO	}
(11) Modulo = Resto de Numero/2	Modulo = Numero mod 2;
(12) SI Modulo = 0 ENTONCES	if (Modulo == 0) {
(13) escribir “El número introducido es par”	printf ("El número introducido es par");
(14) SI NO	}
(15) escribir “El número introducido es impar”	else {
(16) FIN SI	printf ("El número introducido es impar");
(17) FIN SI	}
(18) FIN PROGRAMA	}

1.3. DISEÑO DE UN PROGRAMA.

1.3.6. EJEMPLO.

```
void numeroPar() {
    int Numero, Modulo;
    printf ("Introduce un numero");
    scanf ("%i",Numero);
    if (Numero == 0) {
        printf ("Valor incorrecto, el cero no es un valor válido");
    }
    else if (Numero < 0 || Numero > 100) {
        printf("Número introducido fuera de rango. Rango de número válido
               para comprobar 1 - 100");
    }
    else {
        Modulo = Numero mod 2;
        if (Modulo == 0){
            printf ("El número introducido es par");
        }
        else {
            printf ("El número introducido es impar");
        }
    }
}
```

1.3. DISEÑO DE UN PROGRAMA.

1.3.6. EJEMPLO.

FASE 4. PRUEBAS Y DEPURACIÓN

A partir de ahora comenzamos a ejecutar el software. Se debe comprobar que ante determinadas entradas las salidas son las esperadas. Las entradas que pueden presentar problemas son:

- Número 0.
- Número negativo.
- Número superior a 100.

Al ejecutar la aplicación visualizaremos un entorno en modo texto como el que se observa en la Figura 1.10.

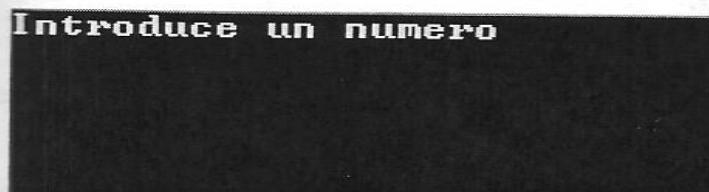


Figura 1.10. Programa sin entorno visual en ejecución.

Escribiremos un cero. Debe aparecer el texto “*Valor incorrecto, el cero no es un valor válido*”.



Figura 1.11. Ejecución del software y comprobación de las salidas en función de los valores de entrada.

Realizaremos este proceso con todos los valores de entrada posibles. Observaremos las salidas, y si son las que se plantearon en el algoritmo el programa estará listo para ser usado. Si se produce algún fallo, o alguna salida no es la esperada para el valor introducido volveremos al código fuente y localizaremos el lugar donde se produce el error en ejecución. Ya estudiaremos cómo visualizar los cambios que se dan en los valores introducidos durante la ejecución mediante la depuración en los diferentes entornos de programación que utilizaremos.

El código planteado realiza las funciones tal y como la empresa detalló de forma que pasaremos a la fase de documentación.

1.3. DISEÑO DE UN PROGRAMA.

1.3.6. EJEMPLO.

FASE 5. DOCUMENTACIÓN

Antes de entregar el software a la empresa que nos lo encargó realizaremos un breve manual sobre el mismo. Este podría ser similar al que se muestra a continuación.

USO DE PAR-IMPAR v1.0

Para ejecutar el programa escribiremos en la línea de comando **numeroPar**.

Veremos la siguiente imagen:

Introduce un numero

El rango de valores que podemos analizar es de 1 a 100.

El valor cero no se analiza, de forma que si lo escribimos se mostrará el mensaje de error:

“Valor incorrecto, el cero no es un valor válido”

Los valores negativos y superiores a 100 tampoco podrán ser analizados, si aparece el mensaje “*Número introducido fuera de rango. Rango de número válido para comprobar 1 - 100*” este será debido a que hemos introducido un número negativo o mayor que 100.

El programa mostrará el texto “*El número introducido es par*” o “*El número introducido es impar*” según el valor utilizado del rango 1 a 100 sea par o impar.

El programa acabará tras mostrar el mensaje de información adecuado.

1.4. DIAGRAMAS DE FLUJO.

- Un diagrama de flujo (o *organigrama*) es una representación gráfica de un algoritmo, de forma que se describen las diferentes tareas y la secuencia en la que se ejecutan para alcanzar una solución.
- En algoritmos complejos podemos usar más de un diagrama de flujo. Diseñaremos el diagrama de flujo principal y a partir de él iremos diseñando diagramas de cada tarea que se puede realizar a partir de la inicial.

1.4. DIAGRAMAS DE FLUJO.

- Reglas para diseñar diagramas de flujo:

- Siempre debemos incluir una forma con el texto de “Inicio” y otra con el texto de “Fin” para delimitar los momentos en que empieza y acaba el algoritmo.
- Debemos usar una serie de símbolos estándar.
- Los elementos del diagrama de flujo deben estar concatenados mediante flechas que indican la dirección de ejecución.
- Los diagrama de flujo deben escribirse de arriba hacia abajo o de izquierda a derecha.
- Debemos evitar el cruce de líneas, para eso se define la forma conector. El uso de conectores debe producirse cuando no exista otra opción.
- Todas las líneas de flujo deben estar conectadas a algún objeto.
- A la hora de escribir texto en las formas, este debe ser escueto y legible.
- Todos los símbolos de decisión deben tener más de una línea de salida, es decir, deben indicar qué camino seguir en función de la decisión tomada.

1.4. DIAGRAMAS DE FLUJO.

	Indican inicio o fin de programa.
	Representan una instrucción, un paso a dar o proceso. Por ejemplo: nota = 4, suma=suma+1, etc.
	Operaciones de entrada/salida de datos. Por ejemplo: visualiza en pantalla suma.
	Usaremos este símbolo si nos encontramos en un punto en el que se realizará una u otra acción en función de la decisión que el usuario tome.
	Conector. Permite unir diferentes zonas del diagrama de forma que se redefine el flujo de ejecución hacia otra parte del diagrama.
	Representa una función o subprograma.
	Conector de página. Se usa cuando llegamos al final de una página y aún no se ha acabado el diagrama de flujo.
	Disco magnético.
	Unidad de cinta magnética.
	Salida. Visualización de los datos de salida por pantalla. Este suele ser siempre el periférico de salida por defecto.
	Salida de datos por impresora.
	Flechas para indicar la dirección de flujo.

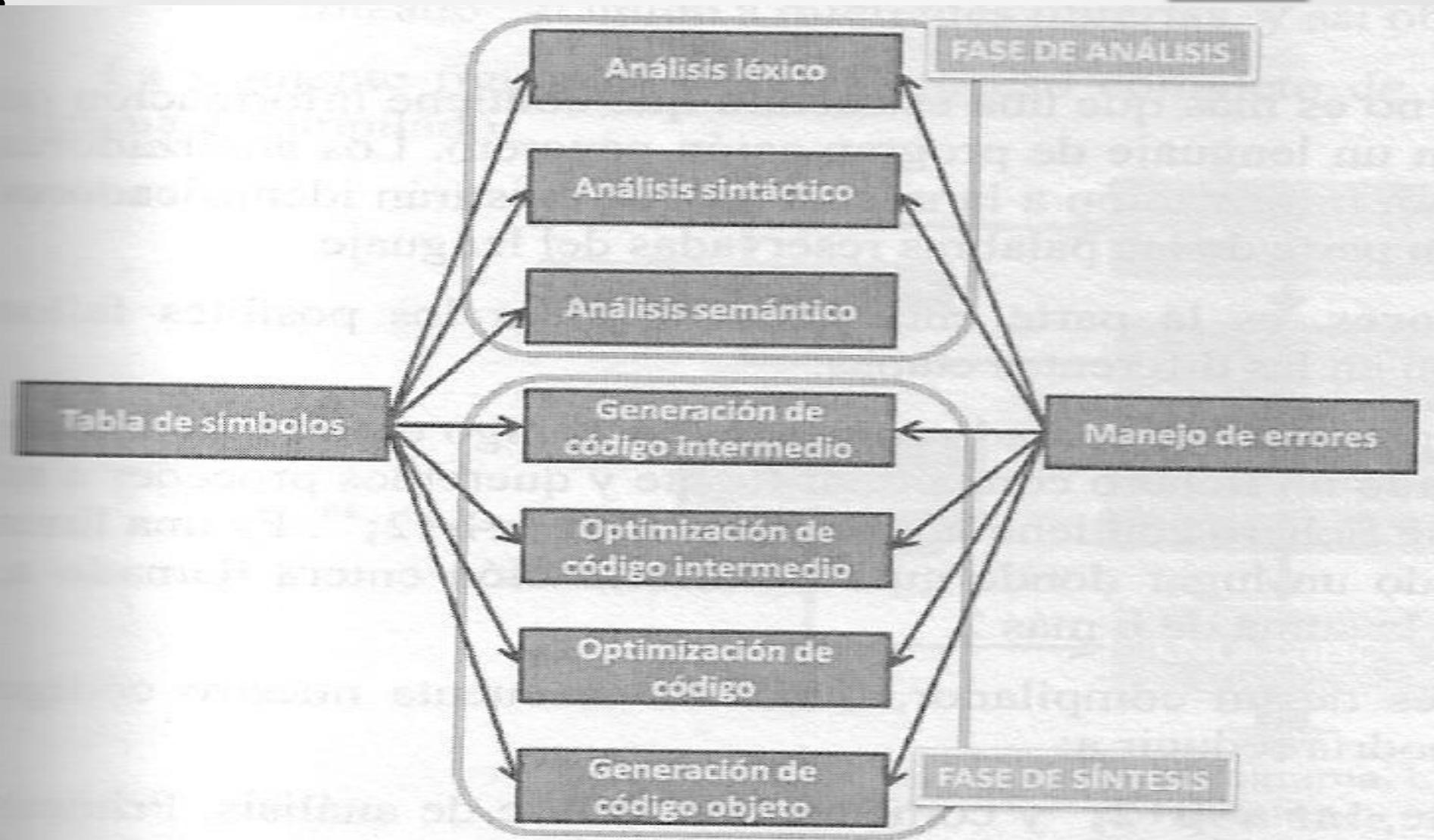
1.5. PSEUDOCÓDIGO.

- La palabra pseudocódigo puede ser traducida como falso lenguaje. Un pseudocódigo es una descripción informal de un programa, siendo de gran utilidad a la hora de diseñar el algoritmo del mismo.
- En pseudocódigo usamos lenguaje natural o similar para representar estructuras propias del lenguaje de programación que usaremos en la codificación, es decir, simulamos un lenguaje de programación.
- El pseudocódigo, aunque sea el lenguaje natural, no utiliza su amplia variedad de palabras, se seleccionan una serie de palabras como reservadas para representar todas las estructuras que posteriormente deberán ser codificadas mediante el lenguaje de programación.

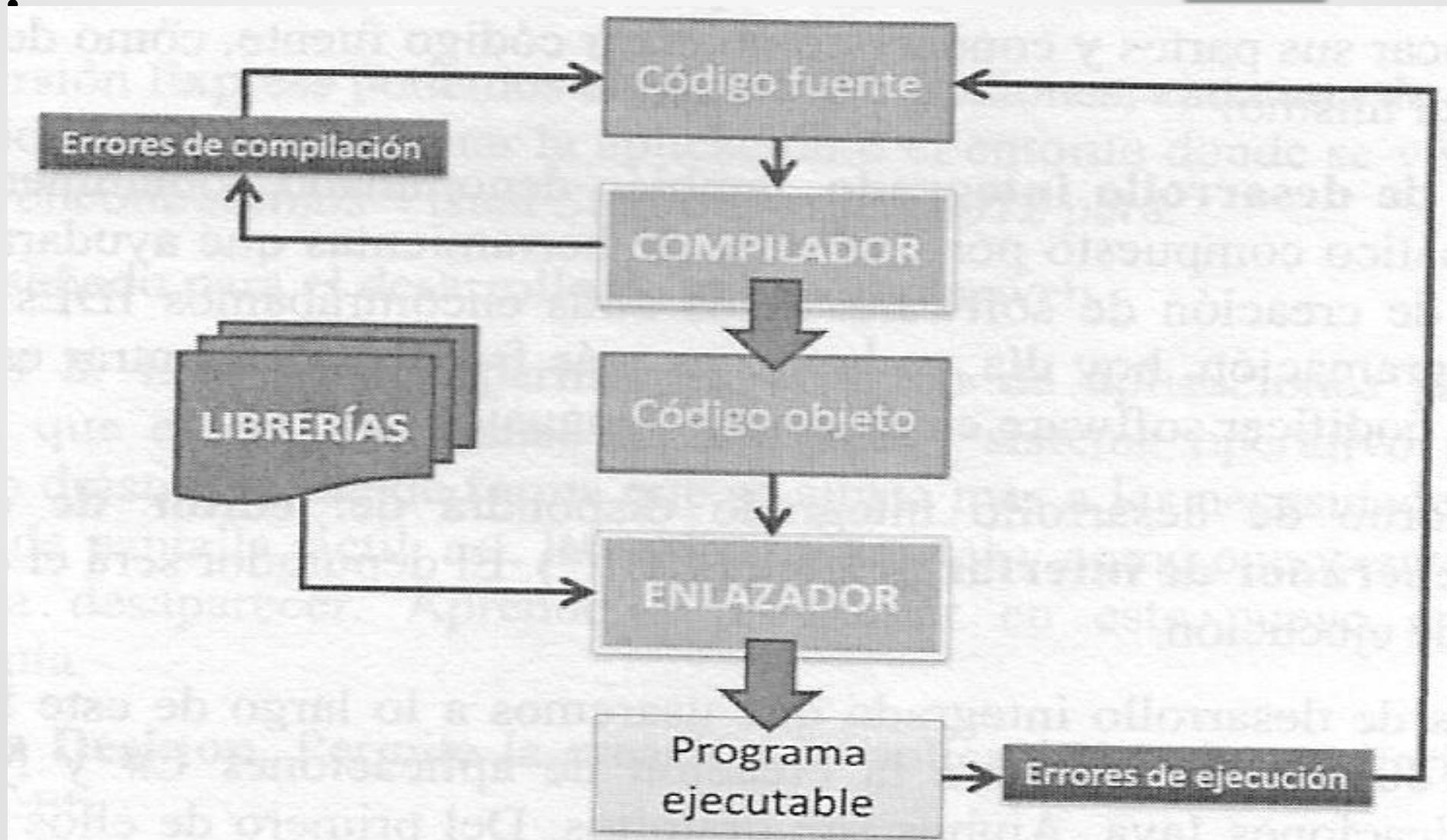
1.6. PROCESO DE COMPILEACIÓN.

- La palabra compilación se refiere a la traducción de un programa en código fuente a código máquina o *bytecode*. Compilador será el nombre de la herramienta software que es capaz de realizar la traducción.
 - Fase de análisis:
 - Análisis léxico. Comprueba que las “palabras” introducidas son correctas.
 - Análisis sintáctico. Comprueba si la estructura de las “frases” son propias del lenguaje.
 - Análisis semántico. Comprueba que las instrucciones tienen un significado correcto.
 - Fase de síntesis:
 - Generación de código intermedio. Permite ser usado por varios hardware.
 - Optimización de código. Mejora el código intermedio (rendimiento).
 - Generación de código objeto.
 - Tabla de símbolos (elementos del lenguaje) y manejador de errores.

1.6. PROCESO DE COMPILEACIÓN.



1.6. PROCESO DE COMPILEACIÓN.



1.6. PROCESO DE COMPILEACIÓN.

- Los lenguajes de programación actuales utilizan un modelo híbrido para la obtención del *código objeto*. Cuando generamos un *código máquina* tenemos en cuenta unas características hardware concretas, es decir, estamos obteniendo un código objeto para un procesador concreto. Así, si queremos que un *código fuente* sea ejecutado en varias arquitecturas será preciso compilar estas para cada una de ellas.
- Con el fin de ser más versátiles, los lenguajes de programación actuales generan un *código intermedio*. Exactamente crean una máquina virtual para la que desarrolla el código compilado que posteriormente será interpretado por el hardware real en el que se ejecutará.
- Es decir, existen lenguajes de programación híbridos tales que compilan el código fuente para obtener un código intermedio y este, es interpretado cuando pasa a ser ejecutado sobre una máquina concreta.

1.7. ENTORNOS DE DESARROLLO INTEGRADOS DE SOFTWARE O IDE.

- Un entorno de desarrollo integrado o IDE, es un software informático compuesto por una serie de herramientas que ayudarán al programador en todas las fases de creación de software.
- Antiguamente los IDEs estaban dedicados a un solo lenguaje de programación, hoy día son capaces de codificar software en diferentes lenguajes.
- Así un entorno de desarrollo integrado dispondrá de: *editor de código, compilador, depurador y generador de interfaz gráfica (GUI)*. El depurador será el encargado de detectar posibles fallos de ejecución.
- Ejemplos:
 - *Eclipse*.
 - *Oracle Netbeans*.
 - *Microsoft Visual Studio*.

TEMA 1

INTRODUCCIÓN A LA PROGRAMACIÓN

FIN