Jesús García García-Doncel
Javier Gómez Delgado

PROGRAMACIÓN EN JAVA

CONSTRUYE APLICACIONES ROBUSTAS Y MULTIPLATAFORMA

ESIC Editorial

Programación en Java

Construye aplicaciones robustas y multiplataforma

Madrid, 2026

Jesús García García-Doncel

Javier Gómez Delgado

Programación en Java

Construye aplicaciones robustas y multiplataforma

ESIC Editorial

Septiembre, *2026*

Programación en Java: Construye aplicaciones robustas y multiplataforma

Jesús García García-Doncel y Javier Gómez Delgado

Avda. de Valdenigriales, s/n
28223 Pozuelo de Alarcón (Madrid)
Tel.: 91 452 41 00
www.esic.edu/editorial
@EsicEditorial

ISBN: 978-84-1192-266-1
Depósito Legal: M-16747-2026

Diseño de cubierta: Zita Moreno Puig
Maquetación: Balloon Comunicación
Lectura: Balloon Comunicación
Impresión: Gráficas Dehon

Un libro de

ESIC Editorial

Impreso en España – *Printed in Spain*

Este libro ha sido impreso con tinta ecológica y papel sostenible.

Índice

1

Introducción a la programación

Objetivos de aprendizaje:

- Conocer qué es y el origen de la programación.
- Entender los principales paradigmas de programación existentes.
- Diferenciar entre cohesión y acoplamiento.
- Aprender los conceptos de programación con pseudocódigo.

Palabras clave: Ada Lovelace, Grace Hopper, cohesión, acoplamiento, módulo, paradigma.

1.1 ¿Qué es programación?

Se utiliza el término *programación* para designar la creación de un conjunto de instrucciones para que un ordenador realice una serie de tareas específicas y resuelva un problema concreto.

Este proceso implica escribir lo que se denomina **código fuente** en un determinado lenguaje de programación, que luego se convertirá en **código máquina**, de manera que el ordenador sea capaz de entenderlo.

La programación no solo es un conjunto de instrucciones, sino que también es un proceso creativo donde el programador puede optar por diferentes formas de resolver un mismo problema. Lo atractivo de la programación reside en sus infinitas posibilidades y en que no existe una única forma correcta de resolver un problema (aunque también existen muchas incorrectas).

Origen de la programación

La historia de la programación se extiende mucho antes de la existencia de los ordenadores. El primer sistema programable se remonta a fechas antes de Cristo, con un reloj astronómico denominado **mecanismo de Anticitera**.

A principios del siglo XIX (1804), el **telar de Jacquard** manejaba un sistema binario (0 y 1) con tarjetas perforadas que automatizaba la creación de patrones en tejidos.

Se considera a **Ada Lovelace** la primera programadora. En la década de 1840 escribió un programa para la máquina analítica de Charles Babbage. Este programa tenía secuencia de instrucciones, bifurcaciones y bucles que permitían calcular los números de Bernoulli.

En los años 40 del siglo XX, con el desarrollo de los primeros ordenadores, se comenzaron a realizar programas para ellos. Estos se escribían con una codificación matemática, y hay que esperar hasta el año 1952 cuando **Grace Hopper** creó el primer lenguaje de programación que se escribía con palabras en inglés, lo que facilitó la programación y popularizó los lenguajes independientes de la máquina. También fue esta programadora la que acuñó el término **"bug"** y **"debug"** cuando se ocasiona un error en un programa, ya que ella misma descubrió una polilla en los circuitos del ordenador en donde estaba trabajando. Esta polilla, produciendo un cortocircuito, era la que estaba causando un error en el programa.

A partir de aquí fueron surgiendo nuevos lenguajes de programación que permitían nuevas formas de enfocar la resolución de los problemas. En la década de 1990, con la explosión de la web, aparecen lenguajes como Java, JavaScript y Python, y en la actualidad siguen apareciendo nuevos lenguajes con un enfoque a la concurrencia, la seguridad y la facilidad de uso.

La programación continúa evolucionando, enfocándose en la inteligencia artificial, el aprendizaje automático, la computación en la nube y la computación cuántica.

1.2 Paradigmas de programación

Los paradigmas de programación son la forma de conceptualizar y estructurar la implementación del código en un lenguaje informático. Son filosofías que guían la escritura del código, ofreciendo una perspectiva única para resolver el problema planteado.

El uso de un paradigma no es excluyente, y una misma aplicación puede estar basada en diferentes paradigmas, incluso en el mismo lenguaje.

Estos paradigmas son importantes, ya que facilitan técnicas para resolver problemas, aumentan la comprensión y el mantenimiento del código y mejoran la comunicación entre el equipo de trabajo. Algunos de los paradigmas más importantes son:

Programación imperativa

Es el enfoque más antiguo y consiste en dar instrucciones detalladas para ejecutarlas en un orden específico. Solo posee tres estructuras de programación:

- Secuencial: las instrucciones se ejecutan una tras otra.
- Bifurcaciones: hay una condición que indica qué conjunto de instrucciones se ejecutarán.
- Repeticiones: permiten ejecutar repetidamente un conjunto de instrucciones.

Programación procedural

Basada en la programación imperativa, añade el concepto de procedimiento, también denominado función o rutina. Mejora la modularidad y la organización del código dividiendo este en fragmentos más pequeños y fáciles de entender.

El problema de este paradigma es que, al enfocarse en una secuencia de pasos, los programas pueden resultar difíciles de manejar cuando son grandes y complejos. También es un código más complicado de mantener, ya que puede tener efectos inesperados en otras partes del código.

Programación orientada a objetos (POO)

Este paradigma se centra en el uso de objetos. Estos son entidades que encapsulan datos (atributos) y comportamientos (métodos). Este paradigma promueve la modularidad y la reutilización del código mediante el uso de clases, que son las plantillas para la creación de objetos. La POO permite modelar una solución

a través de entidades reales (empleados, facturas, clientes…) definiendo sus características y las acciones que puede realizar. Este paradigma permite modelar entidades complejas similares al mundo real, creando soluciones más fáciles de entender. También, a través de la encapsulación limita el acceso a datos, reduciendo errores y efectos secundarios.

Programación declarativa

En la programación declarativa se especifica el resultado deseado para los datos, evitando dar detalles de los pasos exactos para lograrlo. Se enfoca en qué se quiere lograr en lugar de cómo. Un ejemplo son lenguajes como SQL, HTML, CSS, hasta Python también puede comportarse de esta manera.

1.3 Modularización: cohesión y acoplamiento

La modularización implica dividir un programa, aplicación o sistema en múltiples módulos independientes, donde cada uno de ellos funciona de forma autónoma. Esta práctica es fundamental en la ingeniería de software, ya que facilita la comprensión, el mantenimiento y la reutilización del código.

Los puntos clave de la modularización son los siguientes:

- **Independencia**: cada módulo debe trabajar de manera independiente, lo que significa que puede ser desarrollado, probado y modificado sin afectar a otros módulos. Esto reduce la complejidad del sistema, ya que permite trabajar en partes más pequeñas y manejables.
- **Reutilización**: un módulo puede ser utilizado varias veces dentro del mismo proyecto o en diferentes proyectos, lo que evita tener que escribir el mismo código una y otra vez.
- **Organización**: los módulos ayudan a organizar el código de manera lógica, agrupando funcionalidades relacionadas y facilitando el entendimiento del código.

La cohesión y el acoplamiento son dos conceptos fundamentales en la ingeniería del software que se utilizan para medir la calidad del diseño de los módulos de un sistema.

Cohesión

La cohesión se refiere al grado en que los elementos dentro de un módulo trabajan juntos para cumplir un único propósito definido.

Una **alta cohesión** significa que los elementos están estrechamente relacionados y enfocados en un solo propósito.

Una **baja cohesión** significa que los elementos están relacionados de manera débil y sirven a múltiples propósitos.

Es deseable que los módulos tengan una **alta cohesión**, esto mejora la legibilidad y ofrece un mejor aislamiento de los errores.

Acoplamiento

El acoplamiento se refiere al grado de interdependencia entre los módulos.

Un **alto acoplamiento** significa que los módulos están estrechamente conectados, de manera que los cambios en un módulo puedan afectar a otros.

Un **bajo acoplamiento** implica que los módulos son independientes y los cambios que se produzcan en uno no afectan al resto.

Es deseable que los módulos tengan un **bajo acoplamiento**, mejorando la independencia del resto de módulos.

Si tenemos una baja cohesión y alto acoplamiento, esto dificulta el mantenimiento y la prueba del sistema; al contrario, si la aplicación tiene una alta cohesión y un bajo acoplamiento, será un sistema fácil de mantener y comprensible para el resto de los programadores.

1.4 Pseudocódigo

El pseudocódigo es un lenguaje sencillo que se asemeja a un lenguaje de programación, pero sin la rigidez de una sintaxis y semántica formales, de manera que estas reglas son flexibles. Normalmente se usa el lenguaje natural para escribir las sentencias.

Características

- Es un lenguaje informal que permite a los programadores expresar los pasos de un algoritmo de manera concisa.
- No requiere una sintaxis estricta, lo que facilita su escritura y comprensión, asemejándose al lenguaje humano.
- Se utiliza para aprender y planificar la lógica y las estructuras de un programa antes de escribir el código en un lenguaje de programación específico.
- Es fácil de convertir a un lenguaje de programación para crear un programa.

No existe una sintaxis estandarizada para la escritura de pseudocódigo, con lo que es posible encontrar diferencias entre dos pseudocódigos escritos por diferentes programadores.

1.5 PseInt

`PseInt` es un software educativo que permite interpretar los algoritmos diseñados en pseudocódigo como si se tratara de un lenguaje de alto nivel, y posee la flexibilidad necesaria para iniciarse en la programación.

Pseint.sourceforge.net
Página oficial de PseInt

Para descargar esta herramienta se debe ir a su página oficial (**https://pseint.sourceforge.net/**) y bajar la versión adecuada para el sistema operativo, Windows, macOS o Linux.

La instalación es sencilla, dando al botón siguiente hasta finalizar el proceso.

En el primer arranque de la aplicación requiere la selección de un perfil. Lo más sencillo es seleccionar la `"Opción 3: No seleccionar perfil"`

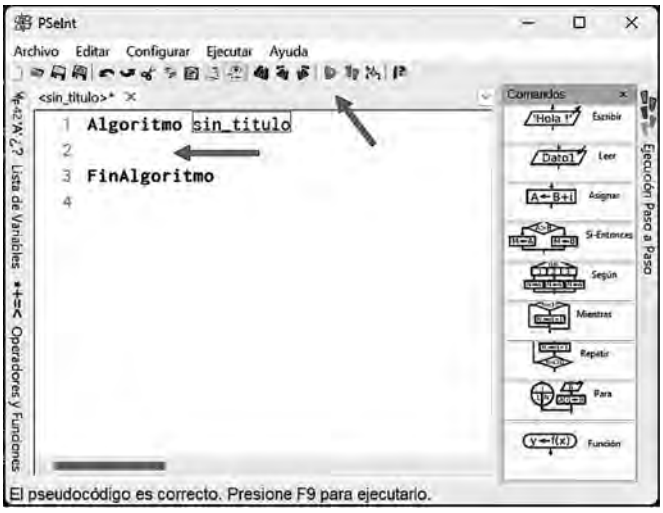


Figura 1.1
Pantalla inicial PseInt

Si se quieren cambiar ciertas características del pseudocódigo que se escriba, se puede entrar en la opción Configurar/Opciones del Lenguaje/Personalizar.

Aparecerá una ventana en donde se podrá cambiar características como "Permitir asignar con el signo igual(=)"

Figura 1.2
Configuración PseInt

En la pantalla principal, entre las líneas `Algoritmo` y `FinAlgoritmo` escribiremos el código. Para ejecutarlo, pulsaremos en el triángulo verde de la barra superior.

Conceptos básicos

Variable: es el espacio en memoria donde se almacenan y se recuperan los datos de un programa. Estas variables pueden representar cualquier información, como el nombre de un cliente o las ventas semanales de un comercial.

Todas las variables tienen un nombre y un tipo.

Nombre de variable

Es el nombre que permite identificar y trabajar con una variable. Cada lenguaje de programación posee unas reglas para la asignación de nombres, pero por línea general deben empezar por una letra, aunque puede llevar números, no pueden contener espacios ni puntos, y siempre se aconseja poner nombres significativos, relacionados con la información que contiene, por ejemplo, nombre, edad, etc.

Tipo de variable

Este tipo se refiere a qué clase de información es capaz de almacenar la variable. Hay dos grandes grupos de tipos de variables: numéricos y no numéricos.

El tipo numérico, a su vez, se divide en enteros, sin decimales, y reales, con decimales.

El tipo no numérico se subdivide en caracteres, lógicos, objetos, etc.

Dependiendo del lenguaje de programación, puede haber más tipos que puedan especificar diferentes tamaños.

Constante

Es el nombre que se le asigna a un dato que no puede ser modificado en el programa.

Instrucción

Una instrucción es una orden que el programador da al programa indicándole la acción que debe realizar.

1.6 Instrucciones en PseInt

Escribir

Esta instrucción permite escribir en la pantalla un texto o el valor de una variable.

Leer

Se utiliza para indicar al programa que pida un dato. Este debe ser introducido por el teclado. La información se guarda en una variable cuyo nombre se escribe a continuación de la instrucción.

Código 1.1
Lectura y escritura

```
Algoritmo sin_titulo
     Escribir "Dime un nombre:"
     leer nombre
     Escribir "El nombre es: " , nombre
FinAlgoritmo
```

Comentarios

Se denomina comentario a aquel texto dentro del programa que no va a ser ejecutado, y que se utiliza para explicar el funcionamiento. En `PseInt` se utiliza la doble barra diagonal (//) antes de especificar el texto.

Código 1.2
Comentario

```
Algoritmo sin_titulo
     //Pide un nombre y lo guarda en nombre
     leer nombre
     Escribir "El nombre es: " , nombre
FinAlgoritmo
```

Definición de tipos de variables

Aunque no es obligatorio definir el tipo de las variables utilizadas en `PseInt`, es posible definirlas.

`Real`: permite almacenar números con decimales.

`Entero`: almacena números sin decimales.

`Carácter`: se utiliza para guardar letras o frases.

`Lógico`: guarda solo los valores VERDADERO o FALSO.

Código 1.3
Tipos de variables

```
Algoritmo sin_titulo
     a=23
     escribir a
     nombre = "Javi"
     escribir nombre

     definir var1 Como Real
     definir var2 Como Entero
     definir var3 Como Caracter
     definir var4 Como Logico
     var1 = 4.5
     var2 = 3
     var3="cadena"
     var4=Verdadero
     escribir var1, var2, var3, var4
FinAlgoritmo
```

Expresiones matemáticas

En los lenguajes de programación se pueden realizar operaciones matemáticas, asignando los resultados a variables. Para ello se utilizan los símbolos matemáticos y el operador igual (=) para asignarlo.

Código 1.4
Expresión matemática

```
Algoritmo sin_titulo
     leer num1
     leer num2
     sum=num1 + num2
     escribir "Resultado: ", sum
FinAlgoritmo
```

Operadores

A continuación, se muestran los operadores que se pueden utilizar en este pseudocódigo.

Aritméticos		Relacionales		Lógicos	
+	Suma	=	Igual	Y	Conjunción
-	Resta	!=	Diferente	O	Disyunción
*	Multiplicación	<	Menor	NO	Negación
/	División	<=	Menor o igual		
^	Potencia	>	Mayor		
MOD	Resto de div.	>=	Mayor o igual		

Funciones matemáticas			
abs	Valor absoluto	rc	Raíz cuadrada
trunc	Valor truncado	exp	Exponencial
redon	Valor redondeado	azar	Núm. aleatorio

Arrays

Los arrays son estructuras de datos homogéneos (del mismo tipo) que pueden contener un determinado número de información bajo un mismo identificador. Para poder trabajar con cada uno de los datos, se identifican a través de un índice.

Antes de utilizar un array, lo primero que se tiene que realizar es decir cuántos elementos es capaz de almacenar. Esto se hace a través de la instrucción Dimensionar.

Código 1.5
Arrays

```
Dimensionar nombreArray[cantidad de elementos]

Algoritmo bucle
     Dimensionar a[3]
     Escribir "Dato 1"
     Leer a[1]
     Escribir "Dato 2"
     Leer a[2]
     Escribir "Dato 3"
     Leer a[3]
     Escribir a[1],"-",a[2],"-",a[3]
FinAlgoritmo
```

Estas estructuras son muy utilizadas para almacenar grandes cantidades de información y poder trabajar con ellas de forma masiva, sin tener que declarar variables diferentes para cada uno de los datos.

1.7 Estructuras de control, bifurcaciones

Estas estructuras ejecutan una serie de instrucciones u otras, dependiendo de una condición determinada.

Bifurcación simple

En el caso que se cumpla la condición, ejecuta una serie de instrucciones.

```
Si <condición> entonces
   <instrucción 1>
   <instrucción n>
Fin Si

Algoritmo sin_titulo
     Escribir "Introduce tu edad"
     leer edad
     si edad < 18
          escribir "Eres menor de edad"
     FinSi
FinAlgoritmo
```

Código 1.6
Bifurcación simple

Bifurcación doble

Si se cumple la condición, ejecuta una serie de instrucciones, y si no se cumple, ejecuta otras.

```
Si <condición> entonces
   <instrucción 1>
SiNo
   <instrucción 2>
Fin Si

Algoritmo sin_titulo
     Escribir "Introduce tu edad"
     leer edad
     si edad < 18
          escribir "Eres menor de edad"
     SiNo
          escribir "Eres mayor de edad"
     FinSi
FinAlgoritmo
```

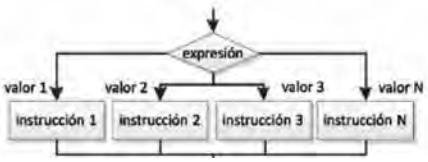
Código 1.7
Bifurcación doble

Dentro de las instrucciones que se ejecutan en la bifurcación puede haber otra condición. De esta manera se construyen condiciones anidadas.

Alternativa múltiple

La última estructura de control es la alternativa múltiple. Estas instrucciones permiten ejecutar diferentes bloques de instrucciones dependiendo del valor que tome una expresión.

```
Según <expresión> hacer
       Caso <valor1>
           <instrucciones1>
       Caso <valor2>
           <instrucciones2>
       Caso <valor3>
           <instrucciones3>
       Otro caso
           <instruccionesN>
  Fin Según
```



Código 1.8
Alternativa múltiple

```
Algoritmo sin_titulo
     Escribir "Introduce una opción:"
     Leer opcion
     Segun opcion Hacer
          1:
               Escribir "Opción 1"
          2:
               Escribir "Opción 2"
          3:
               Escribir "Opción 3"
          De Otro Modo:
               Escribir "No es ningún valor anterior"
     Fin Segun
FinAlgoritmo
```

1.8 Estructuras iterativas

Son aquellas que permiten la ejecución de un bloque de instrucciones mientras se cumpla una condición. Que se verifique o no una condición se traduce en que una expresión lógica tome el valor VERDADERO o el valor FALSO.

Estructura mientras

Primero se comprueba la expresión, y mientras que la condición sea verdadera, se seguirá ejecutando el bloque de instrucciones.

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

Código 1.9
Mientras

```
Algoritmo sin_titulo
     num=0
     Mientras num<10
```

```
          Escribir num
          num=num+1
     FinMientras
 Escribir "Números del 0 al 9"
FinAlgoritmo
```

Puede ocurrir que las instrucciones de dentro de la sentencia **mientras** no se ejecuten nunca si la condición es falsa la primera vez que se evalúa. En el caso de que la condición siempre sea verdadera, no parará de ejecutar el bucle, y acabará fallando el programa.

Estructura repetir-hasta

Aquí se ejecuta un bloque de instrucciones hasta que la condición sea verdadera. A su vez, estas instrucciones se ejecutan siempre una vez, y después de esta primera ejecución se evalúa la condición. Si la condición es falsa, se vuelven a ejecutar las instrucciones en un nuevo ciclo. Cuando la condición es verdadera, sale del bucle.

```
Repetir
     secuencia_de_acciones
Hasta Que expresion_logica
```

Código 1.10
Repetir hasta

```
Algoritmo notas
     Repetir
          Escribir "Introduce una nota
 entre 0 y 10"
          leer nota
     Hasta Que nota>=0 y nota<=10
FinAlgoritmo
```

Iteración con contador. Estructura para

Esta estructura ejecuta un bloque de instrucciones un número determinado de veces.

```
Para <variable>=<inicial> Hasta <final> Con Paso <paso>
     instrucciones
Fin Para
```

Usa una **<variable>** que recibe un valor **<inicial>.** Mientras que no se cumpla que la variable llegue al valor **<final>** se ejecutan las instrucciones del bucle. Una vez que ha finalizado la ejecución, se incrementa **<variable>** con la cantidad expresada en **<paso>** y se vuelve a comprobar si ha llegado al valor **<final>**

Si se omite la cláusula Con paso **<paso>**, la variable se incrementará en 1.

```
Algoritmo bucle
     Para i=1 Hasta 10 Con Paso 1
          Escribir i
```

Código 1.11
Para

```
    Fin Para

    Para i desde 1 hasta 10
        Escribir i
    FinPara

    para i=10 hasta 1 con Paso -1
        Escribir i
    FinPara

    Para i desde 10 hasta 1
        Escribir i
    FinPara
FinAlgoritmo
```

Iteración de elementos. Estructura para cada

Esta estructura recorre un array (conjunto de valores con un mismo nombre) y trata cada uno de los elementos hasta que no haya más en la estructura.

```
Para Cada <elemento> de <array>
    <instrucciones>
FinPara
```

Código 1.12
Para cada

```
Algoritmo bucle
    Dimensionar a[3]
    Para i=1 Hasta 3 Hacer
        Escribir "Dato ", i
        leer a[i]
    FinPara
    Para Cada ele de a
        escribir "El dato es: " , ele
    FinPara
FinAlgoritmo
```

1.9 Funciones

En pseudocódigo también es posible realizar una descomposición en módulos o funciones, agrupando bajo un nombre un conjunto de instrucciones que pueda ser ejecutado en cualquier punto del programa principal.

Para crear este módulo, se pone la palabra clave `Funcion` y a continuación el nombre que tendrá este. Será necesario finalizar con la palabra clave `FinFuncion`.

Para llamar a una función, solo es necesario mencionar su nombre. Su estructura completa es la siguiente:

```
Funcion var_ret <- nom_funcion ( arg_1, arg_2, ... )
          instrucciones
FinFuncion
```

Funciones que no reciben ni devuelven valores

Estas funciones ejecutan un conjunto de instrucciones sin tratar ninguna información de entrada, y tampoco devuelven ningún valor.

Código 1.13
Función básica

```
Funcion Muestra
     escribir "Dentro de la función"
FinFuncion

Algoritmo funciones
     Muestra
FinAlgoritmo
```

Funciones que reciben valores

Los valores que se quieran pasar a la función se especifican entre paréntesis después del nombre de la función.

Código 1.14
Función con entrada de datos

```
Funcion Saluda(nombre)
     escribir "Hola " , nombre
FinFuncion
Algoritmo funciones
     Escribir "Nombre: "
     leer nom
     Saluda(nom)
FinAlgoritmo
```

Funciones que devuelven valores

Una función puede necesitar devolver al programa principal un valor que ha calculado dentro de sus instrucciones. Para ello se debe especificar el nombre de la variable a devolver antes del nombre de la función.

Código 1.15
Función con entrada y salida de datos

```
Funcion res<-SumaNumeros(num1,num2)
     res=num1+num2
FinFuncion
Algoritmo funciones
     Escribir "Valor 1: "
     leer numero1
     Escribir "Valor 2: "
     leer numero2
     suma=SumaNumeros(numero1,numero2)
     Escribir "La suma es: ", suma
FinAlgoritmo
```

1.10 Ejercicios

Ejercicio 1.1: Escribir en `PseInt` un programa que pida al usuario dos datos y muestre la suma de ambos.

Ejercicio 1.2: Pedir al usuario la edad, el peso y la estatura en centímetros y calcular el índice de masa corporal. IMC = peso/(altura^2). En la fórmula, la altura se expresa en metros.

Ejercicio 1.3: Pedir un número por teclado y decir si es par o impar.

Ejercicio 1.4: Pedir por pantalla el radio de un círculo y calcular la circunferencia `(2*PI*r)` y el área `(PI*r^2)`

Ejercicio 1.5: Pedir tres precios y mostrar la media.

Ejercicio 1.6: Crear un programa que pida un tiempo en segundos y muestre cuántas horas, minutos y segundos son.

Ejercicio 1.7: Pedir tres valores por pantalla y determinar cuál es el mayor y cuál es el menor.

Ejercicio 1.8: Crear un menú con las opciones:

1. Alta empleado.
2. Borrar empleado.
3. Mostrar empleados.
4. Salir.

Mostrar un mensaje indicando que opción ha pulsado el usuario

Ejercicio 1.9: Mostrar la tabla de multiplicar del 5.

Ejercicio 1.10: Mostrar todas las tablas de multiplicar, desde la tabla del 1 a la tabla del 10.

Ejercicio 1.11: Pedir 10 números por pantalla. El programa debe decir cuántos números de los introducidos son pares y cuántos son impares.

Ejercicio 1.12: Solicitar por pantalla la cantidad de alumnos que hay en una clase. A continuación, introducir la nota de cada uno de los alumnos. Mostrar la nota media de la clase.

Ejercicio 1.13: Crear un programa que pida el número de niños y el número de niñas que hay en una clase y mostrar el porcentaje de ambos.

Ejercicio 1.14: Pedir por pantalla un número del 1 al 10 y mostrar la tabla de multiplicar de ese número. No dejar de pedir el número hasta que este no sea correcto.

Ejercicio 1.15: Pedir al usuario dos números. Mostrar todos los números impares entre el primer número introducido y el segundo. No dejar de pedir los números hasta que el primer número sea menor que el segundo.

Ejercicio 1.16: Solicitar el precio de un producto y la cantidad que el cliente ha comprado de ese producto. Si la suma de todos los productos pasa de 1.000 se le hará un 10% de descuento. Calcular el total de la factura que tiene que pagar teniendo en cuenta que hay que sumarle el 21% de IVA.

Ejercicio 1.17: Dado un número menor de 10.000, determinar la suma de sus dígitos. El programa debe validar que el valor introducido es menor de 10.000, en caso contrario volver a solicitar el número.

Ejercicio 1.18: Cargar un array con 10 números aleatorios entre el 1 y el 6. Mostrar todos los números y la media aritmética.

Ejercicio 1.19: Solicitar 5 números por teclado. Al finalizar, mostrar los números introducidos del último al primero.

Ejercicio 1.20: Pedir nombres de alumnos de una clase. No dejar de pedirlos hasta que el usuario escriba "FIN".

Ejercicio 1.21: Realizar un reloj digital que nunca pare. También se puede hacer que espere un segundo real para darle más realismo. Utilizar la función "Esperar 1 segundos".

Ejercicio 1.22: Se pide al usuario un tamaño y se debe pintar un cuadrado de asteriscos de esa cantidad introducida.

Ejemplo:

```
Tamaño: 4
  ****
  *  *
  *  *
  ****
```

Ejercicio 1.23: Algoritmo que lea un número (altura) y a partir de él cree una escalera invertida de asteriscos con esa altura.

Ejemplo:

```
Altura: 5
  *****
   ****
    ***
     **
      *
```

Ejercicio 1.24: Programa que almacene una agenda. El menú es el siguiente:

- En la opción 1, pedir por pantalla un nombre y un teléfono y que lo guarde en un array.
- En la opción 2, pedir un nombre y borrar del array ese nombre y teléfono.
- En la opción 3, mostrar todos los contactos almacenados.
- En la opción 4, salir del programa.